Ars Digitale
Engineering Notes Series

JAVA 21 ENGINEERING GUIDE

Linguaggio Java, architettura
e buone pratiche

Alessandro Fabri

Edizione 2026

info@ars-digitale.com

Guida di Ingegneria Java 21

Alessandro Fabri

Ars Digitale

Tutti i diritti riservati.

Guida di Ingegneria Java 21

Una guida di studio strutturata per lo sviluppo Java moderno e la certificazione

Alessandro Fabri

2026 Edition

Contact: info@ars-digitale.com

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

Copyright

Guida di Ingegneria Java 21
Autore: **Alessandro Fabri**

Tutti i diritti riservati.

Questo libro è fornito per studio personale, formazione e uso educativo.

Versione: 1.0
Lingua: Italiano

Contact: info@ars-digitale.com

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

2026 Edition

Prefazione

Questa guida è pensata come un percorso strutturato e pratico per studiare Java 21, con particolare attenzione a chiarezza, correttezza e comprensione duratura.

Il materiale è organizzato in moduli, dai fondamenti del linguaggio fino alle API, alla concorrenza, all'I/O e al Java Platform Module System.

Ogni capitolo è progettato sia come parte di un percorso progressivo sia come riferimento tecnico autonomo.

Informazioni su questo libro

Questo libro è stato progettato come un compagno di studio tecnico per Java 21.

L'obiettivo è combinare:

- progressione strutturata
- spiegazioni concise
- precisione tecnica
- chiarezza orientata alla certificazione
- utilità duratura come riferimento

L'edizione EPUB è ottimizzata per la lettura digitale e la navigazione per capitoli.

Contact: info@ars-digitale.com

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

🌍 **Lingua:** English | Italiano | Français

Indice del corso (Java 21)

Module 00 — Prerequisites & Setup

- Materiale propedeutico
- Eclipse shortcuts

Module 01 — Java Language Basics

- 1. Mattoni Sintattici di Base
- 2. Mattoni di base del linguaggio Java
- 3. Regole di naming Java
- 4. Tipi di dato Java e casting
- 5. Operatori Java
- 6. Istanziazione dei tipi

Module 02 — Control Flow

- 7. Flusso di controllo
- 8. Costrutti di iterazione in Java

Module 03 — Core Standard APIs

- 9. Stringhe in Java
- 10. Array in Java
- 11. Matematica in Java
- 12. Data e ora in Java
- 13. Formattazione e Localizzazione in Java

Module 04 — Object-Oriented Fundamentals

- 14. Metodi, Attributi e Variabili
- 15. Caricamento delle Classi, Inizializzazione e Costruzione degli Oggetti
- 16. Ereditarietà in Java
- 17. Oltre le Classi
- 18. Generics in Java
- 19. Eccezioni e Gestione degli Errori

Module 05 — Functional Programming

- 20. Programmazione Funzionale in Java
- 21. Java Optional e Streams

Module 06 — Collections Framework

- 22. Introduzione al Framework delle Collezioni
- 23. Operazioni Condivise delle Collezioni & Uguaglianza
- 24. Comparable, Comparator & Ordinamento in Java
- 25. L'API List
- 26. Set API
- 27. API Queue & Deque
- 28. Map API
- 29. Collezioni Sequenziate & Map Sequenziate

Module 07 — Concurrency and Threads

- 30. Thread Java – Fondamenti e Modello di Esecuzione
- 31. Java Concurrency APIs

Module 08 — Java I/O and NIO

- 32. Fondamenti di File e Path
- 33. API di Files e Path
- 34. Stream I/O in Java
- 35. API di I/O Java (Legacy e NIO)
- 36. Interagire con l'Utente (Stream I/O Standard)

Module 09 — Java Platform Module System (JPMS)

- 37. Java Platform Module System (JPMS)
- 38. Compilare, Impacchettare ed Eseguire Moduli
- 39. Servizi in JPMS (Il Modello ServiceLoader)

◀ | ▲ Index | Prerequisite material for the course ▶

Module 00

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

◀ Indice del corso (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. Mattoni Sintattici di Base ▶

Module 01

Java Language Basics

1. Mattoni Sintattici di Base

Indice

- 1.1 Valore
- 1.2 Letterale
- 1.3 Identificatore
- 1.4 Variabile
- 1.5 Tipo
- 1.6 Operatore
- 1.7 Espressione
- 1.8 Istruzione
- 1.9 Blocco di Codice
- 1.10 Funzione / Metodo
- 1.11 Classe / Oggetto
- 1.12 Modulo / Package
- 1.13 Programma
- 1.14 Sistema
- 1.15 Riepilogo come scala crescente
- 1.16 Diagramma gerarchico ASCII
- 1.17 Diagramma gerarchico Mermaid

Ogni sistema software o programma informatico è composto da un insieme di **dati** e da un insieme di **operazioni** che vengono applicate a questi ultimi per produrre un risultato.

Più formalmente:

> Un programma informatico consiste in una collezione di strutture dati che rappresentano lo stato del sistema, insieme ad algoritmi che specificano le operazioni da eseguire su questo stato per produrre degli output.

Questo documento descrive una **gerarchia di astrazioni**: i *mattoni elementari* che, combinati in strutture via via più complesse, formano il software.

La sequenza è presentata in **ordine crescente di complessità**, con definizioni generali (informatica) e riferimenti a Java.

1.1 Valore

- **Definizione:** Entità astratta che rappresenta informazione (numero, carattere, boolean, stringa, ecc.).
- **Teoria:** Un valore appartiene a un dominio (insieme) matematico, come $\mathbb{N}$ per i numeri naturali o Σ^* per le stringhe.
- **Esempio (astratto):** il numero quarantadue, il valore di verità *true*, il carattere “a”.

Esempio Java (valori):

```
// Questi sono valori:
42        // un valore int
true      // un valore boolean
'a'       // un valore char
"Hello"   // un valore String
```

1.2 Letterale

- **Definizione:** Un **letterale** è la notazione concreta nel codice sorgente che denota direttamente un valore fisso.
- **In Java:** `42`, `'a'`, `true`, `"Hello"`.
- **Teoria:** Un letterale è **sintassi**, mentre il valore è la sua **semantica**.
- **Nota:** I letterali sono il modo più comune per introdurre valori nei programmi.

Esempio Java (letterali):

```
int answer = 42;          // 42 è un letterale int
char letter = 'a';        // 'a' è un letterale char
boolean flag = true;      // true è un letterale boolean
String msg = "Hello";     // "Hello" è un letterale String
```

1.3 Identificatore

- **Definizione:** Un nome simbolico che associa un valore (o una struttura) a un'etichetta leggibile.
- **In Java:**
  - **Identificatori definiti dall'utente:** scelti dal programmatore per nominare variabili, metodi, classi, ecc.
    Esempi: `x`, `counter`, `MyClass`, `calculateSum`.
  - **Parole chiave (keyword, riservate):** nomi predefiniti riservati dal linguaggio Java che non possono essere ridefiniti.
    Esempi: `class`, `public`, `static`, `if`, `return`.

> **Note**
>
> Gli identificatori devono rispettare le regole di naming di Java: vedi [Regole di naming Java]

- **Teoria:** Funzione di binding: collega un nome a un valore o a una risorsa.

Esempio Java (identificatori):

```
int counter = 0;          // counter è un identificatore (nome di variabile)
String userName = "Bob";  // userName è un identificatore
class MyService { }       // MyService è un identificatore di classe
```

1.4 Variabile

- **Definizione:** Una "cella di memoria" etichettata da un identificatore, che può contenere e cambiare valore.
- **In Java:** `int counter = 0; counter = counter + 1;`.
- **Teoria:** Uno stato mutabile che può variare nel tempo durante l'esecuzione.

Esempio Java (variabile che cambia nel tempo):

```
int counter = 0;        // variabile inizializzata
counter = counter + 1; // variabile aggiornata
counter++;              // altro aggiornamento (post-incremento)
```

1.5 Tipo

- **Definizione:** Un tipo è un insieme di valori e un insieme di operazioni consentite su tali valori.
- **In Java:**
  - **Tipi primitivi (semplici):** rappresentano direttamente valori di base.
    Esempi: `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`.

- **Tipi reference:** rappresentano riferimenti (puntatori) a oggetti in memoria. Esempi: `String`, array (ad es. `int[]`), classi, interfacce e tipi definiti dall'utente.

> **Note**
>
> Vedi Tipi di dato Java e casting.

- **Teoria:** Un sistema di tipi è l'insieme di regole che associa insiemi di valori e operazioni ammissibili.

Esempio Java (tipi):

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

1.6 Operatore

- **Definizione:** Un **simbolo o parola chiave** che esegue un calcolo o un'azione su uno o più operandi.
- **Ruolo:** Gli operatori combinano valori, variabili ed espressioni per produrre nuovi valori o modificare lo stato del programma.
- **In Java:**

> **Note**
>
> Vedi Operatori Java.

- **Teoria:** Gli operatori definiscono le computazioni ammesse sui tipi; insieme a valori e variabili formano le **espressioni**.

Esempio Java (operatori nel contesto):

```
int a = 5 + 3;          // + aritmetico
boolean ok = a > 3;     // > di confronto
ok = ok && true;        // && logico
a += 2;                 // += assegnazione
int sign = (a >= 0) ? 1 : -1; // ?: ternario
```

1.7 Espressione

- **Definizione:** Una combinazione di valori, letterali, variabili, operatori e funzioni che produce un nuovo valore.
- **In Java:** `x + 3`, `Math.sqrt(25)`, `"Hello" + " world"`.
- **Teoria:** Un albero sintattico (syntax tree) che viene valutato producendo un risultato.

Esempio Java (espressioni):

```
int x = 10;
int y = x + 3;              // x + 3 è un'espressione
double r = Math.sqrt(25);   // Math.sqrt(25) è un'espressione
String msg = "Hello" + " "; // "Hello" + " " è un'espressione
msg = msg + "world";        // msg + "world" è un'altra espressione
```

1.8 Istruzione

- **Definizione:** Unità di esecuzione che modifica lo stato o controlla il flusso.
- **In Java:** `x = x + 1;`, `if (x > 0) { ... }`.

- **Teoria:** Sequenza di azioni che non restituisce un valore come risultato dell'istruzione stessa, ma cambia la configurazione della macchina astratta.

Esempio Java (istruzioni):

```
int x = 0;                // istruzione di dichiarazione
x = x + 1;                // istruzione di assegnazione

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

1.9 Blocco di Codice

- **Definizione:** Insieme di istruzioni racchiuse tra delimitatori che formano un'unità eseguibile.
- **In Java:** `{ int y = 5; x = x + y; }`.
- **Teoria:** Composizione sequenziale di istruzioni, con regole di *scope* (visibilità).

Esempio Java (blocco di codice e scope):

```
int x = 10;

{
    int y = 5;         // y è visibile solo dentro questo blocco
    x = x + y;         // OK: x è visibile qui
}

// y non è visibile qui
// x è ancora visibile qui
```

1.10 Funzione / Metodo

- **Definizione:** Sequenza di istruzioni incapsulata, identificata da un nome, che può ricevere input (parametri) e restituire un output (valore).
- **In Java:**

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

- **Teoria:** Una mappatura tra domini di input e di output, con un corpo operativo.

Esempio di utilizzo in Java:

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

1.11 Classe / Oggetto

- **Definizione:**
  - **Classe:** descrizione astratta di un insieme di oggetti (stato + comportamento).
  - **Oggetto:** istanza concreta della classe.
- **In Java:**

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

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

Point p = new Point();    // p è un oggetto (istanza di Point)
p.move(1, 2);             // chiamata di metodo sull'oggetto
```

- **Teoria:** Astrazione di un *ADT* (Abstract Data Type, tipo di dato astratto).

1.12 Modulo / Package

- **Definizione:** Raggruppamento logico di classi, funzioni e risorse con uno scopo comune.
- **In Java:** `package java.util;` → raccoglie utilità varie.
- **Teoria:** Meccanismo di organizzazione e riuso, riduce la complessità.

Esempio Java (package):

```java
package com.example.app;

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

1.13 Programma

- **Definizione:** Insieme coerente di moduli, classi e funzioni che, quando eseguiti su una macchina, realizzano un comportamento globale.
- **In Java:** Il metodo `main` e tutto ciò che esso invoca.
- **Teoria:** Specifica di trasformazioni da input a output su una macchina astratta.

Esempio Java (programma minimale):

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

1.14 Sistema

- **Definizione:** Insieme di programmi cooperanti che interagiscono con risorse esterne (utente, rete, dispositivi).
- **Esempio:** Una piattaforma Java enterprise con database, servizi REST, interfaccia utente.
- **Teoria:** Architettura complessa di componenti software e hardware.

Esempio (concettuale):

- Un backend Java (servizio Spring Boot)
- Un database (PostgreSQL)
- Una web app front-end
- Servizi esterni (API REST, code di messaggi)

Insieme formano un **sistema**.

1.15 📌 Riepilogo come scala crescente

`Valore → Letterale → Identificatore → Variabile → Tipo → Operatore → Espressione → Istruzione → Blocco di Codice → Funzione/Metodo → Classe/Oggetto → Modulo/Package → Programma → Sistema`

Questa scala mostra come unità concettuali piccole vengano combinate in strutture progressivamente più grandi e complesse.

1.16 📊 Diagramma gerarchico (ASCII)

Descrizione: Questo diagramma ASCII mostra la relazione gerarchica tra i mattoni, dal più complesso (Sistema) al più semplice (Valore e la sua forma concreta, il Letterale).

```
Sistema
└── Programma
    └── Modulo / Package
        └── Classe / Oggetto
            └── Funzione / Metodo
                └── Blocco di Codice
                    └── Istruzione
                        └── Espressione
                            └── Operatore
                                └── Tipo
                                    └── Variabile
                                        └── Identificatore
                                            └── Letterale
                                                └── Valore
```

1.17 📊 Diagramma gerarchico (Mermaid)

Descrizione: Il diagramma Mermaid rende la stessa gerarchia in un albero dall'alto verso il basso. Evidenzia che il Letterale è la forma sintattica di un Valore.

```
graph TD
    A[Sistema]
    A --> B[Programma]
    B --> C[Modulo / Package]
    C --> D[Classe / Oggetto]
    D --> E[Funzione / Metodo]
    E --> F[Blocco di Codice]
    F --> G[Istruzione]
    G --> H[Espressione]
    H --> H2[Operatore]
    H2 --> I[Tipo]
    I --> J[Variabile]
    J --> K[Identificatore]
    K --> L[Letterale]
    L --> M[Valore]
```

◀ Eclipse main shortcuts | ▲ Index | 2. Mattoni di base del linguaggio Java ▶

2. Mattoni di base del linguaggio Java

Indice

- 2.1 Definizione di classe
- 2.2 Commenti
- 2.3 Modificatori di accesso
- 2.4 Package
  - 2.4.1 Organizzazione e scopo
  - 2.4.2 Mappatura con il file system e dichiarazione di un package
  - 2.4.3 Appartenere allo stesso package
  - 2.4.4 Importare da un package
  - 2.4.5 Import statici
    - 2.4.5.1 Regole di precedenza
  - 2.4.6 Package standard vs. package definiti dall'utente
- 2.5 Il metodo main
  - 2.5.1 Firma del metodo main
- 2.6 Compilare ed eseguire il codice
  - 2.6.1 Compilare un file, package di default (singola directory)
  - 2.6.2 Più file, package di default (singola directory)
  - 2.6.3 Codice dentro package (layout standard src → out)
  - 2.6.4 Compilare verso un'altra directory (-d)
  - 2.6.5 Più file su più package (compilare l'intero albero sorgente)
  - 2.6.6 Esecuzione di un singolo sorgente (run veloce senza javac)
  - 2.6.7 Passare parametri a un programma Java

Questo capitolo introduce gli elementi strutturali essenziali di un programma Java: `class`, `method`, `comment`, `access modifier`, `package`, il metodo `main` e i comandi di base da riga di comando (`javac` e `java`).

Questi sono gli elementi minimi indispensabili per scrivere, compilare, organizzare ed eseguire codice Java utilizzando il JDK (Java Development Kit) — senza ricorrere ad alcun IDE (Integrated Development Environment).

2.1 Definizione di classe

Una `class` Java è il mattone fondamentale di un programma Java.

Una `classe` rappresenta un **tipo di dato** definito dall'utente, costituito da un insieme di dati interni (`fields`) e dalle operazioni che possono agire su di essi (`methods`).

Una `class` è un **blueprint** (modello), mentre gli `object` sono **istanze concrete** create a runtime sulla base di questo modello.

Una classe Java è composta da due elementi principali, detti **membri** della classe:

- **Field** (o variabili): rappresentano i dati che definiscono lo stato del nuovo tipo.
- **Method** (o funzioni): rappresentano le operazioni che possono essere eseguite su questi dati.

Alcuni membri possono essere dichiarati con la keyword **static**.

Un membro static appartiene alla classe stessa, non agli oggetti creati da essa.

Ciò significa che:

- esiste una sola copia condivisa tra tutte le istanze
- il membro può essere utilizzato senza dover creare un oggetto della classe
- esso è caricato in memoria quando la classe viene caricata dalla JVM

I membri non statici (detti **di istanza**) appartengono invece ai singoli oggetti e ogni istanza ne possiede la propria copia.

Normalmente, ogni classe è definita nel proprio file "**.java**"; per esempio, una classe chiamata `Person` sarà definita nel corrispondente file `Person.java`.

Qualsiasi classe definita in modo indipendente nel proprio file sorgente è detta **top-level class**.

Una classe di questo tipo può essere dichiarata solo come `public` oppure con il modificatore di accesso di default (`package-private` , cioè senza alcun access modifier esplicito).

Un singolo file, tuttavia, può contenere più di una definizione di classe.
In questo caso, solo una classe può essere dichiarata `public`, e il nome del file deve corrispondere a quella classe.

Le **nested class**, ovvero classi dichiarate all'interno di un'altra classe, possono usare qualunque modificatore di accesso: `public` , `protected` , `private` , `default` (package-private).

- Esempio:

```
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

Nella sua forma più semplice, potremmo teoricamente avere una classe senza metodi e senza field. Sebbene una classe del genere venga compilata, avrebbe ben poco senso pratico.

Token / Identifier	Category	Meaning	Optional?
public	Keyword / access modifier	determina quali altre classi possono usare o vedere quell'elemento	Mandatory (quando assente è, per default, package-private)
class	Keyword	Dichiara un tipo di classe.	Mandatory
Person	Class name (identifier)	Il nome della classe.	Mandatory
personName	Field name (identifier)	Memorizza il nome della persona.	Optional
String	Type / Keyword	Tipo del field `personName` e del parametro `newName`.	Mandatory
setPersonName, getPersonName	Method names (identifier)	denominano un comportamento della classe.	Optional
newName	Parameter name (identifier)	input passato al metodo `setPersonName`.	Mandatory (se il metodo richiede un parametro)
return	Keyword	Termina un metodo e restituisce un valore.	Mandatory (nei metodi con tipo di ritorno non void)
void	Return Type / Keyword	Indica che il metodo non restituisce alcun valore.	Mandatory (se il metodo non restituisce alcun valore)

> **Note**
>
> Mandatory = richiesto dalla sintassi Java, Optional = non richiesto dalla sintassi; dipende dal design.

2.2 Commenti

I commenti non sono codice eseguibile: **spiegano** il codice ma vengono ignorati dal compilatore.

In Java esistono 3 tipi di commenti: - Single-line (`//`) - Multi-line (`/* ... */`) - Javadoc (`/** ... */`)

Un **single-line comment** inizia con 2 slash: tutto il testo che segue sulla stessa riga viene ignorato dal compilatore.

- Esempio:

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

Un **multiline comment** include tutto ciò che si trova tra i simboli `/*` e `*/`.

- Esempio:

```
/*
 * 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 **Javadoc comment** è simile a un **multiline comment**, tranne per il fatto che inizia con `/**`: tutto il testo compreso tra i simboli di apertura e chiusura viene elaborato dallo strumento Javadoc per generare la documentazione delle 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**
>
> In Java, ogni block comment deve essere chiuso correttamente con `*/`.

- Esempio:

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

va bene, ma

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

produrrà un errore di compilazione perché, mentre i primi due simboli fanno parte del commento, l'ultimo no. Il simbolo extra `*/` non è sintassi valida, quindi il compilatore segnalerà il problema.

2.3 Modificatori di accesso

In Java, un **access modifier** è una keyword che specifica la visibilità (o accessibilità) di una **class**, **method** o **field**. Questo modificatore determina quali altre classi possono usare o vedere quel particolare elemento.

> **Note**
>
> **Tabella dei modificatori di accesso disponibili in Java**

Token / Identifier	Category	Meaning	Optional?
public	Keyword / access modifier	Visibile da qualsiasi classe in qualunque package	Sì
no modifier (default)	Keyword / access modifier	Visibile solo all'interno dello stesso package	Sì
protected	Keyword / access modifier	Visibile nello stesso package e dalle sottoclassi (anche in altri package)	Sì
private	Keyword / access modifier	Visibile solo all'interno della stessa classe	Sì

> **Tip**
>
> **private > default > protected > public** La "visibilità si amplia" secondo questo schema.

2.4 Package

I **package Java** sono raggruppamenti logici di classi, interfacce e sotto-package.
Aiutano a organizzare codebase grandi, evitare conflitti di nomi e fornire accesso controllato tra parti diverse di un'applicazione.

2.4.1 Organizzazione e scopo

- I nomi dei package seguono le stesse regole dei nomi di variabile. Vedi [Regole di naming Java](#).
- I package sono come **cartelle** per il codice sorgente Java.
- Permettono di raggruppare classi correlate (ad esempio tutte le utility in `java.util`, tutte le classi di rete in `java.net`).
- Usando i package puoi evitare **conflitti di nomi**; ad esempio, puoi avere due classi chiamate `Date`, ma una è `java.util.Date` e l'altra è `java.sql.Date`.

2.4.2 Mappatura con il file system e dichiarazione di un package

- I package corrispondono direttamente alla **gerarchia di directory** sul file system.
- Il package va dichiarato all'inizio del file sorgente (**prima di qualsiasi import**).
- Se non dichiari un package, la classe appartiene al package di default.
  - Questo non è raccomandato nei progetti reali, perché rende più complicate l'organizzazione e gli import.
- Esempio:

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

public class MyApp{

}
```

> **Important**
>
> Questa dichiarazione ci dice che la classe appartiene al package `com.example.myapp.utils` e che il suo file deve trovarsi nel path fisico: **com/example/myapp/utils/MyApp.java**

2.4.3 Appartenere allo stesso package

Due classi appartengono allo stesso package se e solo se:

- Sono dichiarate con la stessa istruzione `package` all'inizio del rispettivo file sorgente.
- Sono collocate nella stessa directory della gerarchia dei sorgenti.
- Esempio:

Una classe nel package `A.B.C` appartiene solo a `A.B.C`, non a `A.B`.
Le classi in `A.B` non possono accedere direttamente ai membri **package-private** delle classi in `A.B.C`, perché si tratta di package diversi.

Le classi nello stesso package:

- Possono accedere ai membri `package-private` l'una dell'altra (cioè membri senza modificatore di accesso esplicito).
- Condividono lo stesso namespace, quindi non si ha bisogno di importarle per poterle usare.

Esempio: Due file nello stesso 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: stesso package, nessun import necessario
    }
}
```

2.4.4 Importare da un package

Per usare classi da un altro package, si deve importarle:

- Esempio:

```java
import java.util.List;       // importa una specifica classe
import java.util.*;          // importa tutte le classi in java.util

import java.nio.file.*.*     // ERROR! solo una wildcard è permessa e deve comparire alla fine
```

> **Note**
>
> Il carattere wildcard `*` importa tutti i tipi nel package, ma non i sotto-package.

Nel codice comunque, puoi sempre usare il nome completo (fully qualified name) della classe invece di importare tutte le classi del package:

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

> **Note**
>
> Se importi esplicitamente un nome di classe, questo ha precedenza su qualsiasi import con wildcard;
>
> Se vuoi usare due classi con lo stesso nome (ad esempio `Date` da `java.util` e da `java.sql`), è piu prudente usare una import con nome completamente qualificato.

2.4.5 Import statici

Oltre a importare classi da un package, Java permette un altro tipo di import: lo **static import**.

Uno *static import* ti consente di importare i **membri statici** di una classe — come `metodi statici` e `variabili statiche` — in modo da poterli usare **senza dover specificare il nome della classe**.

Puoi importare membri statici **specifici** oppure usare una **wildcard** per importare tutti i membri statici di una particolare classe.

Esempio — Static import specifico

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

public class Example {

    List<String> arr = asList("first", "second");
    // Puoi invocare asList() direttamente, senza usare Arrays.asList()
}
```

Esempio — Static import di una costante

```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);
}
```

Esempio — Static import con wildcard

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

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

Gli static import con wildcard si comportano esattamente come gli import normali con wildcard: portano in scope **tutti i membri statici** della classe.

> **Warning**
>
> Puoi **sempre** chiamare un membro statico usando il nome della classe: `Math.sqrt(16)` funziona sempre — anche se è stato importato staticamente.

2.4.5.1 Regole di precedenza

Se la classe corrente dichiara già un metodo o una variabile con lo stesso nome di quella importata staticamente:

- Il **membro locale ha la precedenza**.
- Il membro statico importato viene **oscurato** (shadowing).

Esempio:

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

public class Test {

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

    void run() {
        System.out.println(max(2, 5));
        // Chiama il max() LOCALE, non Math.max()
    }
}
```

> **Warning**
>
> Uno static import deve sempre seguire l'esatta sintassi: `import static`.
>
> Il compilatore proibisce di importare **due membri statici con lo stesso simple name** se questo crea ambiguità — anche se provengono da classi o package diversi.

Esempio — **Non consentito**:

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

import static java.util.List.of;
import static java.util.Set.of;
// ❌ ERROR: entrambi hanno lo stesso nome di metodo `of()`
```

Il compilatore non sa quale `of()` si intenda usare → errore di compilazione.

> **Tip**
>
> - Se due static import introducono lo stesso nome, **qualsiasi tentativo di usare quel nome provoca un errore di compilazione**.
> - Gli static import **non** importano le classi, solo i membri statici.
> - Puoi sempre chiamare il membro statico usando il nome della classe, anche se lo hai importato staticamente.

Esempio:

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

double a = sqrt(16);        // importato
double b = Math.sqrt(25);   // fully qualified — sempre permesso
```

2.4.6 Package standard vs package definiti dall'utente

- **Package standard**: forniti con il JDK (ad esempio `java.lang`, `java.util`, `java.io`).
- **Package definiti dall'utente**: creati dagli sviluppatori per organizzare il codice dell'applicazione.

2.5 Il metodo `main`

In Java, il metodo `main` funge da **punto di ingresso** di un'applicazione standalone.

La sua dichiarazione corretta è fondamentale perché la JVM possa riconoscerlo.

2.5.1 Firma del metodo main

Analizziamo la firma del metodo `main` all'interno di due delle classi tra le più semplici possibili:

- Esempio: senza modificatori opzionali

```
public class MainFirstExample {

    public static void main(String[] args){

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

    }

}
```

- Esempio: con entrambi i modificatori opzionali `final`

```
public class MainSecondExample {

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

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

    }

}
```

> **Note**
>
> **Tabella dei modificatori per il metodo main**

Token / Identifier	Category	Meaning	Optional?
`public`	Keyword / Access Modifier	Rende il metodo accessibile da qualunque punto. Necessario perché la JVM possa chiamarlo dall'esterno della classe.	Mandatory
`static`	Keyword	Indica che il metodo appartiene alla classe stessa e può essere chiamato senza creare un oggetto. Necessario perché la JVM non ha un'istanza quando avvia il programma.	Mandatory
`final` (before return type)	Modifier	Impedisce che il metodo venga sovrascritto (overridden). Può comparire legalmente prima del tipo di ritorno, ma non ha effetti pratici su `main` e non è richiesto.	Optional
`main`	Method name (predefined)	Il nome esatto che la JVM cerca come punto di ingresso del programma. Deve essere scritto esattamente `main` (minuscolo).	Mandatory
`void`	Return Type / Keyword	Dichiara che il metodo non restituisce alcun valore alla JVM.	Mandatory
`String[] args`	Parameter list	Array di `String` che contiene gli argomenti da riga di comando passati al programma. Può essere scritto anche come `String args[]` o `String... args`. Il nome del parametro (`args`) è arbitrario.	Mandatory (il tipo del parametro è richiesto, il nome può variare)
`final (in parameter)`	Modifier	Indica che il parametro non può essere riassegnato all'interno del metodo (non puoi riassegnare `args` a un altro array).	Optional

Important

I modificatori `public`, `static` (obbligatori) e `final` (se presente) possono essere scambiati d'ordine; `public` e `static` **non possono essere omessi**.

Java considera `String[] args` e `String... args` equivalenti.
Entrambe le firme compilano e funzionano correttamente come punti di ingresso.

2.6 Compilare ed eseguire il codice

Questa sezione mostra l'uso dei comandi `javac` e `java` per i casi più comuni in Java 21: file singoli, più file, package, directory di output separate, uso di classpath/module-path.

Segui i layout delle directory esattamente.

> check your tools

```
javac -version   # output atteso: javac 21.x
java  -version   # output atteso: java version "21.0.7" ... (l'output potrebbe variare a secon
```

> **Warning**
>
> Quando esegui una classe all'interno di un package, **java richiede il nome completamente qualificato**, MAI il path:
>
> `java com.example.app.Main` ✓
>
> `java src/com/example/app/Main` ❌

2.6.1 Compilare un file, package di default (singola directory)

File

```
.
└── Hello.java
```

Hello.java

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

Compilare (nella stessa directory)

```
javac Hello.java
```

Questo comando creerà, nella stessa directory, un file con lo stesso nome del file ".java" ma con estensione ".class"; questo è il file di bytecode che verrà interpretato ed eseguito dalla JVM.

Una volta che hai il file `.class`, in questo caso `Hello.class`, puoi eseguire il programma con:

Esecuzione

```
java Hello
```

> **Important**
>
> Non devi specificare l'estensione ".class" quando esegui il programma.

2.6.2 Più file, package di default (singola directory)

File

```
.
├── A.java
└── B.java
```

Compilare tutto

```
javac *.java
```

Oppure, se le classi appartengono a uno specifico package:

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

Oppure specificando singolarmente:

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

e

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

Eseguire il punto di ingresso del programma: la classe che contiene un metodo `main`

```
java A    # se A ha main(...)
# oppure
java B
```

> **Important**
>
> Il path verso le classi è, in Java, il **classpath**. Puoi specificare il **classpath** con una delle seguenti opzioni:
>
> - `-cp <classpath>`
> - `-classpath <classpath>`
> - `--class-path <classpath>`

2.6.3 Codice dentro package (layout standard src → out)

File

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

> **Note**
>
> Le cartelle `src` e `out` non fanno parte dei nostri package: sono solo le directory che contengono tutti i file sorgenti e i file `.class` compilati.

Main.java

```
package com.example.app;

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

Compilare nella stessa directory

```
# Crea il file .class accanto al file sorgente
javac src/com/example/app/Main.java
```

2.6.4 Compilare verso un'altra directory (`-d`)

L'opzione `-d out` colloca i file `.class` compilati nella directory `out/`, creando sottocartelle che rispecchiano i nomi dei package:

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

Eseguire (usa il classpath puntando a out/)

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

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

2.6.5 Più file su più package (compilare l'intero albero sorgente)

File

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

Compilare l'intero albero sorgente in `out/`

```
# Opzione A: indicare a javac i package di livello più alto
javac -d out   src/com/example/util/Utils.java   src/com/example/app/Main.java

# Opzione B: usare -sourcepath per far trovare a javac le dipendenze
javac -d out -sourcepath src   src/com/example/app/Main.java
```

> **Important**
>
> `-sourcepath <sourcepath>` dice a `javac` dove cercare altri file `.java` da cui i sorgenti dipendono.

2.6.6 Esecuzione di un singolo sorgente (run veloce senza `javac`)

Java 21 (a partire da Java 11) permette di eseguire piccoli programmi direttamente dal sorgente:

```
# Solo package di default
java Hello.java
```

Sono consentiti più file sorgente se si trovano nel **package di default** e nella **stessa directory**:

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

> Se usi i **package**, è preferibile compilare in `out/` ed eseguire con `-cp`.

2.6.7 Passare parametri a un programma Java

Puoi inviare dati al tuo programma Java attraverso i parametri del punto di ingresso `main`.

Come abbiamo visto, il metodo `main` può ricevere un array di stringhe nella forma: **`String[] args`**. Vedi la sezione sul main.

Main.java che stampa due parametri ricevuti in ingresso dal metodo `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]);
    }
}
```

Per passare i parametri, ti basta scrivere (per esempio):

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

Se vuoi passare un argomento contenente spazi, usa le virgolette:

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

> Se dichiari di usare (in questo caso stampare) i primi due elementi dell'array di parametri (come nel nostro esempio), ma in realtà passi meno argomenti, la JVM ti segnalerà il problema con una `java.lang.ArrayIndexOutOfBoundsException`.

> Se, al contrario, passi più argomenti di quelli che il metodo usa, verranno semplicemente utilizzati (in questo caso) solo i primi due.

> `args.length` ti dice quanti argomenti sono stati forniti.

◀ 1. Mattoni Sintattici di Base | ▲ Index | 3. Regole di naming Java ▶

3. Regole di naming Java

Indice

- 3.1 Regole per gli identificatori
  - 3.1.1 Parole riservate
    - 3.1.1.1 Keyword Java riservate
    - 3.1.1.2 Letterali riservati
  - 3.1.2 Sensibilità alle maiuscole/minuscole
  - 3.1.3 Inizio degli identificatori
  - 3.1.4 Numeri negli identificatori
  - 3.1.5 Singolo token `underscore`
  - 3.1.6 Letterali numerici e carattere underscore

Java definisce regole precise per gli **identificatori**, ovvero i nomi assegnati a variabili, metodi, classi, interfacce e package.

Finché si rispettano le regole di naming descritte di seguito, si è liberi di scegliere nomi significativi per gli elementi del programma.

3.1 Regole per gli identificatori

3.1.1 Parole riservate

Gli `identifier` **non possono** coincidere con le **keyword** Java o con i **letterali riservati**.

Le `keyword` sono parole speciali predefinite nel linguaggio Java che non si possono usare come identificatori (vedi tabella qui sotto).

I `literal` come `true`, `false` e `null` sono anch'essi riservati e non possono essere usati come identificatori.

- Esempio:

```
int class = 5;          // non valido: 'class' è una keyword
boolean true = false; // non valido: 'true' è un literal
int year = 2024;        // valido
```

3.1.1.1 Keyword Java riservate

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` e `const` sono riservate ma non utilizzate.

3.1.1.2 Letterali riservati

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

3.1.2 Sensibilità alle maiuscole/minuscole

Gli identificatori in Java sono **case sensitive**.
Questo significa che `myVar` , `MyVar` e `MYVAR` sono tutti identificatori diversi.

- Esempio:

```
int myVar = 1;
int MyVar = 2;
int MYVAR = 3;
int CLASS = 6; // legale ma, please, non farlo!!
```

> **Tip**
>
> Java tratta gli identificatori letteralmente: `Count` , `count` e `COUNt` sono entità distinte e possono coesistere.
>
> A causa della sensibilità a maiuscole/minuscole, si potrebbero usare varianti delle keyword che differiscono solo nel case.
> Anche se è legale, questo stile è fortemente sconsigliato perché riduce la leggibilità ed è considerata una pessima pratica.

3.1.3 Inizio degli identificatori

Gli identificatori in Java devono iniziare con una **lettera**, un **simbolo di valuta** (`$` , `€` , `£` , `₹` ...) oppure il simbolo `_` .

Esempio:

```
int myVarA;
int $myVarB;
int _myVarC;
String €uro = "currency"; // legale (usato raramente in pratica)
```

> **Note**
>
> I simboli di valuta sono legali ma non raccomandati nel codice reale.

3.1.4 Numeri negli identificatori

Gli identificatori in Java possono includere numeri, ma **non possono iniziare** con un numero.

Esempio:

```
int my33VarA;
int $myVar44;
int 3myVarC; // non valido: identifier non possono iniziare con una cifra
int var2024 = 10; // valido
```

3.1.5 Singolo token `underscore`

- Un singolo underscore (`_`) non è consentito come identificatore.
- A partire da Java 9, `_` è un token riservato per un possibile uso futuro del linguaggio.
- Esempio:

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

> **Warning**
>
> `_` è legale all'interno dei letterali numerici (vedi sezione successiva), ma non come identificatore a sé stante.

3.1.6 Letterali numerici e carattere underscore

Si possono usare uno o più caratteri `_` (underscore) nei letterali numerici per renderli più facili da leggere.

Puoi inserire underscore quasi ovunque, **tranne** all'inizio, alla fine o immediatamente prima/dopo il punto decimale.

- Esempio:

```
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**
>
> Gli underscore migliorano la leggibilità: `1_000_000` è più leggibile di `1000000`.

◀ 2. Mattoni di base del linguaggio Java | ▲ Index | 4. Tipi di dato Java e casting ▶

4. Tipi di dato Java e casting

Indice

- 4.1 Tipi primitivi
- 4.2 Tipi reference
- 4.3 Tabella dei tipi primitivi
- 4.4 Note
- 4.5 Riepilogo
- 4.6 Aritmetica e promozione numerica dei primitivi
  - 4.6.1 Regole di promozione numerica in Java
    - 4.6.1.1 Regola 1 – Tipi misti → il tipo più piccolo viene promosso a quello più grande
    - 4.6.1.2 Regola 2 – Integrale + floating-point → l'integrale viene promosso a floating-point
    - 4.6.1.3 Regola 3 – byte, short e char vengono promossi a int durante l'aritmetica
    - 4.6.1.4 Regola 4 – Il tipo del risultato coincide con il tipo promosso
  - 4.6.2 Riepilogo del comportamento di promozione numerica
    - 4.6.2.1 Punti chiave
- 4.7 Casting in Java
  - 4.7.1 Casting dei primitivi
    - 4.7.1.1 Widening implicit casting
    - 4.7.1.2 Narrowing explicit casting
    - 4.7.1.3 Narrowing Implicito a Compile-Time
  - 4.7.2 Perdita di dati, overflow e underflow
  - 4.7.3 Casting di valori vs. variabili
  - 4.7.4 Casting di reference (oggetti)
    - 4.7.4.1 Upcasting (widening reference cast)
    - 4.7.4.2 Downcasting (narrowing reference cast)
  - 4.7.5 Riepilogo dei punti chiave
  - 4.7.6 Esempi
- 4.8 Sommario

Come abbiamo visto in Mattoni Sintattici di Base, Java ha due categorie di tipi di dato:

- **Primitive types**
- **Reference types**

👉 Per una panoramica completa dei tipi primitivi con dimensioni, range, valori di default ed esempi, vedi la Tabella dei tipi primitivi.

4.1 Tipi primitivi

I `primitive` rappresentano **singoli valori grezzi** memorizzati direttamente in memoria.

Ogni tipo primitivo ha una dimensione fissa che determina quanti byte occupa.

Concettualmente, un primitivo è semplicemente una **cella di memoria** che contiene un valore:

```
+-------+
|  42   |   ← value of type short (2 bytes in memory)
+-------+
```

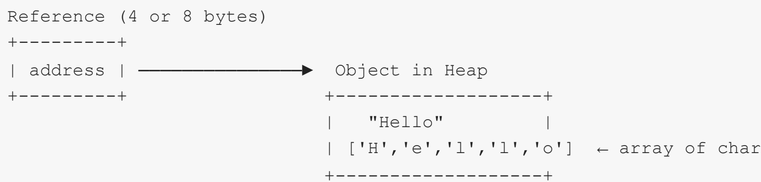
4.2 Tipi reference

Un tipo `reference` contiene l'indirizzo di memoria di un istanza di un tipo complesso; esso non contiene l'`object` stesso, ma un **reference (puntatore)**, appunto, ad esso.

Il reference ha dimensione fissa (dipende dalla JVM, spesso 4 o 8 byte) e punta a una locazione di memoria dove è memorizzato l'oggetto reale.

- Esempio: una variabile reference di tipo `String` punta a un oggetto stringa nello heap, che internamente è composto da un array di primitivi `char`.

Diagramma:



4.3 Tabella dei tipi primitivi

Keyword	Type	Size	Min Value	Max Value	Default Value	Example
`byte`	8-bit int	1 byte	–128	127	0	`byte b = 100;`
`short`	16-bit int	2 bytes	–32,768	32,767	0	`short s = 2000;`
`int`	32-bit int	4 bytes	–2,147,483,648 (`-2^31`)	2,147,483,647 (`2^31-1`)	0	`int i = 123456;`
`long`	64-bit int	8 bytes	–2^63	2^63–1	0L	`long l = 123456789L;`
`float`	32-bit FP	4 bytes	see note	see note	0.0f	`float f = 3.14f;`
`double`	64-bit FP	8 bytes	see note	see note	0.0	`double d = 2.718;`
`char`	UTF-16	2 bytes	`'\u0000'` (0)	`'\uffff'` (65,535)	`'\u0000'`	`char c = 'A';`
`boolean`	true/false	JVM-dep. (often 1 byte)	`false`	`true`	`false`	`boolean b = true;`

4.4 Note

`float` e `double` non hanno limiti interi fissi come i tipi interi.
Invece, seguono lo standard IEEE 754:

- **Più piccoli valori positivi non nulli**:
  - `Float.MIN_VALUE ≈ 1.4E-45`
  - `Double.MIN_VALUE ≈ 4.9E-324`
- **Valori finiti massimi**:
  - `Float.MAX_VALUE ≈ 3.4028235E+38`
  - `Double.MAX_VALUE ≈ 1.7976931348623157E+308`

Supportano anche valori speciali: **`+Infinity`**, **`-Infinity`** e **`NaN`** (Not a Number).

- **FP** = floating point.
- La dimensione di `boolean` dipende dalla JVM ma il comportamento logico è semplicemente `true` / `false`.
- I valori di default si applicano ai **field** (variabili di istanza e di classe).
  Le **variabili locali** devono essere inizializzate esplicitamente prima dell'uso.

4.5 Riepilogo

- **Primitive** = valore reale, memorizzato direttamente in memoria.
- **Reference** = puntatore a un oggetto; l'oggetto stesso può contenere primitivi e altri reference.
- Per i dettagli sui primitivi, vedi la Tabella dei tipi primitivi.

4.6 Aritmetica e promozione numerica dei primitivi

Quando si applicano operatori aritmetici o di confronto ai **tipi primitivi**, Java converte automaticamente (o *promuove*) i valori a tipi compatibili secondo regole ben definite di **numeric promotion**.

Queste regole garantiscono calcoli coerenti e riducono il rischio di perdita di dati quando si mescolano tipi numerici differenti.

4.6.1 Regole di promozione numerica in Java

4.6.1.1 Regola 1 – Tipi misti → il tipo più piccolo viene promosso a quello più grande

Se due operandi appartengono a **tipi numerici diversi**, Java promuove automaticamente il tipo **più piccolo** al tipo **più grande** prima di eseguire l'operazione.

Example	Explanation
`int x = 10; double y = 5.5;` `double result = x + y;`	La variabile `x` di tipo `int` viene promossa a `double`, quindi il risultato è un `double` (`15.5`).

Ordine di promozione valido (dal più piccolo al più grande):
`byte → short → int → long → float → double`

4.6.1.2 Regola 2 – Intero + floating-point → l'intero viene promosso a floating-point

Se un operando è di tipo **intero** (`byte`, `short`, `char`, `int`, `long`) e l'altro è di tipo **floating-point** (`float`, `double`),
il valore intero viene **promosso** al tipo **floating-point** prima dell'operazione.

Example	Explanation
`float f = 2.5F; int n = 3;` `float result = f * n;`	`n` (int) viene promosso a `float`. Il risultato è un `float` (`7.5`).
`double d = 10.0; long l = 3;` `double result = d / l;`	`l` (long) è promosso a `double`. Il risultato è un `double` (`3.333...`).

4.6.1.3 Regola 3 – `byte`, `short` e `char` vengono promossi a `int` durante l'aritmetica

Quando effettui operazioni aritmetiche **con variabili** (non costanti letterali) di tipo `byte`, `short` o `char`,
Java le promuove automaticamente a `int`, anche se **entrambi gli operandi sono più piccoli di `int`**.

Example	Explanation
`byte a = 10, b = 20;` `byte c = a + b;`	❌ Errore di compilazione: il risultato di `a + b` è di tipo `int`, non `byte`. Serve il cast → `byte c = (byte)(a + b);`
`short s1 = 1000, s2 = 2000;` `short sum = (short)(s1 + s2);`	Gli operandi sono promossi a `int`, quindi è richiesto un cast esplicito per assegnare a `short`.
`char c1 = 'A', c2 = 2;` `int result = c1 + c2;`	`'A'` (65) e `2` sono promossi a `int`, risultato = `67`.

> **Note**
>
> Questa regola si applica quando si usano **variabili**.
>
> Quando si usano **letterali costanti**, il compilatore può a volte valutare l'espressione a compile-time e assegnarla in sicurezza.

```
byte a = 10 + 20;   // ✅ OK: espressione costante che rientra in byte
byte b = 10;
byte c = 20;
byte d = b + c;     // ❌ Errore: b + c è valutato a runtime → int
```

4.6.1.4 Regola 4 – Il tipo del risultato coincide con il tipo promosso

Dopo l'applicazione delle promozioni, quando entrambi gli operandi sono dello stesso tipo,
il **risultato** dell'espressione avrà quel **medesimo tipo promosso**.

Example	Explanation
`int i = 5; double d = 6.0;` `var result = i * d;`	`i` viene promosso a `double`, il risultato è `double`.
`float f = 3.5F; long l = 4L;` `var result = f + l;`	`l` viene promosso a `float`, il risultato è `float`.
`int x = 10, y = 4;` `var div = x / y;`	Entrambi sono `int`, il risultato è `int` (`2`), la parte frazionaria viene troncata.

> **Warning**
>
> La divisione tra interi produce sempre un **risultato intero**.
>
> Per ottenere un risultato decimale, **almeno un operando deve essere di tipo floating-point**:

```
double result = 10.0 / 4; // ✅ 2.5
int result = 10 / 4;      // ❌ 2 (la parte frazionaria è scartata)
```

4.6.2 Riepilogo del comportamento di promozione numerica

Situation	Promotion Result	Example
Mix di tipi numerici piccoli e grandi	Il tipo più piccolo è promosso a quello più grande	`int + double → double`
Integrale + floating-point	L'integrale è promosso a floating-point	`long + float → float`
Aritmetica con `byte`, `short`, `char`	Promozione a `int`	`byte + byte → int`
Risultato dopo la promozione	Il risultato ha il tipo promosso	`float + long → float`

4.6.2.1 Punti chiave

- Considera sempre la **promozione di tipo** quando misceli tipi diversi in un'espressione aritmetica.
- Per i tipi piccoli (`byte`, `short`, `char`), la promozione a `int` è automatica quando si usano **variabili** in un'operazione aritmetica.
- Usa il **casting esplicito** solo quando sei sicuro che il risultato rientri nel tipo di destinazione.
- Ricorda: la **divisione tra interi tronca**, la **divisione tra floating-point mantiene i decimali**.
- Comprendere le regole di promozione è cruciale per evitare **perdite di precisione inattese** o **errori di compilazione**.

4.7 Casting in Java

Il `casting` in Java è il processo con cui si converte esplicitamente un valore da un tipo a un altro.
Si applica sia ai `primitive types` (numeri) sia ai `reference types` (oggetti in una gerarchia di classi).

4.7.1 Casting dei primitivi

Il casting dei primitivi cambia il tipo di un valore numerico.

Esistono due categorie di casting:

Type	Description	Example	Explicit?	Risk
Widening	tipo più piccolo → tipo più grande	int → double	No	nessuna perdita
Narrowing	tipo più grande → tipo più piccolo	double → int	Sì	possible loss

4.7.1.1 Widening implicit casting

Conversione automatica da un tipo "più piccolo" a un tipo "più grande" compatibile.
Gestita dal compilatore, **non richiede sintassi esplicita**.

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

✅ **Sicuro** – nessun overflow (anche se bisogna comunque essere consapevoli della precisione dei floating-point).

4.7.1.2 Narrowing Explicit Casting

Conversione manuale da un tipo "più grande" a uno "più piccolo".
Richiede una **cast expression** perché può causare perdita di dati.

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

> **Warning**
>
> ⚠ Usare solo quando si è sicuri che il valore rientri nel tipo di destinazione.

4.7.1.3 Narrowing Implicito a Compile-Time

In alcuni casi specifici, il compilatore permette una conversione di narrowing **senza un cast esplicito**.

Se una variabile è dichiarata `final` ed è inizializzata con una constant expression il cui valore rientra nel tipo di destinazione, il compilatore può eseguire la conversione in modo sicuro a compile time.

```java
final int k = 11;
byte b = k;  // allowed: value 11 fits into byte range

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

Questo funziona perché il compilatore conosce il valore esatto di una variabile `final` e può verificare che sia all'interno dell'intervallo del tipo più piccolo.

Questo tipo di narrowing è consentito tra: - `byte` - `short` - `char` - `int`

Tuttavia, non si applica a: - `long` - `float` - `double`

Per esempio:

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

Anche se il valore sembra compatibile, i tipi floating-point non sono idonei per questa forma di narrowing implicito.

4.7.2 Perdita di dati, overflow e underflow

Quando un valore eccede la capacità di un tipo, puoi ottenere:

- **Overflow**: il risultato è maggiore del massimo valore rappresentabile
- **Underflow**: il risultato è minore del minimo valore rappresentabile
- **Troncamento**: la parte di dato che non entra viene persa (ad esempio, i decimali)
- Esempio – overflow/underflow con `int`

```java
int max = Integer.MAX_VALUE;
int overflow = max + 1;      // "wrap-around" verso il negativo

int min = Integer.MIN_VALUE;
int underflow = min - 1;     // "wrap-around" verso il positivo
```

- Esempio: troncamento

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

> **Note**
>
> I tipi floating-point (`float` , `double`) **non fanno wrap-around**: - overflow → `Infinity` / `-Infinity`
> - underflow (valori molto piccoli) → 0.0 o valori denormalizzati.

4.7.3 Casting di valori vs. variabili

Java tratta:

- I **letterali interi** come `int` di default
- I **letterali floating-point** come `double` di default

Il compilatore **non richiede cast** quando un letterale rientra nel range del tipo di destinazione:

```
byte first = 10;          // OK: 10 rientra in byte
short second = 9 * 10;    // OK: espressione costante valutata a compile time
```

Ma:

```
long a = 5729685479;     // ❌ errore: letterale int fuori range
long b = 5729685479L;    // ✅ letterale long (suffisso L)

float c = 3.14;          // ❌ double → float: richiede F o cast
float d = 3.14F;         // ✅ letterale float

int e = 0x7FFF_FFFF;     // ✅ max int in esadecimale
int f = 0x8000_0000;     // ❌ fuori range int (serve L)
```

Tuttavia, quando si applicano le regole di promozione numerica:

> Con **variabili** di tipo `byte`, `short` e `char` in un'espressione aritmetica, gli operandi vengono promossi a `int` e il risultato è `int`.

```
byte first = 10;
short second = 9 + first;        // ❌ 9 (int literal) + first (byte → int) = int
// second = (short) (9 + first);  // ✅ cast dell'intera espressione
```

```
short b = 10;
short a = 5 + b;                // ❌ 5 (int) + b (short → int) = int
short a2 = (short) (5 + b);     // ✅ cast dell'intera espressione
```

> **Warning**
>
> Il cast è un **operatore unario**:
>
> `short a = (short) 5 + b;`
> Il cast si applica solo a `5` → il risultato dell'espressione resta di tipo `int` → l'assegnazione fallisce comunque.

4.7.4 Casting di reference (oggetti)

Il casting si applica anche ai **reference a oggetti** in una gerarchia di classi.
Non modifica l'oggetto in memoria — cambia solo **il tipo di reference** usato per accedervi.

Regole fondamentali:

- Il **tipo reale dell'oggetto** determina quali field/metodi esistono effettivamente.
- Il **tipo del reference** determina cosa puoi accedere in quel punto del codice.

4.7.4.1 Upcasting (widening reference cast)

Conversione da **sottoclasse** a **superclasse**.

È **implicita** e **sempre sicura**: ogni `Dog` è anche un `Animal`.

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

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

4.7.4.2 Downcasting (narrowing reference cast)

Conversione da **superclasse** a **sottoclasse**.

- È **esplicita**
- Può fallire a runtime con `ClassCastException` se l'oggetto non è realmente di quel tipo

```
Animal a = new Dog();
Dog d = (Dog) a;    // OK: a punta davvero a un Dog

Animal x = new Animal();
Dog d2 = (Dog) x;   // ⚠ Errore a runtime: ClassCastException
```

Per sicurezza, usa `instanceof`:

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

4.7.5 Riepilogo dei punti chiave

Casting Type	Applies To	Direction	Syntax	Safe?	Performed By
Widening Primitive	Primitives	small → large	Implicit	Sì	Compiler
Narrowing Primitive	Primitives	large → small	Explicit	No	Programmer
Upcasting	Objects	subclass → superclass	Implicit	Sì	Compiler
Downcasting	Objects	superclass → subclass	Explicit	Runtime check	Programmer

4.7.6 Esempi

```
// Primitive casting
short s = 50;
int i = s;           // widening
byte b = (byte) i;   // narrowing (possible loss)

// Object casting
Object obj = "Hello";
String str = (String) obj; // OK: obj points to a String

Object n = Integer.valueOf(10);
// String fail = (String) n;  // ClassCastException at runtime
```

4.8 Sommario

- Il **casting dei primitivi** cambia il tipo numerico.
- Il **casting delle reference** cambia la "vista" di un oggetto nella gerarchia.
- **Upcasting** → sicuro e implicito.
- **Downcasting** → esplicito, da usare con cautela (spesso dopo `instanceof`).

---

5. Operatori Java

Indice

- 5.1 Definizione
- 5.2 Tipi di operatori
- 5.3 Categorie di operatori
- 5.4 Precedenza degli operatori e ordine di valutazione
- 5.5 Tabella riassuntiva degli operatori Java
    - 5.5.1 Note aggiuntive
- 5.6 Operatori unari
    - 5.6.1 Categorie di operatori unari
    - 5.6.2 Esempi
- 5.7 Operatori binari
    - 5.7.1 Categorie di operatori binari
    - 5.7.2 Operatori di divisione e resto (modulus)
        - 5.7.2.1 Operatore di divisione
        - 5.7.2.2 Operatore Modulo
    - 5.7.3 Il valore di ritorno dell'operatore di assegnazione
    - 5.7.4 Operatori di assegnazione composta
    - 5.7.5 Operatori di uguaglianza == e !=
        - 5.7.5.1 Uguaglianza con tipi primitivi
        - 5.7.5.2 Uguaglianza con tipi reference (oggetti)
    - 5.7.6 L'operatore instanceof
        - 5.7.6.1 Controllo in fase di compilazione vs fase di esecuzione
        - 5.7.6.2 Pattern matching per instanceof
        - 5.7.6.3 Flow scoping e logica short-circuit
        - 5.7.6.4 Array e tipi reificabili
- 5.8 Operatore Ternario
    - 5.8.1 Regole di Tipo per l'Operatore Ternario
        - 5.8.1.1 Operandi Numerici
        - 5.8.1.2 Tipi di Riferimento
    - 5.8.2 Sintassi
    - 5.8.3 Esempio
    - 5.8.4 Esempio di Ternario Annidato
    - 5.8.5 Note

5.1 Definizione

In Java, gli **operatori** sono simboli speciali che eseguono operazioni su variabili e valori.
Sono i mattoni fondamentali delle espressioni e permettono agli sviluppatori di manipolare i dati, confrontare valori, eseguire operazioni aritmetiche e controllare il flusso logico.

Un'**espressione** è una combinazione di operatori e operandi che produce un risultato.

Per esempio:

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

Qui, `+` e `*` sono operatori, e `a`, `b` e `c` sono operandi.

5.2 Tipi di operatori

Java definisce tre tipi di operatori, raggruppati in base al numero di operandi che utilizzano:

Type	Descrizione	Esempi
Unary	Opera su un singolo operando	`+x`, `-x`, `++x`, `--x`, `!flag`, `~num`
Binary	Opera su due operandi	`a + b`, `a - b`, `x * y`, `x / y`, `x % y`
Ternary	Opera su tre operandi (ce n'è uno solo in Java)	`condition ? valueIfTrue : valueIfFalse`

5.3 Categorie di operatori

Gli operatori possono anche essere raggruppati, in base al loro scopo, in categorie:

Categoria	Descrizione	Esempi
Assignment	Assegna valori alle variabili	`=`, `+=`, `-=`, `*=`, `/=`, `%=`
Relational	Confronta valori	`==`, `!=`, `<`, `>`, `<=`, `>=`
Logical	Combina o inverte espressioni booleane	`|`, `&`, `^`
Conditional	Combina o inverte espressioni booleane	`||`, `&&`
Bitwise	Manipola i bit	`&`, `|`, `^`, `~`, `<<`, `>>`, `>>>`
Instanceof	Verifica il tipo di un oggetto	`obj instanceof ClassName`
Lambda	Usato nelle espressioni lambda	`(x, y) -> x + y`

5.4 Precedenza degli operatori e ordine di valutazione

La **precedenza degli operatori** determina come gli operatori sono raggruppati in un'espressione — cioè, quali operazioni vengono eseguite per prime.

L'**associatività** (o **ordine di valutazione**) determina se l'espressione viene valutata da **sinistra a destra** o da **destra a sinistra** quando gli operatori hanno la stessa precedenza.

Esempio:

```java
int result = 10 + 5 * 2;  // La moltiplicazione avviene prima dell'addizione → result = 20
```

Le parentesi tonde `()` possono essere usate per **forzare la precedenza**:

```java
int result = (10 + 5) * 2;  // Le parentesi vengono valutate per prime → result = 30
```

Note

- La **precedenza** degli operatori riguarda il *raggruppamento*, non l'ordine effettivo di valutazione nel bytecode.
- Usa sempre le parentesi per rendere esplicita la precedenza e migliorare la leggibilità nelle espressioni complesse.

5.5 Tabella riassuntiva degli operatori Java

Precedenza (Alta → Bassa)	Tipo	Operatori	Esempio	Ordine di valutazione	Applicabile a
1	Postfix Unary	`expr++`, `expr--`	`x++`	Sinistra → Destra	Tipi numerici
2	Prefix Unary	`++expr`, `--expr`	`--x`	Sinistra → Destra	Numerici
3	Other Unary	`(type)`, `+`, `-`, `~`, `!`	`-x`, `!flag`	Destra → Sinistra	Numerici, boolean
4	Cast Unary	`(Type) reference`	`(short) 22`	Destra → Sinistra	reference, primitivi
5	Multiplication/division/modulus	`*`, `/`, `%`	`a * b`	Sinistra → Destra	Tipi numerici
6	Additive	`+`, `-`	`a + b`	Sinistra → Destra	Numerici, String (concatenazione)
7	Shift	`<<`, `>>`, `>>>`	`a << 2`	Sinistra → Destra	Tipi interi
8	Relational	`<`, `>`, `<=`, `>=`, `instanceof`	`a < b`, `obj instanceof Person`	Sinistra → Destra	Numerici, reference
9	Equality	`==`, `!=`	`a == b`	Sinistra → Destra	Tutti i tipi (eccetto boolean per `<`, `>`)
10	Logical AND	`&`	`a & b`	Sinistra → Destra	boolean
11	Logical exclusive OR	`^`	`a ^ b`	Sinistra → Destra	boolean
12	Logical inclusive OR	`|`	`a | b`	Sinistra → Destra	boolean
13	Conditional AND	`&&`	`a && b`	Sinistra → Destra	boolean
14	Conditional OR	`||`	`a || b`	Sinistra → Destra	boolean
15	Ternary (Conditional)	`? :`	`a > b ? x : y`	Destra → Sinistra	Tutti i tipi
16	Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	`x += 5`	Destra → Sinistra	Tutti i tipi assegnabili
17	Arrow operator	`->`	`(x, y) -> x + y`	Destra → Sinistra	Espressioni lambda, switch rules

5.5.1 Note aggiuntive

- La **concatenazione di stringhe (`+`)** ha una precedenza più bassa rispetto all'`+` aritmetico sui numeri.
- Usa le parentesi `()` per la precedenza e la leggibilità — non cambiano la semantica ma rendono l'intento più chiaro.

5.6 Operatori unari

Gli operatori unari operano su **un solo operando** per produrre un nuovo valore.
Sono usati per operazioni come incremento/decremento, negazione di un valore, inversione di un booleano o complemento bit a bit.

5.6.1 Categorie di operatori unari

Operatore	Nome	Descrizione	Esempio	Risultato
`+`	Unary plus	Indica un valore positivo (di solito ridondante).	`+x`	Uguale a `x`
`-`	Unary minus	Indica che un numero letterale è negativo o nega un'espressione.	`-5`	`-5`
`++`	Increment	Incrementa una variabile di 1. Può essere prefisso o postfisso.	`++x`, `x++`	`x+1`
`--`	Decrement	Decrementa una variabile di 1. Può essere prefisso o postfisso.	`--x`, `x--`	`x-1`
`!`	Logical complement	Inverte un valore booleano.	`!true`	`false`
`~`	Bitwise complement	Inverte ogni bit di un intero. Quick rule: ~n = -(n + 1)	`~5`	`-6`
`(type)`	Cast	Converte il valore in un altro tipo.	`(int) 3.9`	`3`

5.6.2 Esempi

```
int x = 5;
System.out.println(++x);  // 6  (prefisso: incrementa x a 6, poi restituisce 6)
System.out.println(x++);  // 6  (postfisso: restituisce 6, poi incrementa x a 7)
System.out.println(x);    // 7

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

int a = 5;                  // binario: 0000 0101
System.out.println(~a);     // -6 → binario: 1111 1010 (complemento a due)
```

> **Note**
>
> - Prefisso (`++x` / `--x`): aggiorna prima il valore, poi restituisce il nuovo valore.
> - Postfisso (`x++` / `x--`): restituisce prima il valore corrente, poi lo aggiorna.
> - L'operatore `!` si applica a valori boolean; `~` si applica a tipi numerici interi.

5.7 Operatori binari

Gli operatori binari richiedono **due operandi**.
Eseguono operazioni aritmetiche, relazionali, logiche, bit a bit e di assegnazione.

5.7.1 Categorie di operatori binari

Categoria	Operatori	Esempio	Descrizione
Arithmetic	`+`, `-`, `*`, `/`, `%`	`a + b`	Operazioni matematiche di base.
Relational	`<`, `>`, `<=`, `>=`, `==`, `!=`	`a < b`	Confrontano valori.
Logical (boolean)	`&`, `|`, `^`	`a & b`	Vedi nota sotto.
Conditional	`&&`, `||`	`a && b`	Vedi nota sotto.
Bitwise (integral)	`&`, `|`, `^`, `<<`, `>>`, `>>>`	`a << 2`	Operano sui bit.
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	`x += 3`	Modificano e assegnano.
String Concatenation	`+`	`"Hello " + name`	Uniscono stringhe.

Important

- Gli operatori **logici** (`&`, `|`, `^`) *valutano sempre entrambi i lati*.
- Gli operatori **condizionali** (`&&`, `||`) sono **short-circuit**:
  - `a && b` → `b` è valutato solo se `a` è true
  - `a || b` → `b` è valutato solo se `a` è false

Important

Cheat Sheet Pattern Bitwise e Booleani

```
a ^ a = 0

a ^ 0 = a

a ^ -1 = ~a

a ^ ~a = -1

a & a = a

a | 0 = a
```

Esempio aritmetico:

```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
```

Esempio relazionale:

```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
```

Esempio logico:

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

Esempio bit a bit:

```java
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 Operatori di divisione e resto (modulus)

5.7.2.1 Operatore di Divisione

Dividere un `intero` per zero (ad esempio, 10 / 0) provoca il lancio da parte della JVM di una `java.lang.ArithmeticException: / by zero`.

Tuttavia, la divisione in virgola mobile si comporta in modo diverso.

Quando un valore `float` o `double` viene diviso per 0 o 0.0, non viene lanciata alcuna eccezione. Invece, il risultato è:

- **Float.POSITIVE_INFINITY** oppure **Float.NEGATIVE_INFINITY**
- **Double.POSITIVE_INFINITY** oppure **Double.NEGATIVE_INFINITY**

Il segno dipende dagli operandi coinvolti nell'operazione.

Per determinare se un valore in virgola mobile rappresenta l'infinito, le classi `Float` e `Double` forniscono metodi di utilità:

Metodi statici:

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

Metodi di istanza:

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

Questi metodi restituiscono `true` se il valore corrisponde a infinito positivo o infinito negativo.

5.7.2.2 Operatore Modulo

L'operatore di resto (*modulus*) restituisce il resto della divisione tra due numeri.
Se due numeri si dividono esattamente, il resto è 0: per esempio **10 % 5** è 0.
Al contrario, **13 % 4** restituisce il resto 1.

Possiamo usare il resto anche con numeri negativi secondo le regole seguenti:

- se il **divisore** è negativo (es.: 7 % -5), il segno viene ignorato e il risultato è **2**;
- se il **dividendo** è negativo (es.: -7 % 5), il segno viene preservato e il risultato è **-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 Il valore di ritorno dell'operatore di assegnazione

In Java, l'**operatore di assegnazione (=)** non si limita a memorizzare un valore in una variabile — restituisce anche **il valore assegnato** come risultato dell'intera espressione.

Questo significa che l'operazione di assegnazione può essere **usata come parte di un'altra espressione**,
ad esempio all'interno di un'istruzione `if`, nella condizione di un ciclo o perfino in un'altra assegnazione.

```
int x;
int y = (x = 10);   // l'assegnazione (x = 10) restituisce 10
System.out.println(y);  // 10

// x = 10 assegna 10 a x.
// L'espressione (x = 10) viene valutata a 10.
// Questo valore viene poi assegnato a y.
// Quindi sia x che y finiscono con lo stesso valore (10).
```

Poiché l'assegnazione restituisce un valore, può comparire anche all'interno di un'istruzione **if**.
Tuttavia, ciò porta spesso a errori logici se usata involontariamente.

```
boolean flag = false;

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

// Qui la condizione (flag = true) assegna true a flag, poi viene valutata a true,
// quindi il blocco if viene sempre eseguito.

// Uso corretto (confronto invece di assegnazione):

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

> **Warning**
>
> Se vedi `if (x = qualcosa)` , fermati: è una **assegnazione**, non un confronto.

5.7.4 Operatori di assegnazione composta

Gli **operatori di assegnazione composta** in Java combinano un'operazione aritmetica o bit a bit con l'assegnazione in un unico passaggio.
Invece di scrivere `x = x + 5`, puoi usare la forma abbreviata `x += 5`.
Essi eseguono automaticamente un **cast di tipo** verso il tipo della variabile a sinistra quando necessario.

Gli operatori composti più comuni includono:
`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, e `>>>=`.

```java
int x = 10;

// Assegnazioni composte aritmetiche
x += 5;   // equivale a x = x + 5 → x = 15
x -= 3;   // equivale a x = x - 3 → x = 12
x *= 2;   // equivale a x = x * 2 → x = 24
x /= 4;   // equivale a x = x / 4 → x = 6
x %= 5;   // equivale a x = x % 5 → x = 1

// Assegnazioni composte bit a bit
int y = 6;    // 0110 (binario)
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

// Assegnazioni composte con shift
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

// Esempio di cast di tipo
byte b = 10;
// b = b + 1;    // ❌ errore di compilazione: il risultato int non può essere assegnato a byte
b += 1;          // ✅ funziona: cast implicito di nuovo verso byte
```

> **Note**
>
> Gli operatori di assegnazione composta **eseguono un cast implicito** verso il tipo della variabile a sinistra. Per questo motivo `b += 1` compila anche se `b = b + 1` non compila.

5.7.5 Operatori di uguaglianza (`==` e `!=`)

Gli **operatori di uguaglianza** in Java `==` (uguale a) e `!=` (diverso da) vengono usati per confrontare due operandi.
Tuttavia, il loro comportamento differisce **a seconda che siano applicati a tipi primitivi o a tipi reference (oggetti)**.

> **Note**
>
> - `==` confronta i **valori** per i tipi primitivi
> - `==` confronta i **riferimenti** per gli oggetti
> - `.equals()` confronta il **contenuto dell'oggetto** (se implementato)

5.7.5.1 Uguaglianza con tipi primitivi

Quando si confrontano **valori primitivi**, `==` e `!=` confrontano i **valori effettivamente memorizzati**.

```java
int a = 5, b = 5;
System.out.println(a == b);  // true  → hanno lo stesso valore
System.out.println(a != b);  // false → i valori sono uguali
```

> **Important**
>
> - Se gli operandi sono di tipi numerici diversi, Java li promuove automaticamente a un tipo comune prima del confronto.
> - Tuttavia, confrontare float e double può produrre risultati inattesi a causa di errori di precisione (vedi esempio sotto).

```
int x = 10;
double y = 10.0;
System.out.println(x == y);  // true → x è promosso a double (10.0)

double d = 0.1 + 0.2;
System.out.println(d == 0.3); // false → problema di arrotondamento dei floating point
```

5.7.5.2 Uguaglianza con tipi reference (oggetti)

Per gli oggetti, `==` e `!=` confrontano i riferimenti, non il contenuto dell'oggetto.
Restituiscono true solo se entrambi i riferimenti puntano **allo stesso oggetto** in memoria.

```
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1 == s2);      // false → oggetti diversi in memoria
System.out.println(s1 != s2);      // true  → non lo stesso riferimento
```

Anche se due oggetti hanno contenuto identico, `==` confronta i loro **indirizzi**, non i valori.
Per confrontare il **contenuto** degli oggetti, usa il metodo **`.equals()`**.

```
System.out.println(s1.equals(s2)); // true → stesso contenuto della stringa
```

Caso speciale: null e letterali String

- Qualsiasi reference può essere confrontato con `null` usando `==` o `!=`.

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

- I letterali String sono *internati* dalla Java Virtual Machine (JVM):
  ciò significa che letterali identici possono puntare allo stesso riferimento in memoria:

```
String a = "Java";
String b = "Java";
System.out.println(a == b);       // true → stesso letterale internato
```

- Uguaglianza con tipi misti:
  quando si usa `==` tra operandi di categorie diverse (es. primitivo vs oggetto),
  il compilatore prova a eseguire l'unboxing se uno dei due è una **wrapper class**.

```
Integer i = 100;
int j = 100;
System.out.println(i == j);   // true → unboxing prima del confronto
```

5.7.6 L'operatore `instanceof`

`instanceof` è un **operatore relazionale** che verifica se un valore reference è un'**istanza di** un certo **tipo reference** a runtime.
Restituisce un `boolean`.

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

Comportamento con **null**:
se l'espressione è null, **expr instanceof Type** è sempre **false**.

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

> **Warning**
>
> `instanceof` restituisce sempre `false` quando l'operando a sinistra è `null`.

5.7.6.1 Controllo in fase di compilazione vs fase di esecuzione

- In fase di compilazione, il compilatore rifiuta tipi *inconvertibili* (che non possono essere in relazione a runtime).
- A runtime, se il controllo in compilazione è passato, la JVM valuta il tipo reale dell'oggetto.

```
// ❌ Errore di compilazione: tipi inconvertibili (String non è correlato a Integer)
boolean bad = ("abc" instanceof Integer);

// ✅ Compila, ma il risultato a runtime dipende dall'oggetto reale:

Number num = Integer.valueOf(10);
System.out.println(num instanceof Integer); // true a runtime
System.out.println(num instanceof Double);  // false a runtime
```

5.7.6.2 Pattern matching per instanceof

Java supporta i *type pattern* con `instanceof`, che eseguono sia il test sia il binding della variabile quando il test ha successo.
Aggiungere una variabile dopo il tipo indica al compilatore di trattare il costrutto come *Pattern Matching*.

Sintassi (forma *pattern*):

```
Object obj = "Hello";

if (obj instanceof String str) {
    // Aggiungere la variabile str dopo il tipo istruisce il compilatore a usare il Pattern Ma

    System.out.println(str.toUpperCase()); // l'identificatore è in scope qui, di tipo String
}
```

Proprietà fondamentali:

- Se il test ha successo, la variabile di pattern (es. `str`) è sicuramente assegnata ed è in scope nel ramo `true`.
- Le variabili di pattern sono implicitamente `final` (non possono essere riassegnate).
- Il nome non deve entrare in conflitto con una variabile esistente nello stesso scope.

5.7.6.3 Flow scoping e logica short-circuit

Le variabili di pattern diventano disponibili in base all'analisi di flusso:

```
Object obj = "data";

// Test negato, variabile disponibile nel ramo else
if (!(obj instanceof String s)) {
    // s non è in scope qui
} else {
    System.out.println(s.length()); // s è in scope qui
}

// Con &&, la variabile di pattern può essere usata a destra se a sinistra è stata stabilita
if (obj instanceof String s && s.length() > 3) {
    System.out.println(s.substring(0, 3)); // s in scope
}

// Con ||, la variabile di pattern NON è sicura a destra (lo short-circuit può impedire che ve
if (obj instanceof String s || s.length() > 3) {  // ❌ s non è in scope qui
    // ...
}

// Le parentesi possono aiutare a raggruppare la logica
if ((obj instanceof String s) && s.contains("a")) { // ✅ s in scope dopo il test raggruppato
    System.out.println(s);
}
```

Il pattern matching con `null` viene valutato, come sempre per `instanceof`, a `false`:

```java
String str = null;

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

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

Tipi supportati:

Il tipo della variabile di pattern deve essere un sottotipo, un supertipo o lo stesso tipo della variabile reference.

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

if (num instanceof String s) {}   // ❌ Errore di compilazione
if (num instanceof Short s) {}    // ✅ Ok
if (num instanceof Object s) {}   // ✅ Ok
if (num instanceof Number s) {}   // ✅ Ok
```

5.7.6.4 Array e tipi reificabili

`instanceof` funziona con gli array (che sono reificabili) e con forme generiche *erased* o con wildcard.
I **tipi reificabili** sono quelli la cui rappresentazione a runtime preserva completamente il tipo (per esempio: raw types, array, classi non generiche, wildcard `?`).
A causa del *type erasure*, `List<String>` non può essere testata direttamente a runtime.

```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>); // ❌ Errore di compilazione: tipo parar
System.out.println(list instanceof java.util.List<?>); // ✅ true
```

5.8 Operatore ternario

L'**operatore ternario** (`? :`) è l'unico operatore in Java che accetta **tre operandi**.
Rappresenta una forma compatta dell'istruzione `if-else`.

5.8.1 Regole di Tipo per l'Operatore Ternario

Il tipo di un'espressione condizionale (ternaria) è determinato dai tipi del secondo e del terzo operando.

5.8.1.1 Operandi Numerici

- Se un operando è di tipo `byte` e l'altro è di tipo `short`, il tipo risultante è `short`.
- Se un operando è di tipo `T` (`byte`, `short` o `char`) e l'altro è un'espressione costante di tipo `int` il cui valore è rappresentabile in `T`, allora il tipo risultante è `T`.
- In tutti gli altri casi numerici si applica la **binary numeric promotion** ai due operandi.
  Il tipo dell'espressione condizionale diventa il tipo promosso.

> La binary numeric promotion include **unboxing conversion** e **value set conversion**.

5.8.1.2 Tipi di Riferimento

- Se un operando è `null` e l'altro è un tipo di riferimento, il tipo risultante è quel tipo di riferimento.
- Se i due operandi sono tipi di riferimento diversi, uno deve essere assegnabile all'altro (compatibilità per assegnazione).
  Il tipo risultante è il tipo più generale, cioè quello a cui l'altro può essere assegnato.
- Se nessuno dei due tipi è compatibile per assegnazione con l'altro, si verifica un **errore a compile-time**.

In sintesi, l'operatore ternario determina il proprio tipo applicando:

- Regole speciali di narrowing per piccoli tipi integrali
- Binary numeric promotion per i valori numerici
- Regole di compatibilità per assegnazione per i tipi di riferimento

> **Tip**
>
> L'operatore ternario **deve** produrre un valore di tipo compatibile. Se i due rami restituiscono tipi non correlati, la compilazione fallisce.
>
> ```
> String s = true ? "ok" : 5; // ❌ errore di compilazione: tipi incompatibili
> ```

5.8.2 Sintassi

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

5.8.3 Esempio

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

5.8.4 Esempio di Ternario Annidato

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

5.8.5 Note

> **Warning**
>
> - Le espressioni ternarie annidate possono ridurre la leggibilità. Usare le parentesi per maggiore chiarezza.
> - L'operatore ternario restituisce un **valore**, a differenza di `if-else`, che è un'istruzione.

◀ 4. Tipi di dato Java e casting | ▲ Index | 6. Istanziazione dei tipi ▶

6. Istanziazione dei tipi

Indice

- 6.1 Introduzione
  - 6.1.1 Gestione dei tipi primitivi
    - 6.1.1.1 Dichiarare un primitivo
    - 6.1.1.2 Assegnare un primitivo
  - 6.1.2 Gestione dei tipi reference
    - 6.1.2.1 Creare e assegnare un reference
    - 6.1.2.2 Costruttori
    - 6.1.2.3 Blocchi di inizializzazione-istanza
- 6.2 Inizializzazione predefinita delle variabili
  - 6.2.1 Variabili di istanza e di classe
  - 6.2.2 Variabili final di istanza
  - 6.2.3 Variabili locali
    - 6.2.3.1 Inferire i tipi con var
- 6.3 Tipi wrapper
  - 6.3.1 Scopo dei tipi wrapper
  - 6.3.2 Autoboxing e unboxing
  - 6.3.3 Parsing e conversione
  - 6.3.4 Metodi di supporto
  - 6.3.5 Valori null
- 6.4 Uguaglianza in Java
  - 6.4.1 Uguaglianza con i tipi primitivi
    - 6.4.1.1 Punti chiave
  - 6.4.2 Uguaglianza con i tipi reference
    - 6.4.2.1 Confronto di identità
    - 6.4.2.2 equals Confronto logico
    - 6.4.2.3 Punti chiave
  - 6.4.3 String Pool e uguaglianza
    - 6.4.3.1 Il metodo intern
  - 6.4.4 Uguaglianza con i tipi wrapper
    - 6.4.4.1 Caching dei Wrapper
    - 6.4.4.2 La keyword `new` aggira la cache
    - 6.4.4.3 Confronto dei wrapper
    - 6.4.4.4 Tipi Wrapper diversi non possono essere confrontati
  - 6.4.5 Uguaglianza e null
  - 6.4.6 Tabella riepilogativa

6.1 Introduzione

In Java, un **tipo** può essere un **tipo primitivo** (come `int`, `double`, `boolean`, ecc.) oppure un **tipo reference** (classi, interfacce, array, enum, record, ecc.). Vedi: Tipi di dato Java e casting

Il modo in cui vengono create le istanze dipende dalla categoria del tipo:

- **Tipi primitivi**
  Le istanze dei tipi primitivi vengono create semplicemente dichiarando una variabile.
  La JVM alloca automaticamente la memoria per contenere il valore e non è necessaria alcuna keyword esplicita.

```
int age = 30;          // crea un primitivo int con valore 30
boolean flag = true;   // crea un primitivo boolean con valore true
double pi = 3.14159;   // crea un primitivo double con valore 3.14159
```

- **Tipi reference (oggetti)**
  Le istanze dei tipi classe vengono create usando la keyword `new` (eccetto alcuni casi speciali come i letterali String, i record con costruttori canonici, o i metodi factory). La keyword `new` alloca memoria nell'heap e invoca un costruttore della classe.

```
String name = new String("Alice"); // crea esplicitamente un nuovo oggetto String
Person p = new Person();           // crea un nuovo oggetto Person usando il suo costruttore
```

È anche comune affidarsi a letterali o metodi factory per la creazione degli oggetti.

```
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**
>
> I letterali String **non richiedono** **`new`** e sono memorizzati nello **String pool**. Usare `new String("x")` crea invece sempre un nuovo oggetto nell'heap.

6.1.1 Gestione dei tipi primitivi

6.1.1.1 Dichiarare un primitivo

Dichiarare un tipo primitivo (come per i tipi reference) significa riservare spazio in memoria per una variabile di un determinato tipo, senza necessariamente assegnarle un valore.

> **Warning**
>
> A differenza dei primitivi, la cui dimensione dipende dal tipo specifico (es. `int` vs `long`), i tipi reference occupano sempre la stessa dimensione fissa in memoria — ciò che varia è la dimensione dell'oggetto a cui puntano.

- Esempi di sintassi (solo dichiarazione):

```
int number;

boolean active;

char letter;

int x, y, z;          // Dichiarazioni multiple in un'unica istruzione: Java consente di dichi
```

> **Important**
>
> I `modificatori` e il `tipo` dichiarati all'inizio di una dichiarazione multipla di variabili si applicano a tutte le variabili dichiarate in quella stessa istruzione.
>
> **Eccezione**: quando si dichiarano array usando le parentesi quadre dopo il nome della variabile, le parentesi fanno parte del dichiaratore della singola variabile, non del tipo base.

- Esempi

```
static int a, b, c;

// è equivalente a:

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


int[] a, b;    // entrambi sono arrays di int
int c[], d;    // solo c è un array, d è un normale int
```

6.1.1.2 Assegnare un primitivo

Assegnare un tipo primitivo (come per i tipi reference) significa memorizzare un valore in una variabile dichiarata di quel tipo.

Per i primitivi, la variabile contiene il valore stesso; per i tipi reference, la variabile contiene l'indirizzo di memoria (un reference) dell'oggetto puntato.

- Esempi di sintassi:

```
int number;                      // Dichiarazione di un int: variabile chiamata "number"

number = 10;                     // Assegnazione del valore 10 a questa variabile

char letter = 'A';               // Dichiarazione e assegnazione in un'unica istruzione: dichia

int a1, a2;                       // Dichiarazioni multiple

int a = 1, b = 2, c = 3;         // Dichiarazioni multiple e assegnazioni

char b1, b2, b3 = 'C';           // Dichiarazioni miste (2 dichiarazioni + 1 assegnazione)

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

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

> **Important**
>
> Quando scrivi un numero direttamente nel codice (un letterale numerico), Java assume per default che sia di tipo **int**. Se il valore non entra in un `int`, il codice non compila a meno che il letterale non sia marcato esplicitamente con il suffisso corretto.

- Esempio di sintassi per un letterale numerico:

```
long exNumLit = 5729685479; // ❌ Does not compile.
                            // Anche se il valore potrebbe rientrare in un long,
                            // un literal numerico semplice è considerato un int,
                            // e questo numero è troppo grande per un int.

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

long exNumLit = 5729685479L;

or

long exNumLit = 5729685479l;
```

Dichiarare un tipo `reference` significa riservare spazio in memoria per una variabile che conterrà un reference (puntatore) a un oggetto del tipo specificato.

A questo stadio non viene ancora creato alcun oggetto — la variabile ha solo la potenzialità di puntarne uno.

> **Warning**
>
> A differenza dei primitivi, la cui dimensione dipende dal tipo specifico (es. `int` vs `long`), le variabili reference occupano sempre la stessa dimensione fissa in memoria (sufficiente per memorizzare un reference). Ciò che varia è la dimensione dell'oggetto puntato, che viene allocato separatamente nell'heap.

- Esempi di sintassi (solo dichiarazione):

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

Person p1, p2, p3;   // Dichiarazioni multiple in un'unica istruzione

String a = "abc", b = "def", c = "ghi";      // Dichiarazioni multiple e assegnazioni

String b1, b2, b3 = "abc"                    // Dichiarazioni miste (b1, b2) con una assegnazic

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

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

6.1.2 Gestione dei tipi reference

6.1.2.1 Creare e assegnare un reference

Assegnare un tipo `reference` significa memorizzare nella variabile l'indirizzo di memoria di un oggetto.

Questo si fa normalmente dopo la creazione dell'oggetto con la keyword **new** e un costruttore, oppure usando un letterale o un metodo factory.

Un reference può anche essere assegnato a un altro oggetto dello stesso tipo o di tipo compatibile.

I tipi reference possono anche essere assegnati a **null**, il che significa che non faranno riferimento ad alcun oggetto.

- Esempi di sintassi:

```
Person person = new Person(); // Esempio con 'new' e un costruttore 'Person()':
                              // 'new Person()' crea un nuovo oggetto Person nell'heap
                              // e restituisce il suo reference, che viene memorizzato nella var

String greeting = "Hello";   // Esempio con letterale (per String).

List<Integer> numbers = List.of(1, 2, 3);   // Esempio con un metodo factory.
```

6.1.2.2 Costruttori

Nell'esempio, `Person()` è un costruttore — un tipo speciale di metodo usato per inizializzare nuovi oggetti.

Ogni volta che chiami `new Person()`, il costruttore viene eseguito e imposta la nuova istanza creata.

I costruttori hanno tre caratteristiche principali:

- Il nome del costruttore **deve corrispondere esattamente al nome della classe** (case-sensitive).
- I costruttori **non dichiarano un tipo di ritorno** (nemmeno `void`).
- Se non definisci alcun costruttore nella tua classe, il compilatore fornisce automaticamente un **costruttore di default senza argomenti** che non fa nulla.

> **Warning**
>
> Se vedi un metodo che ha lo stesso nome della classe **ma dichiara anche un tipo di ritorno**, **non** è un costruttore. È semplicemente un metodo normale (anche se iniziare i nomi dei metodi con una lettera maiuscola va contro le convenzioni di naming in Java).

Lo **scopo di un costruttore** è inizializzare lo stato di un oggetto appena creato — tipicamente assegnando valori ai suoi campi, con valori di default oppure usando parametri passati al costruttore.

- Esempio 1: Costruttore di default (senza parametri)

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

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

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

- Esempio 2: Costruttore con parametri

```java
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
```

- Esempio 3: Costruttori multipli (overloading dei costruttori)

```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

- **I costruttori non sono ereditati**: se una superclasse definisce costruttori, non sono automaticamente disponibili nella sottoclasse — devi dichiararli esplicitamente.
- Se dichiari un qualsiasi costruttore in una classe, il compilatore non genera il costruttore di default senza argomenti: se ti serve ancora un costruttore senza argomenti, devi dichiararlo manualmente.

6.1.2.3 Blocchi di inizializzazione istanza

Oltre ai costruttori, Java offre un meccanismo chiamato **initializer blocks** per l'inizializzazione degli oggetti.

Sono blocchi di codice all'interno di una classe, racchiusi tra `{ }`, che vengono eseguiti **ogni volta che viene creata un'istanza**, subito prima dell'esecuzione del corpo del costruttore.

Caratteristiche

- Chiamati anche **instance initializer blocks**.
- Eseguiti, insieme all'inizializzazione dei campi, nell'ordine in cui appaiono nella definizione della classe ma sempre prima dei costruttori.
- Utili quando più costruttori devono condividere un codice comune di inizializzazione.

Esempio: usare 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

In questo esempio, il blocco di inizializzazione viene eseguito prima del corpo di entrambi i costruttori. Sia p1 che p2 partiranno con age = 18, indipendentemente da quale costruttore viene usato.

Blocchi di inizializzazione multipli: se una classe contiene più initializer blocks, essi vengono eseguiti nell'ordine in cui compaiono nel file sorgente:

- Esempio:

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

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

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

Note

I blocchi di inizializzazione d'istanza sono meno comuni nella pratica, perché una logica simile può spesso essere messa direttamente nei costruttori. È importante sapere che: - Vengono sempre eseguiti prima del corpo del costruttore. - Sono eseguiti nell'ordine di dichiarazione nella classe. - Possono essere combinati con i costruttori per evitare duplicazioni di codice.

Warning

Ordine di inizializzazione quando si crea un oggetto 1. Campi statici 2. Blocchi di inizializzazione statici 3. Campi di istanza 4. Blocchi di inizializzazione d'istanza 5. Corpo del costruttore

---

6.2 Inizializzazione predefinita delle variabili

6.2.1 Variabili di istanza e di classe

- Una **variabile di istanza (un field)** è un valore definito all'interno di un'istanza di un oggetto;
- Una **variabile di classe** (definita con la keyword **static**) è definita a livello di classe ed è condivisa tra tutti gli oggetti (istanze della classe)

Se non inizializzate, variabili di istanza e di classe ricevono un valore di default dal compilatore.

- Tabella dei valori di default per variabili di istanza e di classe:

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

6.2.2 Variabili final di istanza

A differenza delle normali variabili di istanza e di classe, le variabili `final` **non vengono inizializzate di default dal compilatore**.

Una variabile `final` **deve essere assegnata esplicitamente esattamente una sola volta**, altrimenti il codice non compila.

Questo vale sia per:

- **variabili final di istanza**
- **variabili di classe static final**

> **Note**
>
> Possiamo assegnare un valore `null` a una variabile final di istanza o di classe, purché venga impostato esplicitamente.

Java impone questa regola perché una variabile `final` rappresenta un valore che deve essere *noto e fissato* prima dell'uso.

`Final Instance Variables`

Una **variabile final di istanza** deve essere assegnata **esattamente una sola volta**, e l'assegnazione deve avvenire in *una* delle seguenti modalità:

1. **Nel punto di dichiarazione**
2. **In un blocco di inizializzazione d'istanza**
3. **All'interno di ogni costruttore**

Se la classe ha *più costruttori*, la variabile deve essere assegnata in **tutti**.

- Esempio:

```java
public class Person {
    final int id;   // deve essere assegnata prima che il costruttore termini
    String name;

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

    // Costruttore 2
    public Person() {
        this.id = 0;          // richiesto anche qui
        this.name = "Unknown";
    }
}
```

> **Warning**
>
> Provare a compilare senza assegnare `id` dentro **ogni** costruttore produce un errore a compile-time: *variable id might not have been initialized*

Variabili di classe `static final` (Costanti)

Una **variabile static final** appartiene alla classe, non a una specifica istanza.

Deve anch'essa essere assegnata esattamente una sola volta, ma l'assegnazione può avvenire in uno dei seguenti punti:

1. **Nel punto di dichiarazione**
2. **Dentro un blocco di inizializzazione statico**

- Esempio:

```
public class AppConfig {

    static final int TIMEOUT = 5000;    // assegnata nella dichiarazione

    static final String VERSION;        // assegnata nel blocco static

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

Tentare di assegnare una `static final` in un costruttore non è consentito.

Regole chiave per i campi `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

Esempio di situazione **illegale**:

```
public class Example {
    final int x;        // non inizializzata
}

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

Perché le variabili `final` non vengono inizializzate di default?

Perché:

- Il loro valore deve essere **noto e immutabile**, e
- Java deve garantire che il valore sia impostato **prima dell'uso**,
- L'inizializzazione di default creerebbe una situazione in cui `0`, `null`, o `false` potrebbero diventare involontariamente il valore permanente.

Per questo Java costringe gli sviluppatori a inizializzare esplicitamente i campi `final`.

> **Tip**
>
> `final` significa **assegnato una volta**, non **oggetto immutabile**.
>
> Un reference final può comunque puntare a un oggetto mutabile.

```
final List<String> list = new ArrayList<>();
list.add("ok");       // consentito
list = new ArrayList<>(); // ❌ non puoi riassegnare il reference
```

6.2.3 Variabili locali

Le **variabili locali** sono variabili definite all'interno di un `costruttore`, di un `metodo` o di un `blocco di inizializzazione`;

Le variabili locali non hanno valori di default e devono essere inizializzate prima di poter essere usate. Se provi a usare una variabile locale non inizializzata, il compilatore segnalerà un ERRORE.

- Esempio

```
public int localMethod {

    int firstVar = 25;
    int secondVar;
    secondVar = 35;
    int firstSum = firstVar + secondVar;    // OK: entrambe le variabili sono inizializzate pr

    int thirdVar;
    int secondSum = firstSum + thirdVar;    // ERROR: la variabile thirdVar non è stata inizia
}
```

6.2.3.1 Inferire i tipi con var

In certe condizioni puoi usare la keyword **var** al posto del tipo appropriato quando dichiari variabili **locali**;

> **Warning**
>
> - **var** NON è una parola riservata in Java;
> - **var** può essere usata solo per variabili locali: NON può essere usata per **parametri del costruttore**, **variabili di istanza** o **parametri dei metodi**;
> - Il compilatore inferisce il tipo guardando SOLO il codice **sulla riga della dichiarazione**; una volta inferito il tipo, non puoi riassegnare a un altro tipo.

- Esempio

```
public int localMethod {

    var inferredInt = 10;   // Il compilatore inferisce int dal contesto
    inferredInt = 25;       // OK

    inferredInt = "abcd";   // ERROR: il compilatore ha già inferito il tipo della variabile c

    var notInferred;
    notInferred = 30;       // ERROR: per inferire il tipo, il compilatore guarda SOLO la riga

    var first, second = 15; // ERROR: var non può essere usata per definire due variabili nell

    var x = null;           // ERROR: var non può essere inizializzata con null ma può essere
}
```

> **Warning**
>
> Le variabili locali **non** ricevono mai valori di default. Le variabili di istanza e di classe (static) **sì**, sempre.

6.3 Tipi wrapper

In Java, i **tipi wrapper** sono rappresentazioni a oggetti degli otto tipi primitivi.

Ogni primitivo ha una corrispondente classe wrapper nel package `java.lang`:

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

Gli oggetti wrapper sono immutabili — una volta creati, il loro valore non può cambiare.

6.3.1 Scopo dei tipi wrapper

- Consentono di usare i primitivi in contesti che richiedono oggetti (es. collezioni, generics).
- Forniscono metodi di utilità per parsing, conversione e manipolazione dei valori.
- Supportano costanti come `Integer.MAX_VALUE` o `Double.MIN_VALUE`.

6.3.2 Autoboxing e unboxing

Da Java 5, il compilatore converte automaticamente tra primitivi e wrapper:

- **Autoboxing**: primitivo → wrapper
- **Unboxing**: wrapper → primitivo

```
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 **non** esegue mai `autoboxing + widening/narrowing` in un solo passo.

> **Warning**
>
> - **AUTOBOXING** e **cast implicito** non sono consentiti nella stessa istruzione: non puoi fare entrambe le cose contemporaneamente. (vedi esempio sopra)
> - Questa regola vale anche nelle chiamate ai metodi.

6.3.3 Parsing e conversione

I wrapper forniscono metodi statici per convertire stringhe o altri tipi in primitivi:

```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**
>
> I metodi **parseXxx()** restituiscono un primitivo mentre **valueOf()** restituisce un oggetto wrapper.

6.3.4 Metodi di supporto

Tutte le classi wrapper numeriche estendono la classe `Number` e, per questo, ereditano alcuni metodi di supporto come: `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, `doubleValue()`.

Le classi wrapper `Boolean` e `Character` includono: `booleanValue()` e `charValue()`.

- Esempio:

```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 Valori null

A differenza dei primitivi, i tipi wrapper possono contenere **null**.
Tentare di fare unboxing di null causa una `NullPointerException`:

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

6.4 Uguaglianza in Java

Java fornisce due meccanismi distinti per verificare l'uguaglianza:

- `==` (operatore di uguaglianza)
- `.equals()` (metodo definito in `Object` e ridefinito in molte classi)

Capirne la differenza è essenziale.

6.4.1 Uguaglianza con i tipi primitivi

Per i **valori primitivi** (`int` , `double` , `char` , `boolean` , ecc.),
l'operatore `==` confronta il loro reale **valore numerico o booleano**.

Esempio:

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

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

6.4.1.1 Punti chiave

- `==` esegue un **confronto di valori** per i primitivi.
- I tipi primitivi **non** hanno un metodo `.equals()` .
- Tipi primitivi misti seguono le **regole di promozione** numerica
  (es. `int == long` → `int` promosso a `long`).

6.4.2 Uguaglianza con i tipi reference

Con gli oggetti (tipi reference), il significato di `==` cambia.

6.4.2.1 `==` (Confronto di identità)

`==` verifica se **due riferimenti puntano allo stesso oggetto in memoria**.

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

System.out.println(s1 == s2);       // false → oggetti diversi
```

Anche se i contenuti sono identici, `==` è false a meno che entrambe le variabili non si riferiscano
allo stesso identico oggetto.

6.4.2.2 `.equals()` (Confronto logico)

Molte classi ridefiniscono `.equals()` per confrontare i **valori**, non gli indirizzi di memoria.

```
System.out.println(s1.equals(s2)); // true → stesso contenuto
```

6.4.2.3 Punti chiave

- `.equals()` è definito in `Object` .
- Se una classe *non* ridefinisce `.equals()` , si comporta come `==` .
- Classi come `String` , `Integer` , `List` , ecc. ridefiniscono `.equals()`
  per fornire un confronto di valori significativo.

6.4.3 String Pool e uguaglianza

I letterali String sono memorizzati nello **String pool**, quindi letterali identici che si riferiscono **allo stesso oggetto**.

```java
String a = "Java";
String b = "Java";
System.out.println(a == b);        // true → stesso literal nel pool
```

Ma usare `new` crea un oggetto diverso:

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

System.out.println(x == y);        // false → x non è nello pool
System.out.println(x.equals(y));   // true
```

Errori comuni

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

System.out.println(x == y);        // false → x e y non sono lo stesso a compile-time

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

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

Warning

Qualsiasi String creata a **runtime** *non* entra nel pool automaticamente. Si usi `intern()` se si vuole il pooling.

Tip

`"Hello" == "Hel" + "lo"` → true (costante a compile-time)

`"Hello" == getHello()` → false (concatenazione a runtime)

```java
String x = "Hello";
String y = "Hel" + "lo";    // compile-time → stesso literal
String z = "Hel";
z += "lo";                  // runtime → nuova String

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

6.4.3.1 Il metodo intern

Puoi anche dire a Java di usare una String dallo String Pool (nel caso esista già) tramite il metodo `intern()`:

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

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

6.4.4 Uguaglianza con i Tipi Wrapper

Le classi wrapper (`Integer`, `Double`, `Boolean`, ecc.) si comportano come oggetti normali.

Pertanto:

- `==` → confronta **i riferimenti degli oggetti**
- `.equals()` → confronta **i valori numerici**

Esempio:

```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 → oggetti diversi

System.out.println(c.equals(d));    // true → stesso valore numerico
```

Poiché, come precedentemente ricordato, tutte le classi wrapper sono **immutabili**, il loro valore interno **non può essere modificato** una volta create.

Operazioni che sembrano modificare un wrapper in realtà creano **un nuovo oggetto**.

Esempio:

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

Questo è concettualmente equivalente a:

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

Quindi un **nuovo oggetto `Integer`** viene creato e assegnato a `i`.

6.4.4.1 Caching dei wrapper

Per ridurre l'uso della memoria e la creazione di oggetti, Java **riutilizza alcune istanze di wrapper**.

I seguenti valori sono memorizzati in cache:

- Tutti i valori `Boolean` (`true` e `false`)
- Tutti i valori `Byte`
- Tutti i valori `Character` da `\u0000` a `\u007f` (0–127)
- Tutti i valori `Short` da **−128 a 127**
- Tutti i valori `Integer` da **−128 a 127**

A causa di questo meccanismo di caching:

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

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

Entrambe le variabili fanno riferimento allo **stesso oggetto in cache**.

Tuttavia, valori al di fuori dell'intervallo della cache producono **oggetti distinti**:

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

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

6.4.4.2 La keyword `new` aggira la cache

Quando un oggetto wrapper viene creato usando `new`, **una nuova istanza viene sempre creata**, anche se esiste un valore in cache.

Esempio:

```java
Integer i = 10;           // oggetto in cache
Integer j = 10;           // stesso oggetto in cache
Integer k = new Integer(10); // nuovo oggetto (non in cache)

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

Tuttavia, i costruttori dei wrapper sono stati **deprecati in Java 9** e marcati per la rimozione.
Il codice moderno dovrebbe usare **autoboxing** o metodi factory come `Integer.valueOf()`.

6.4.4.3 Confronto dei wrapper

Quando due riferimenti wrapper vengono confrontati usando `==`, il risultato dipende dal fatto che **si riferiscano allo stesso oggetto**, non dal fatto che i loro valori siano uguali.

Pertanto, i test di uguaglianza tra wrapper dovrebbero normalmente usare:

```java
equals()
```

invece di `==`.

6.4.4.4 Tipi wrapper diversi non possono essere confrontati

Oggetti wrapper di **tipi diversi** non possono essere confrontati usando `==`.

Esempio:

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

b == i;   // errore di compilazione
```

Gli operandi devono essere **tipi compatibili**, altrimenti il confronto non è valido.

> **Warning**
>
> Fare molta attenzione quando si confrontano oggetti wrapper con `==`.
> A causa del caching dei wrapper, i confronti possono a volte restituire `true` e a volte `false` a seconda del valore.

6.4.5 Uguaglianza e `null`

- `== null` è sempre sicuro.
- Chiamare `.equals()` su un reference `null` genera una `NullPointerException`.

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

6.4.6 Tabella riepilogativa

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. Flusso di controllo

Indice

- 7.1 L'istruzione if
- 7.2 L'istruzione Switch & la Switch Expression
  - 7.2.1 La variabile target dello switch può essere
  - 7.2.2 Valori case accettabili
  - 7.2.3 Compatibilità di tipo tra selector e case
  - 7.2.4 Pattern Matching nello Switch
    - 7.2.4.1 Nomi delle variabili e scope tra i rami
    - 7.2.4.2 Ordinamento, dominanza ed esaustività negli switch con pattern
- 7.3 Due forme di switch: switch Statement vs switch Expression
  - 7.3.1 L'istruzione switch
    - 7.3.1.1 Comportamento di fall-through
  - 7.3.2 L'espressione switch
    - 7.3.2.1 yield nei blocchi di espressione switch
    - 7.3.2.2 Esaustività per le espressioni switch
- 7.4 Gestione di null

Il **flusso di controllo** in Java si riferisce all'**ordine in cui le singole istruzioni, i comandi o le chiamate a metodo vengono eseguiti** durante l'esecuzione del programma.

Per impostazione predefinita, le istruzioni vengono eseguite sequenzialmente dall'alto verso il basso, ma le istruzioni di controllo del flusso consentono al programma di **prendere decisioni**, **ripetere azioni** o **diramare i percorsi di esecuzione** in base a condizioni.

Java fornisce tre categorie principali di costrutti di controllo del flusso:

- **Istruzioni decisionali** — `if`, `if-else`, `switch`
- **Istruzioni di iterazione** — `for`, `while`, `do-while` e il `for` avanzato
- **Istruzioni di diramazione** — `break`, `continue` e `return`

> **Tip**
>
> Comprendere il flusso di controllo è essenziale per vedere come i dati si muovono all'interno del programma e come ogni decisione logica viene valutata passo dopo passo.

7.1 L'istruzione `if`

L'istruzione `if` è una struttura condizionale di controllo del flusso che esegue un blocco di codice solo se una specifica espressione booleana viene valutata come `true`.

L'istruzione consente al programma di prendere decisioni a runtime.

Sintassi:

```
if (condition) {
    // eseguito solo quando la condizione è true
}
```

Una clausola `else` opzionale gestisce il percorso alternativo:

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

Più condizioni possono essere concatenate usando `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 condizione di `if` deve essere valutata come **boolean**; i tipi numerici o gli oggetti non possono essere usati direttamente come condizioni.
>
> Le parentesi graffe `{}` sono opzionali per singole istruzioni ma sono fortemente consigliate per prevenire sottili errori di logica.
>
> Una catena `if-else` viene valutata dall'alto verso il basso, e viene eseguito solo il primo ramo con una condizione valutata come `true`.

7.2 L'istruzione `switch` & la `switch Expression`

Il costrutto `switch` è una struttura di controllo del flusso che seleziona un ramo tra più alternative in base al valore di un'espressione (il **selector**).

Rispetto a lunghe catene di `if-else-if`, uno `switch`:

- È spesso **più facile da leggere** quando si testano molti valori discreti (costanti, enum, stringhe).
- Può essere **più sicuro e più conciso** quando usato come **espressione switch**

perché:

- Produce un valore.
- Il compilatore può imporre **esaustività** e **coerenza di tipo**.

Java 21 supporta:

- La `switch` classica come **istruzione** (solo controllo del flusso).
- La `switch` come **Expression** (produce un risultato).
- **Pattern matching** dentro `switch`, inclusi type pattern e guard.

Entrambe le forme di `switch` condividono le stesse regole riguardanti il selector (la **variabile target** dello switch) e i valori `case` accettabili.

7.2.1 La `variable target` dello switch può essere

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 consentiti** come type selector per switch:
>
> - `boolean`
> - `long`
> - `float`
> - `double`

7.2.2 Valori `case` accettabili

Per uno switch `non-pattern`, ogni etichetta `case` deve essere una **costante a compile-time compatibile con il tipo del selector**.

Sono consentite, come etichette `case`:

- **Letterali** come `0`, `'A'`, `"ON"`.
- **Costanti enum**, ad es. `RED` o `Color.GREEN`.
- **Variabili costanti final** (costanti a compile-time).

Una costante a compile-time:

- Deve essere dichiarata con `final` e inizializzata nella stessa istruzione.
- Il suo inizializzatore deve a sua volta essere un'espressione costante (tipicamente usando letterali e altre costanti a compile-time).

7.2.3 Compatibilità di tipo tra `selector` e `case`

Il tipo del `selector` e ogni etichetta `case` devono essere compatibili:

- Le costanti numeriche dei case devono essere entro l'intervallo del tipo del selector.
- Per un selector `enum`, le etichette `case` devono essere costanti di quell' `enum`.
- Per un selector `String`, le etichette `case` devono essere costanti stringa.

7.2.4 Pattern Matching nello Switch

Lo switch in Java 21 supporta il `pattern matching`, includendo:

- **Type pattern**: `case String s`
- **Guarded pattern**: `case String s when s.length() > 3`
- **Null pattern**: `case null`

Esempio:

```java
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";
    };
}
```

Punti chiave:

- Ogni pattern introduce una `pattern variable` (come `i` o `s`).
- Le pattern variable sono in scope solo all'interno del proprio `ramo` (o dei percorsi in cui è noto che il pattern corrisponde).
- L'ordine è importante a causa della **dominanza**: **i pattern più specifici devono precedere quelli più generali**.

7.2.4.1 Nomi delle variabili e scope tra i rami

Con il `pattern matching`, la variabile di pattern esiste solo nello scope del ramo in cui è definita.

Questo significa che puoi riutilizzare lo stesso nome di variabile in diversi rami `case` senza che i nomi entrino in conflitto.

- Esempio:

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

> **Note**
>
> Quest'ultimo esempio non restituisce un valore, quindi è in realtà una **istruzione switch**, non una switch Expression.

7.2.4.2 Ordinamento, dominanza ed esaustività negli switch con pattern

Quando si gestisce il pattern matching, l'ordinamento dei rami è cruciale a causa della **dominanza** e del potenziale **codice irraggiungibile**.

Un pattern più generale **non** deve apparire prima di uno più specifico, altrimenti quello specifico diventa irraggiungibile.

- Esempio (ramo irraggiungibile):

```java
return switch (o) {
    case Object obj -> "object";
    case String s -> "string"; // ❌ DOES NOT COMPILE: irraggiungibile, String è già intercett
};
```

- Un altro esempio con una guard:

```java
return switch (o) {
    case Integer a -> "First";
    case Integer a when a > 0 -> "Second"; // ❌ DOES NOT COMPILE: irraggiungibile, il primo c
    // ...
};
```

Quando si usa il pattern matching, gli switch devono essere **esaustivi**; cioè, devono gestire tutti i possibili valori del selector.

Questo può essere ottenuto tramite:

- Fornire un case `default` che gestisce tutti i valori non corrispondenti a nessun altro case.
- Fornire una clausola `case terminale` con un tipo di pattern che corrisponde al tipo reference del selector.
- Esempio (non esaustivo):

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

switch (number) {
    case Short s -> System.out.println("A"); // ❌ DOES NOT COMPILE: non esaustivo, il selecto
}
```

Per correggere questo, puoi:

- Cambiare il tipo reference di `number` in `Short` (allora l'esaustività è soddisfatta dal singolo case).
- Aggiungere una clausola `default` che copre tutti i valori rimanenti.
- Aggiungere una clausola `case` finale che copre il tipo della variabile selector, per esempio:

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

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

> **Warning**
>
> Il seguente esempio, che usa sia una clausola `default` sia una clausola finale con lo stesso tipo della variabile selector, **non** compila: il compilatore considera uno dei due case come sempre dominante rispetto all'altro.

```
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 Due forme di `switch`: `switch` Statement vs `switch` Expression

7.3.1 L'istruzione Switch

Una **istruzione switch** è usata come costrutto di controllo del flusso.

Non viene valutata, di per sé, come un valore, anche se i suoi rami possono contenere istruzioni `return` che ritornano dal metodo contenitore.

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

Punti chiave:

- Ogni clausola `case` include uno o più valori corrispondenti separati da virgole `,`. Segue un separatore, che può essere due punti `:` o, meno comunemente per le `istruzioni`, l'operatore

freccia `->`. Infine, un'espressione o un blocco (racchiuso in `{}`) definisce il codice da eseguire quando si verifica una corrispondenza. **Se si usa l'operatore freccia per una clausola, si deve usare per tutte le clausole in quella istruzione switch**.

- Il fall-through è possibile per i `case` in stile "due punti" a meno che un ramo usi `break`, `return` o `throw`. Quando presente, `break` termina lo switch dopo l'esecuzione del suo case; senza di esso, l'esecuzione continua, in ordine, nei rami successivi.
- Una clausola `default` è opzionale e può apparire ovunque nell'istruzione switch. Viene eseguita se non c'è corrispondenza per i case precedenti.
- Un'istruzione switch non produce un valore come nell'Expression; non puoi assegnare un'istruzione switch direttamente a una variabile.

7.3.1.1 Comportamento di Fall-Through

Con i `case` in stile "due punti", l'esecuzione salta all'etichetta `case` corrispondente.

Se non c'è un `break`, continua nel case successivo finché non viene incontrato un `break`, `return` o `throw`.

```
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");
}
```

Output:

```
2
3
```

> **Note**
>
> Se nell'esempio precedente rimuoviamo il `break` sul `case 3`, verrà stampato anche il messaggio del ramo `default`.

7.3.2 L'espressione Switch

Una **espressione switch** produce sempre un singolo valore come suo risultato.

- Esempio:

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

Punti chiave:

- Ogni clausola `case` include uno o più valori corrispondenti separati da virgole `,`, seguiti dall'operatore freccia `->`. Poi un'espressione o un blocco (racchiuso in `{}`) definisce il risultato per quel ramo.
- Quando usata con un'assegnazione o un'istruzione `return`, un'espressione switch richiede un punto e virgola finale `;` dopo l'espressione.
- Non c'è fall-through tra i rami in stile "freccia". Ogni ramo corrispondente viene eseguito esattamente una volta.
- Un'espressione switch deve essere **esaustiva**: tutti i possibili valori del selector devono essere coperti (tramite case espliciti e/o `default`).

- Il tipo del risultato deve essere coerente tra tutti i rami. Per esempio, se un ramo produce un `int`, gli altri rami devono produrre valori compatibili con `int`.

7.3.2.1 `yield` nei blocchi di espressione switch

Quando un ramo di un'espressione switch usa un blocco invece di una singola espressione, devi usare `yield` per fornire il risultato di quel ramo.

```
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` è usato solo nelle Expressions switch. `break value;` non è consentito come modo per restituire un valore da un'espressione switch.

7.3.2.2 Esaustività per le espressioni switch

Poiché un'espressione switch deve restituire un valore, deve anche essere **esaustiva**; in altre parole, deve gestire tutti i possibili valori del selector.

Puoi assicurare questo tramite:

- Fornire un case `default`.
- Per un selector enum: coprire esplicitamente tutte le costanti enum.
- Per tipi sealed o pattern switch: coprire tutti i sottotipi permessi o fornire un `default`.

Esempio, esaustivo tramite `default`:

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

7.4 Gestione di null

Switch classico (senza pattern)

Se l'espressione selector di uno switch classico (senza pattern matching) viene valutata come `null`, viene lanciata una `NullPointerException` a runtime.

Per evitare questo, controlla `null` prima di fare lo switch:

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

Pattern switch (con `case null`)

Con il pattern matching, puoi gestire `null` direttamente dentro lo switch:

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

> **Note**
>
> Per le Expressions switch:
>
> Se non gestisci `null` e il selector è `null`, viene lanciata una `NullPointerException`.
>
> Usare `case null` rende lo switch esplicitamente null-safe.

> **Warning**
>
> Ogni volta che `case null` viene usato in uno switch, lo switch viene trattato come un `pattern switch`, e si applicano tutte le regole per i pattern switch (incluse esaustività e dominanza).

---

8. Costrutti di iterazione in Java

Indice

- 8.1 Il ciclo while
- 8.2 Il ciclo do-while
- 8.3 Il ciclo for
- 8.4 Il ciclo for-each avanzato
- 8.5 Cicli annidati
- 8.6 Cicli infiniti
- 8.7 break e continue
- 8.8 Cicli etichettati
- 8.9 Ambito delle variabili di ciclo
- 8.10 Codice irraggiungibile dopo break continue e return
  - 8.10.1 Codice irraggiungibile dopo break
  - 8.10.2 Codice irraggiungibile dopo continue
  - 8.10.3 Codice irraggiungibile dopo return

Java fornisce diversi **costrutti di iterazione** che consentono l'esecuzione ripetuta di un blocco di codice finché una condizione è vera.

I cicli sono essenziali per l'iterazione, l'attraversamento di strutture dati, calcoli ripetuti e l'implementazione di algoritmi.

8.1 Il ciclo `while`

Il ciclo `while` valuta la propria **condizione booleana prima di ogni iterazione**.

Se la condizione è `false` fin dall'inizio, il corpo non viene mai eseguito.

Sintassi

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

- La condizione deve essere valutata come un **booleano**.
- Il ciclo può essere eseguito zero o più volte.
- Tra gli errori comuni c'è il dimenticare di aggiornare la variabile del ciclo, causando un ciclo infinito.
- Esempio:

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

Output:

```
0
1
2
```

8.2 Il ciclo `do-while`

Il ciclo `do-while` valuta la propria condizione **dopo** aver eseguito il corpo, assicurando che **il corpo venga eseguito almeno una volta**.

Sintassi

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

> **Tip**
>
> `do-while` richiede un **punto e virgola** dopo la parentesi di chiusura.

- Esempio:

```
int x = 5;
do {
    System.out.println(x);
    x--;
} while (x > 5); // il body è eseguito almeno una volta anche se la condizione è false
```

Output:

```
5
```

8.3 Il ciclo `for`

Il ciclo `for` tradizionale è più adatto per cicli con una variabile contatore.

È composto da tre parti: `inizializzazione`, `condizione`, `aggiornamento`.

Sintassi

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

- L' `inizializzazione` viene **eseguita una volta prima dell'inizio del ciclo**.
- La `condizione` viene **valutata prima di ogni iterazione**.
- L' `aggiornamento` viene **eseguito dopo ogni iterazione**.
- `Inizializzazione` e `aggiornamento` possono contenere più istruzioni separate da virgole.
- Le variabili nell'inizializzazione devono essere **tutte dello stesso tipo**.
- Qualsiasi componente può essere omesso, ma i punti e virgola rimangono.
- Esempio:

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

Omettendo parti:

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

Istruzioni multiple:

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

8.4 Il ciclo `for-each` avanzato

Il `for` avanzato semplifica l'iterazione su `array` e `collezioni`.

Sintassi

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

- La variabile di ciclo è di sola lettura rispetto alla collezione sottostante.
- Funziona con qualsiasi `Iterable` o array.
- Non può rimuovere elementi senza un iteratore.
- Esempio:

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

Output:

```
A
B
C
```

8.5 Cicli annidati

I cicli possono essere annidati; ciascuno mantiene le proprie variabili e condizioni.

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

Output:

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

8.6 Cicli infiniti

Un ciclo è infinito quando la sua condizione viene sempre valutata come `true` o è omessa.

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

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

> **Tip**
>
> I cicli infiniti devono contenere `break`, `return` o un controllo esterno.

8.7 `break` e `continue`

break

Esce immediatamente dal ciclo più interno.

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

continue

Salta il resto del corpo del ciclo e continua alla successiva iterazione.

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

> **Note**
>
> `break` e `continue` si applicano al ciclo più vicino a meno che non vengano usate etichette.

8.8 Cicli etichettati

Un' `etichetta` (identificatore + due punti) può essere applicata a un ciclo per consentire a break/continue di influire sui cicli esterni.

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

- Esempio:

```
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 Ambito delle variabili di ciclo

- Le variabili dichiarate nell'intestazione del ciclo hanno ambito limitato a quel ciclo.
- Le variabili dichiarate all'interno del corpo esistono solo all'interno di quel blocco.

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

8.10 Codice irraggiungibile dopo `break`, `continue` e `return`

Qualsiasi istruzione posizionata **dopo** `break`, `continue` o `return` nello stesso blocco è considerata irraggiungibile e non compila.

8.10.1 Codice irraggiungibile dopo `break`

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

8.10.2 Codice irraggiungibile dopo `continue`

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

> **Note**
>
> `continue` salta alla successiva iterazione, quindi il codice successivo non viene mai eseguito.

8.10.3 Codice irraggiungibile dopo `return`

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

> **Note**
>
> `return` esce immediatamente dal metodo; nessuna istruzione può seguirlo.

◀ 7. Flusso di controllo | ▲ Index | 9. Stringhe in Java ▶

Core Standard APIs

9. Stringhe in Java

Indice

- 9.1 Stringhe & Text Blocks
    - 9.1.1 Stringhe
        - 9.1.1.1 Inizializzare le stringhe
        - 9.1.1.2 Lo String Pool
        - 9.1.1.3 Caratteri speciali e sequenze di escape
        - 9.1.1.4 Regole per la concatenazione di stringhe
        - 9.1.1.5 Regole di concatenazione
    - 9.1.2 Text Blocks (da Java 15)
        - 9.1.2.1 Formattazione: whitespace essenziale vs incidentale
        - 9.1.2.2 Conteggio righe, righe vuote e line break
        - 9.1.2.3 Text Blocks e caratteri di escape
        - 9.1.2.4 Errori comuni con correzioni
- 9.2 Metodi principali delle stringhe
    - 9.2.1 Indicizzazione delle stringhe
    - 9.2.2 Metodo length
    - 9.2.3 Regole dei limiti: indice iniziale vs indice finale
    - 9.2.4 Metodi che usano solo l'indice iniziale (inclusivo)
    - 9.2.5 Metodi con inizio inclusivo / fine esclusivo
    - 9.2.6 Metodi che operano sull'intera stringa
    - 9.2.7 Accesso ai caratteri
    - 9.2.8 Ricerca
    - 9.2.9 Metodi di sostituzione
    - 9.2.10 Suddivisione e unione
    - 9.2.11 Metodi che restituiscono array
    - 9.2.12 Indentazione
    - 9.2.13 Esempi aggiuntivi

9.1 Stringhe & Text Blocks

9.1.1 Stringhe

9.1.1.1 Inizializzare le stringhe

In Java, una **String** è un oggetto della classe `java.lang.String`, usato per rappresentare una sequenza di caratteri.

Le stringhe sono **immutabili**: una volta create, il loro contenuto non può essere cambiato. Qualsiasi operazione che sembra modificare una stringa in realtà ne crea una nuova.

Puoi creare e inizializzare le stringhe in diversi modi:

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

> **Note**
>
> - I letterali stringa sono memorizzati nello `String pool`, un'area speciale di memoria usata per evitare di creare oggetti stringa duplicati.
> - L'uso della keyword `new` crea sempre un nuovo oggetto al di fuori del pool.

9.1.1.2 Lo String Pool

Poiché gli oggetti `String` sono immutabili e ampiamente usati, potrebbero facilmente occupare una grande quantità di memoria in un programma Java.

Per ridurre la duplicazione, Java riutilizza tutte le stringhe dichiarate come letterali (vedi l'esempio sopra), memorizzandole in un'area dedicata della JVM nota come **String Pool** o **Intern Pool**.

Per una spiegazione più approfondita e esempi, controlla il paragrafo: **"6.4.3 String Pool and Equality"** nel capitolo: Istanziazione dei tipi.

9.1.1.3 Caratteri speciali e sequenze di escape

Le stringhe possono contenere caratteri di escape, che permettono di includere simboli speciali o caratteri di controllo (caratteri con un significato speciale in Java).
Una sequenza di escape inizia con un backslash `\`.

> **Note**
>
> **Tabella dei caratteri speciali & sequenze di escape nelle stringhe**

Escape	Significato	Esempio Java	Risultato
`\"`	doppia virgoletta	`"She said \"Hi\""`	`She said "Hi"`
`\\`	backslash	`"C:\\Users\\Alex"`	`C:\Users\Alex`
`\n`	a capo (LF)	`"Hello\nWorld"`	`Hello` + line break + `World`
`\r`	ritorno carrello (CR)	`"A\rB"`	`CR before B`
`\t`	tab	`"Name\tAge"`	`Name Age`
`\'`	virgoletta singola	`"It\'s ok"`	`It's ok`
`\b`	backspace	`"AB\bC"`	`AC` (il `B` viene rimosso visivamente)
`\uXXXX`	unità di codice Unicode	`"\u00A9"`	`©`

9.1.1.4 Regole per la concatenazione di stringhe

Come introdotto nel capitolo su Operatori Java, il simbolo `+` normalmente rappresenta l'**addizione aritmetica** quando viene usato con operandi numerici.

Tuttavia, quando applicato alle **String**, lo stesso operatore esegue la **concatenazione di stringhe** — crea una nuova stringa unendo gli operandi tra loro.

Poiché l'operatore `+` può apparire in espressioni in cui sono presenti sia numeri sia stringhe, Java applica un insieme specifico di regole per determinare se `+` significa **addizione numerica** o **concatenazione di stringhe**.

9.1.1.5 Regole di concatenazione

- Se entrambi gli operandi sono numerici, `+` esegue l'**addizione numerica**.
- Se almeno un operando è una `String`, l'operatore `+` esegue la **concatenazione di stringhe**.
- La valutazione è strettamente da sinistra a destra, perché `+` è **associativo a sinistra**.

Questo significa che, una volta che una `String` appare sul lato sinistro dell'espressione, tutte le successive operazioni `+` diventano concatenazioni.

> **Tip**
>
> Poiché la valutazione è da sinistra a destra, la posizione del primo operando `String` determina come viene valutato il resto dell'espressione.

- Esempi

```
// *** 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 (da Java 15)

Un text block è un letterale stringa multi-linea introdotto per semplificare la scrittura di stringhe grandi (come HTML, JSON o codice) senza la necessità di molte sequenze di escape.

Un text block inizia e termina con tre doppi apici (`"""`).

Puoi usare i text block ovunque useresti le stringhe.

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

Note

- I text block includono automaticamente line break e indentazione per la leggibilità. I newline sono normalizzati a `\n`.
- Le doppie virgolette all'interno del blocco di solito non richiedono escape.
- Il compilatore interpreta il contenuto tra le triple virgolette di apertura e chiusura come il valore della stringa.

9.1.2.1 Formattazione: whitespace essenziale vs incidentale

- **Whitespace essenziale**: spazi e newline che fanno parte del contenuto della stringa.
- **Whitespace incidentale**: indentazione nel codice sorgente che non consideri concettualmente parte del testo.

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

Important

- **Carattere più a sinistra (baseline)**: la posizione del primo carattere non-spazio su tutte le righe (o la `"""` di chiusura) definisce la baseline di indentazione. Gli spazi a sinistra di questa baseline sono considerati incidentali e vengono rimossi.
- La riga immediatamente successiva alla `"""` di apertura non è inclusa nell'output se è vuota (formattazione tipica).
- Il newline prima della `"""` di chiusura è incluso nel contenuto.
  Nell'esempio sopra, la stringa risultante termina con un newline dopo `"Line 3"`: in totale ci sono 4 righe.

Output con numeri di riga (mostrando la riga vuota finale):

```
1: Line 1
2: Line 2
3: Line 3
4:
```

Per sopprimere il newline finale:

- Usa un backslash di continuazione riga alla fine dell'ultima riga di contenuto.
- Metti le triple virgolette di chiusura sulla stessa riga dell'ultimo contenuto.

```
String textNoTrail_1 = """
        Line 1
        Line 2
        Line 3\
        """;

// OR

String textNoTrail_2 = """
        Line 1
        Line 2
        Line 3""";
```

9.1.2.2 Conteggio righe, righe vuote e line break

- Ogni line break visibile dentro il blocco diventa `\n`.
- Le righe vuote dentro il blocco vengono preservate.

```
String textNoTrail_0 = """
        Line 1
        Line 2 \n
        Line 3

        Line 4
        """;
```

Output:

```
1: Line 1
2: Line 2
3:
4: Line 3
5:
6: Line 4
7:
```

9.1.2.3 Text Blocks e caratteri di escape

Le sequenze di escape funzionano ancora dentro i text block quando necessario (per esempio, per backslash o caratteri di controllo espliciti).

```
String json = """
    {
      "name": "Alice",
      "path": "C:\\\\Users\\\\Alice"
    }\
    """;
```

Puoi anche formattare un text block usando placeholder e `formatted()`:

```
String card = """
    Name: %s
    Age:  %d
    """.formatted("Alice", 30);
```

9.1.2.4 Errori comuni (con correzioni)

```
// ❌ 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 Metodi principali delle stringhe

9.2.1 Indicizzazione delle stringhe

Le stringhe in Java usano l'**indicizzazione a base zero**, il che significa:

- Il primo carattere è all'indice `0`
- L'ultimo carattere è all'indice `length() - 1`
- Accedere a qualsiasi indice fuori da questo intervallo causa una `StringIndexOutOfBoundsException`
- Esempio:

```
String s = "Java";
// Indexes:  0    1    2    3
// Chars:    J    a    v    a

char c = s.charAt(2); // 'v'
```

9.2.2 Metodo `length()`

`length()` restituisce il numero di caratteri nella stringa.

```
String s = "hello";
System.out.println(s.length());  // 5
```

L'ultimo indice valido è sempre `length() - 1`.

9.2.3 Regole dei limiti: indice iniziale vs indice finale

Molti metodi di `String` usano due indici:

- **Indice iniziale** — inclusivo
- **Indice finale** — esclusivo

In altre parole, `substring(start, end)` include i caratteri dall'indice `start` fino a (ma non includendo) l'indice `end`.

- L'indice iniziale deve essere `>= 0` e `<= length() - 1`
- L'indice finale può essere uguale a `length()` (la "posizione virtuale" dopo l'ultimo carattere).
- L'indice finale non deve superare `length()`.
- L'indice iniziale non deve mai essere maggiore dell'indice finale.
- Esempio:

```
String s = "abcdef";
s.substring(1, 4); // "bcd" (indexes 1,2,3)
```

Questa regola si applica alla maggior parte dei metodi basati su substring.

9.2.4 Metodi che usano solo l'indice iniziale (inclusivo)

Metodo	Descrizione	Parametri	Regola indice	Esempio
substring(int start)	Restituisce la sottostringa da start alla fine	start	start inclusivo	"abcdef".substring(2) → "cdef"
indexOf(String)	Prima occorrenza	—	—	"Java".indexOf("a") → 1
indexOf(String, start)	Inizia la ricerca all'indice	start	start inclusivo	"banana".indexOf("a", 2) → 3
lastIndexOf(String)	Ultima occorrenza	—	—	"banana".lastIndexOf("a") → 5
lastIndexOf(String, fromIndex)	Cerca all'indietro dall'indice	fromIndex	fromIndex inclusivo	"banana".lastIndexOf("a", 3) → 3

9.2.5 Metodi con inizio inclusivo / fine esclusivo

Questi metodi seguono lo stesso comportamento di slicing: `start` incluso, `end` escluso.

Metodo	Descrizione	Firma	Esempio
substring(start, end)	Estrae una parte della stringa	(int start, int end)	"abcdef".substring(1,4) → "bcd"
regionMatches	Confronta regioni di sottostringhe	(toffset, other, ooffset, len)	"Hello".regionMatches(1, "ell", 0, 3) → true
getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)	Copia caratteri in un array di byte	start inclusivo, end esclusivo	Copia i caratteri in [srcBegin, srcEnd)
copyValueOf(char[] data, int offset, int count)	Crea una nuova stringa	offset inclusivo; offset+count esclusivo	Stessa regola di substring

9.2.6 Metodi che operano sull'intera stringa

Metodo	Descrizione	Esempio
toUpperCase()	Versione maiuscola	"java".toUpperCase() → "JAVA"
toLowerCase()	Versione minuscola	"JAVA".toLowerCase() → "java"
trim()	Rimuove whitespace iniziale/finale	" hi ".trim() →"hi"
strip()	Trim Unicode-aware	" hi003".strip() → "hi"
stripLeading()	Rimuove whitespace iniziale	" hi".stripLeading() → "hi"
stripTrailing()	Rimuove whitespace finale	"hi".stripTrailing() → "hi"
isBlank()	Vero se vuota o solo whitespace	" ".isBlank() → true
isEmpty()	Vero se length == 0	"".isEmpty() → true

9.2.7 Accesso ai caratteri

Metodo	Descrizione	Esempio
charAt(int index)	Restituisce il carattere all'indice	"Java".charAt(2) → 'v'
codePointAt(int index)	Restituisce il code point Unicode	Utile per emoji o caratteri oltre BMP

9.2.8 Ricerca

Metodo	Descrizione	Esempio
contains(CharSequence)	Test di sottostringa	"hello".contains("ell") → true
startsWith(String)	Prefisso	"abcdef".startsWith("abc") → true
startsWith(String, offset)	Prefisso all'indice	"abc".startsWith("b", 1) → true
endsWith(String)	Suffisso	"abcdef".endsWith("def") → true

9.2.9 Metodi di sostituzione

Metodo	Descrizione	Esempio
replace(char old, char new)	Sostituisce caratteri	"banana".replace('a','o') → "bonono"
replace(CharSequence old, CharSequence new)	Sostituisce sottostringhe	"ababa".replace("aba","X") → "Xba"
replaceAll(String regex, String replacement)	Sostituzione regex globale	"a1a2".replaceAll("\d","") →"aa"
replaceFirst(String regex, String replacement)	Solo prima corrispondenza regex	"a1a2".replaceFirst("\d","") →"aa2"

9.2.10 Suddivisione e unione

Metodo	Descrizione	Esempio
split(String regex)	Suddivide per regex	"a,b,c".split(",") → ["a","b","c"]
split(String regex, int limit)	Suddivide con limite	limit < 0 mantiene tutte le stringhe vuote finali

9.2.11 Metodi che restituiscono array

Metodo	Descrizione	Esempio
toCharArray()	Restituisce char[]	"abc".toCharArray()
getBytes()	Restituisce byte[] usando encoding di piattaforma/default	"á".getBytes()

9.2.12 Indentazione

Metodo	Descrizione	Esempio
indent(int numSpaces)	Aggiunge (positivo) o rimuove (negativo) spazi dall'inizio di ogni riga; aggiunge anche un line break alla fine se non già presente	str.indent(-20)
stripIndent()	Rimuove tutto il whitespace iniziale incidentale da ogni riga; non aggiunge un line break finale	str.stripIndent()

- Esempio:

```
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());
```

Output:

```
length: 9

a
  b
 c


a
  b
 c
length: 9
*********************
length: 8
 a
 b
 c

a
b
c
length: 5
```

9.2.13 Esempi aggiuntivi

- Esempio 1 — Estrarre `[start, end)`

```
String s = "012345";
System.out.println(s.substring(2, 5));
// includes 2,3,4 → prints "234"
```

- Esempio 2 — Ricerca da un indice iniziale

```
String s = "hellohello";
int idx = s.indexOf("lo", 5); // search begins at index 5
```

- Esempio 3 — Errori comuni

```
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
```

---

10. Array in Java

Indice

- 10.1 Che cos'è un array
  - 10.1.1 Dichiarare gli array
  - 10.1.2 Creare array (istanza)
  - 10.1.3 Valori predefiniti negli array
  - 10.1.4 Accedere agli elementi
  - 10.1.5 Scorciatoie di inizializzazione degli array
    - 10.1.5.1 Creazione di array anonimi
    - 10.1.5.2 Sintassi breve (solo in dichiarazione)
- 10.2 Array multidimensionali (array di array)
  - 10.2.1 Creare un array rettangolare
  - 10.2.2 Creare un array frastagliato (irregolare)
- 10.3 Lunghezza degli array vs lunghezza delle stringhe
- 10.4 Assegnazioni di riferimenti a array
  - 10.4.1 Assegnare riferimenti compatibili
  - 10.4.2 Assegnazioni incompatibili (errori a compile-time)
  - 10.4.3 Rischio runtime della covarianza: ArrayStoreException
- 10.5 Confrontare gli array
- 10.6 Metodi di utilita di Arrays
  - 10.6.1 Arrays.toString
  - 10.6.2 Arrays.deepToString per array annidati
  - 10.6.3 Arrays.sort
  - 10.6.4 Arrays.binarySearch
  - 10.6.5 Arrays.compare
- 10.7 Enhanced for-loop con array
- 10.8 Errori comuni
- 10.9 Riepilogo

10.1 Che cos'è un array

Gli array in Java sono collezioni **a dimensione fissa**, **indicizzate**, **ordinate** di elementi dello stesso tipo.

Sono **oggetti**, anche quando gli elementi contenuti sono primitivi.

10.1.1 Dichiarare gli array

Puoi dichiarare un array in due modi:

```java
int[] a;      // sintassi moderna preferita
int b[];      // legale, stile vetusto
String[] names;
Person[] people;

// [] possono trovarsi prima o dopo il nome: tutte le seguenti dichiarazioni sono equivalenti.

int[] x;
int [] x1;
int []x2;
int x3[];
int x5 [];

// MULTIPLE ARRAY DECLARATIONS

int[] arr1, arr2;   // Dichiara due array di interi

// WARNING:
// QUi arr1 è un int[] e arr2 è solo un int (NON un array!)
int arr1[], arr2;
```

Dichiarare NON crea l'array — crea solo una variabile in grado di referenziarne uno.

10.1.2 Creare array (istanza)

Un array viene creato usando `new` seguito dal tipo dell'elemento e dalla lunghezza dell'array:

```java
int[] numbers = new int[5];
String[] words = new String[3];
```

Regole chiave - La lunghezza deve essere non negativa e specificata al momento della creazione. - La lunghezza non può essere cambiata in seguito. - La lunghezza dell'array può essere qualsiasi espressione `int`.

```java
int size = 4;
double[] values = new double[size];
```

- Esempi illegali di creazione di array:

```java
// 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 Valori predefiniti negli array

Gli array (poiché sono oggetti) ricevono sempre una **inizializzazione predefinita**:

Tipo di elemento	Valore predefinito
Numerico	0
boolean	false
char	'000'
Tipi di riferimento	null

- Esempio:

```java
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 Accedere agli elementi

Gli elementi si accedono usando l'indicizzazione a base zero:

```java
int[] a = new int[3];
a[0] = 10;
a[1] = 20;
System.out.println(a[1]); // 20
```

Eccezione comune
- `ArrayIndexOutOfBoundsException` (runtime)

```java
// int[] x = new int[2];
// System.out.println(x[2]); // ❌ indice 2 out of bounds
```

10.1.5 Scorciatoie di inizializzazione degli array

10.1.5.1 Creazione di array anonimi

```java
int[] a = new int[] {1,2,3};
```

10.1.5.2 Sintassi breve (solo in dichiarazione)

```java
int[] b = {1,2,3};
```

> La sintassi breve `{1,2,3}` può essere usata **solo nel punto di dichiarazione**.

```java
// int[] c;
// c = {1,2,3};  // ❌ does not compile
```

10.2 Array multidimensionali (array di array)

Java implementa gli array multi-dimensionali come **array di array**.

Dichiarazione:

```java
int[][] matrix;
String[][][] cube;
```

10.2.1 Creare un array rettangolare

```java
int[][] rect = new int[3][4]; // 3 righe, 4 colonne
```

10.2.2 Creare un array frastagliato (irregolare)

Puoi creare righe con lunghezze diverse:

```java
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[5];
jagged[2] = new int[1];
```

10.3 Lunghezza degli array vs lunghezza delle stringhe

- Gli array usano `.length` (campo `public final`).
- Le stringhe usano `.length()` (metodo).

> **Tip**
>
> Questo è un classico errore: campi vs metodi.

```java
// int x = arr.length;   // OK
// int y = s.length;     // ❌ does not compile: missing ()
int yOk = s.length();
```

10.4 Assegnazioni di riferimenti a array

10.4.1 Assegnare riferimenti compatibili

```java
int[] a = {1,2,3};
int[] b = a; // entrambi puntano ora allo stesso array
```

Modificare un riferimento influisce sull'altro:

```java
b[0] = 99;
System.out.println(a[0]); // 99
```

10.4.2 Assegnazioni incompatibili (errori a compile-time)

```java
int[] x = new int[3];
// long[] y = x;     // ❌ tipo incompatibile
```

I riferimenti ad array seguono le normali regole di ereditarietà:

```java
String[] s = new String[3];
Object[] o = s;      // OK: arrays are covariant
```

10.4.3 Rischio runtime della covarianza: `ArrayStoreException`

```java
Object[] objs = new String[3];
// objs[0] = Integer.valueOf(5); // ❌ ArrayStoreException a runtime
```

10.5 Confrontare gli array

`==` confronta i riferimenti (identità):

```java
int[] a = {1,2};
int[] b = {1,2};
System.out.println(a == b); // false
```

`equals()` sugli array non confronta i contenuti (si comporta come `==`):

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

Per confrontare i contenuti, usa i metodi di `java.util.Arrays`:

```java
Arrays.equals(a, b);         // shallow comparison
Arrays.deepEquals(o1, o2);   // deep comparison per arrays annidati
```

10.6 Metodi di utilità di `Arrays`

10.6.1 `Arrays.toString()`

```java
System.out.println(Arrays.toString(new int[]{1,2,3})); // [1, 2, 3]
```

10.6.2 `Arrays.deepToString()` (per array annidati)

```
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

- Le stringhe sono ordinate in ordine naturale (lessicografico).
- **I numeri ordinano prima delle lettere, e le lettere maiuscole ordinano prima delle minuscole** (numeri < maiuscole < minuscole).
- Per i tipi di riferimento, `null` è considerato più piccolo di qualsiasi valore non-null.

```
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()`

Requisiti: l'array deve essere ordinato; altrimenti il risultato è imprevedibile.

```
int[] a = {1,3,5,7};
int idx = Arrays.binarySearch(a, 5); // returns 2
```

Quando il valore non viene trovato, `binarySearch` restituisce `-(insertionPoint) - 1`:

```
int pos = Arrays.binarySearch(a, 4); // returns -3
// Il punto d'inserimento sarebbe all'indice 2 → -(2) - 1 = -3
```

10.6.5 `Arrays.compare()`

La classe `Arrays` offre un `equals()` sovraccarico che verifica se due array contengono gli stessi elementi (e hanno la stessa lunghezza):

```
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
```

Fornisce anche un metodo `compare()` con queste regole:

- Se il risultato `n < 0` → il primo array è considerato "più piccolo" del secondo.
- Se il risultato `n > 0` → il primo array è considerato "più grande" del secondo.
- Se il risultato `n == 0` → gli array sono uguali.
- Esempi:

```
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 Enhanced for-loop con array

```
for (int value : new int[]{1,2,3}) {
    System.out.println(value);
}
```

Regole - Il lato destro deve essere un array o un `Iterable`. - Il tipo della variabile di ciclo deve essere compatibile con il tipo degli elementi (**qui non c'è widening di primitivi**).

Errore comune:

```
// for (long v : new int[]{1,2}) {} // ❌ not allowed: int elements cannot be assigned to long
```

10.8 Errori comuni

- **Accesso fuori dai limiti** → lancia `ArrayIndexOutOfBoundsException`.
- **Uso errato dell'inizializzatore breve**

```
// int[] x;
// x = {1,2}; // ❌ does not compile
```

- **Confondere `.length` e `.length()`**
- **Dimenticare che gli array sono oggetti** (vivono nell'heap e sono referenziati).
- **Miscelare array di primitivi e array di wrapper**

```
// int[] p = new Integer[3]; // ❌ incompatible
```

- **Usare `binarySearch` su array non ordinati** → risultati imprevedibili.
- **Eccezioni runtime dovute a array covarianti** (`ArrayStoreException`).

10.9 Riepilogo

Gli array in Java sono:

- Oggetti (anche se contengono primitivi).
- Collezioni indicizzate a dimensione fissa.
- Sempre inizializzati con valori predefiniti.

- Type-safe, ma soggetti alle regole di covarianza (che possono causare eccezioni a runtime se usate in modo improprio).

◀ 9. Stringhe in Java | ▲ Index | 11. Matematica in Java ▶

11. Matematica in Java

Indice

- 11.1 API Math
  - 11.1.1 Massimo e minimo tra due valori
  - 11.1.2 Math.round
  - 11.1.3 Math.ceil (Ceiling)
  - 11.1.4 Math.floor (Floor)
  - 11.1.5 Math.pow
  - 11.1.6 Math.random
  - 11.1.7 Math.abs
  - 11.1.8 Math.sqrt
  - 11.1.9 Tabella riassuntiva
- 11.2 BigInteger e BigDecimal
  - 11.2.1 Perché double e float non sono sufficienti
  - 11.2.2 BigInteger — Interi a precisione arbitraria
  - 11.2.3 Creare BigInteger
  - 11.2.4 Operazioni (niente operatori)

11.1 API Math

La classe `java.lang.Math` fornisce un insieme di metodi statici utili per operazioni numeriche.

Questi metodi funzionano con i tipi numerici primitivi.

Di seguito una sintesi di quelli usati più frequentemente, insieme alle loro forme sovraccaricate.

11.1.1 Massimo e minimo tra due valori

`Math.max()` e `Math.min()` confrontano i due valori forniti e restituiscono il massimo o il minimo tra di essi.

Esistono quattro versioni sovraccaricate per ciascun metodo:

```
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);
```

- Esempio:

```
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()` restituisce l'intero più vicino al suo argomento, seguendo le regole standard di arrotondamento: i valori con parte frazionaria 0.5 e superiore vengono arrotondati verso l'alto; sotto 0.5 vengono arrotondati verso il basso (verso l'intero più vicino).

Overload

- `long round(double value)`
- `int round(float value)`
- Esempi:

```
Math.round(3.2);    // 3   (returns long)
Math.round(3.6);    // 4
Math.round(-3.5f);  // -3  (float version returns int)
```

> **Note**
>
> - La versione `float` restituisce un `int`.
> - La versione `double` restituisce un `long`.

11.1.3 `Math.ceil()` (Ceiling)

`ceil()` restituisce il più piccolo valore `double` che è maggiore o uguale all'argomento.

Overload

- `double ceil(double value)`
- Esempi:

```
Math.ceil(3.1);   // 4.0
Math.ceil(-3.1);  // -3.0
```

11.1.4 `Math.floor()` (Floor)

`floor()` restituisce il più grande valore `double` che è minore o uguale all'argomento.

Overload

- `double floor(double value)`
- Esempi:

```
Math.floor(3.9);   // 3.0
Math.floor(-3.1);  // -4.0
```

11.1.5 `Math.pow()`

`pow()` eleva un valore a una potenza.

Overload

- `double pow(double base, double exponent)`
- Esempi:

```
Math.pow(2, 3);      // 8.0
Math.pow(9, 0.5);    // 3.0  (radice quadrata)
Math.pow(10, -1);    // 0.1
```

11.1.6 `Math.random()`

`random()` restituisce un `double` randomico nell'intervallo `[0.0, 1.0)` (0.0 incluso, 1.0 escluso).

Overload

- `double random()`
- Esempi:

```
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()` restituisce il valore assoluto (distanza da zero).

Overload

- `int abs(int value)`
- `long abs(long value)`
- `float abs(float value)`
- `double abs(double value)`

11.1.8 `Math.sqrt()`

`sqrt()` calcola la radice quadrata e restituisce un `double`.

```
Math.sqrt(9);     // 3.0
Math.sqrt(-1);    // NaN (not a number)
```

11.1.9 Tabella riassuntiva

Metodo	Restituisce	Overload	Note
`round()`	int o long	float, double	Intero più vicino
`ceil()`	double	double	Valore più piccolo >= argomento
`floor()`	double	double	Valore più grande <= argomento
`pow()`	double	double, double	Esponenziazione
`random()`	double	none	0.0 <= r < 1.0
`min()/max()`	stesso tipo	int, long, float, double	Confronta due valori
`abs()`	stesso tipo	int, long, float, double	Valore assoluto
`sqrt()`	double	double	Radice quadrata

11.2 BigInteger e BigDecimal

Le classi `BigInteger` e `BigDecimal` (in `java.math`) forniscono tipi numerici a precisione arbitraria.

Si usano quando:

- I tipi primitivi (`int`, `long`, `double`, ecc.) non hanno abbastanza range.
- Gli errori di arrotondamento in virgola mobile di `float` / `double` non sono accettabili (ad esempio, nei calcoli finanziari).

Entrambi sono **immutabili**: ogni operazione restituisce una nuova istanza.

11.2.1 Perché `double` e `float` non sono sufficienti

I tipi in virgola mobile (`float`, `double`) usano una rappresentazione binaria. Molte frazioni decimali non possono essere rappresentate esattamente (come 0.1 o 0.2), quindi si ottengono errori di arrotondamento:

```
System.out.println(0.1 + 0.2); // 0.30000000000000004
```

Per attività come i calcoli finanziari, questo è inaccettabile.

`BigDecimal` risolve il problema rappresentando i numeri usando un modello decimale con una scala configurabile (numero di cifre dopo il separatore decimale).

11.2.2 BigInteger — Interi a precisione arbitraria

`BigInteger` rappresenta valori interi di dimensione praticamente qualsiasi, limitata solo dalla memoria disponibile.

11.2.3 Creare BigInteger

Modi comuni:

Da un long

```
static BigInteger valueOf(long val);
```

Da una String

```
BigInteger(String val);        // decimal by default
BigInteger(String val, int radix);
```

Valore randomico

```
BigInteger(int numBits, Random rnd);
```

- Esempi:

```
import java.math.BigInteger;
import java.math.BigDecimal;
import java.util.Random;

BigInteger a = BigInteger.valueOf(10L);

// Si puo passare un long a entrambi, ma un double solo a 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 Operazioni (niente operatori!)

Non puoi usare gli operatori aritmetici standard (`+`, `-`, `*`, `/`, `%`) con `BigInteger` o `BigDecimal`.

Devi invece chiamare metodi (tutti i quali restituiscono nuove istanze). Ecco alcuni di quelli comuni per `BigInteger`:

- `add(BigInteger val)`
- `subtract(BigInteger val)`
- `multiply(BigInteger val)`
- `divide(BigInteger val)` – divisione intera
- `remainder(BigInteger val)`
- `pow(int exponent)`
- `negate()`
- `abs()`
- `gcd(BigInteger val)`
- `compareTo(BigInteger val)` – ordinamento
- Esempio:

```java
BigInteger x = new BigInteger("100000000000000000000");
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");
}
```

---

12. Data e ora in Java

Indice

- 12.1 Data e ora
    - 12.1.1 Creare date e ore specifiche
    - 12.1.2 Aritmetica su data e ora: metodi plus e minus
    - 12.1.3 Pattern comuni
    - 12.1.4 Aritmetica su LocalDate
    - 12.1.5 Aritmetica su LocalTime
    - 12.1.6 Aritmetica su LocalDateTime
    - 12.1.7 Aritmetica su ZonedDateTime
    - 12.1.8 Tabella riassuntiva
- 12.2 Metodi withXxx
- 12.3 Conversione e metodi at: collegare data, ora e zona
- 12.4 Period, Duration e Instant
- 12.5 Period — quantità “umane” di data
- 12.6 Duration — quantità “macchina” di tempo
- 12.7 Instant — punto sulla timeline UTC
- 12.8 Tabella riassuntiva: Period vs Duration vs Instant
- 12.9 TemporalUnit e TemporalAmount
    - 12.9.1 TemporalUnit
    - 12.9.2 enum ChronoUnit
    - 12.9.3 TemporalAmount
    - 12.9.4 Period come TemporalAmount
    - 12.9.5 Duration come TemporalAmount
    - 12.9.6 Usare TemporalAmount vs TemporalUnit
    - 12.9.7 Metodi between
    - 12.9.8 Problemi comuni
    - 12.9.9 Riepilogo

12.1 Data e ora

Java fornisce un’API moderna, coerente e immutabile per data/ora nel package `java.time.*`.

Questa API sostituisce le vecchie classi `java.util.Date` e `java.util.Calendar`.

A seconda del livello di dettaglio richiesto, Java offre quattro classi principali:

- `LocalDate` → rappresenta solo una data (anno–mese–giorno)
- `LocalTime` → rappresenta solo un orario (ora–minuto–secondo–nanosecondo)
- `LocalDateTime` → combina data + ora, ma senza fuso orario
- `ZonedDateTime` → data + ora + offset + fuso orario completi

> **Note**
>
> - Un **fuso orario** definisce regole come i cambi dell'ora legale (ad esempio, `Europe/Paris`).
> - Un **offset di zona** è uno spostamento fisso rispetto a UTC/GMT (ad esempio, `+01:00` , `-07:00`).
> - Per confrontare due istanti provenienti da fusi orari diversi, convertili in UTC (GMT) applicando l'offset.

Ottenere la data/ora corrente

Puoi recuperare i valori correnti del sistema usando i metodi statici `now()` :

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

- Esempio di output (il tuo sistema potrebbe differire):

```
2025-12-01
19:11:53.213856300
2025-12-01T19:11:53.213856300
2025-12-01T19:11:53.214856900+01:00[Europe/Paris]
```

- Esempio: conversione di `ZonedDateTime` in GMT (UTC)

```
// Conceptual examples (codice d'esempio, solo per illustrare l'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
```

Entrambi rappresentano lo stesso istante nel tempo, semplicemente espresso in fusi orari diversi.

12.1.1 Creare date e ore specifiche

Puoi costruire oggetti di data/ora precisi usando i metodi factory `of()`. Tutte le classi includono più versioni sovraccaricate di `of()` (qui sono elencate solo le più comuni).

LocalDate — forme sovraccaricate di `of()`

- `of(int year, int month, int dayOfMonth)`
- `of(int year, Month month, int dayOfMonth)`

LocalTime — forme sovraccaricate di `of()`

- `of(int hour, int minute)`
- `of(int hour, int minute, int second)`
- `of(int hour, int minute, int second, int nanoOfSecond)`

LocalDateTime — forme sovraccaricate di `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 — forme sovraccaricate di `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)`
- Esempi

```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 Aritmetica su data e ora: metodi `plus` e `minus`

Tutte le classi nel package `java.time` (come `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, ecc.) sono **immutabili**. Ciò significa che metodi come `plusXxx()` e `minusXxx()` non modificano mai l'oggetto originale — restituiscono invece una nuova istanza con il valore regolato.

12.1.3 Pattern comuni

La maggior parte delle classi di data/ora supporta tre tipi di metodi aritmetici:

- **Scorciatoie specifiche per tipo**
- `plusDays(long daysToAdd)`
- `plusHours(long hoursToAdd)`
- ecc.
- **Metodi generici basati su "amount"**
- `plus(TemporalAmount amount)` → ad esempio `Period`, `Duration`
- `minus(TemporalAmount amount)`
- **Metodi generici basati su "unit"**
- `plus(long amountToAdd, TemporalUnit unit)`
- `minus(long amountToSubtract, TemporalUnit unit)`

Questi consentono un'aritmetica su data/ora flessibile e leggibile.

12.1.4 Aritmetica su `LocalDate`

`LocalDate` rappresenta solo una data (niente ora, niente zona).

Principali metodi `plus` / `minus` (overload)

Metodo	Descrizione
`plusDays(long days)`	Aggiunge giorni
`plusWeeks(long weeks)`	Aggiunge settimane
`plusMonths(long months)`	Aggiunge mesi
`plusYears(long years)`	Aggiunge anni
`minusDays(long days)`	Sottrae giorni
`minusWeeks(long weeks)`	Sottrae settimane
`minusMonths(long months)`	Sottrae mesi
`minusYears(long years)`	Sottrae anni
`plus(TemporalAmount amount)`	Aggiunge un Period
`minus(TemporalAmount amount)`	Sottrae un Period
`plus(long amountToAdd, TemporalUnit unit)`	Aggiunge usando ChronoUnit (es., DAYS, MONTHS)
`minus(long amountToSubtract, TemporalUnit unit)`	Sottrae usando ChronoUnit

- Esempi:

```
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 Aritmetica su `LocalTime`

`LocalTime` rappresenta solo un orario (niente data, niente zona).

Principali metodi `plus` / `minus` (overload)

Metodo	Descrizione
`plusNanos(long nanos)`	Aggiunge nanosecondi
`plusSeconds(long seconds)`	Aggiunge secondi
`plusMinutes(long minutes)`	Aggiunge minuti
`plusHours(long hours)`	Aggiunge ore
`minusNanos(long nanos)`	Sottrae nanosecondi
`minusSeconds(long seconds)`	Sottrae secondi
`minusMinutes(long minutes)`	Sottrae minuti
`minusHours(long hours)`	Sottrae ore
`plus(TemporalAmount amount)`	Aggiunge una Duration
`minus(TemporalAmount amount)`	Sottrae una Duration
`plus(long amountToAdd, TemporalUnit unit)`	Aggiunge usando ChronoUnit
`minus(long amountToSubtract, TemporalUnit unit)`	Sottrae usando ChronoUnit

- Esempi

```
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

Quando l'aritmetica sull'ora supera la mezzanotte, con `LocalTime` la data viene ignorata. Per esempio, 23:30 + 2 ore = 01:30 (senza alcuna data coinvolta).

12.1.6 Aritmetica su `LocalDateTime`

`LocalDateTime` combina data + ora, ma ancora senza fuso orario. Supporta sia le scorciatoie legate alla data sia quelle legate all'ora.

Principali metodi `plus` / `minus` (overload)

Metodo	Descrizione
`plusYears(long years)` / `minusYears(long years)`	Regola gli anni
`plusMonths(long months)` / `minusMonths(long months)`	Regola i mesi
`plusWeeks(long weeks)` / `minusWeeks(long weeks)`	Regola le settimane
`plusDays(long days)` / `minusDays(long days)`	Regola i giorni
`plusHours(long hours)` / `minusHours(long hours)`	Regola le ore
`plusMinutes(long minutes)` / `minusMinutes(long minutes)`	Regola i minuti
`plusSeconds(long seconds)` / `minusSeconds(long seconds)`	Regola i secondi
`plusNanos(long nanos)` / `minusNanos(long nanos)`	Regola i nanosecondi
`plus(TemporalAmount amount)` / `minus(TemporalAmount amount)`	Aggiunge/sottrae Period o Duration
`plus(long amountToAdd, TemporalUnit unit)` / `minus(long amountToSubtract, TemporalUnit unit)`	Usando ChronoUnit

- Esempi

```
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 Aritmetica su `ZonedDateTime`

`ZonedDateTime` rappresenta data + ora + fuso orario + offset.

Supporta gli stessi metodi `plus` / `minus` di `LocalDateTime`, ma con attenzione aggiuntiva ai fusi orari e all'ora legale (DST).

Principali metodi `plus` / `minus` (overload)

Metodo	Descrizione
`plusYears(long years)` / `minusYears(long years)`	Regola gli anni
`plusMonths(long months)` / `minusMonths(long months)`	Regola i mesi
`plusWeeks(long weeks)` / `minusWeeks(long weeks)`	Regola le settimane
`plusDays(long days)` / `minusDays(long days)`	Regola i giorni
`plusHours(long hours)` / `minusHours(long hours)`	Regola le ore
`plusMinutes(long minutes)` / `minusMinutes(long minutes)`	Regola i minuti
`plusSeconds(long seconds)` / `minusSeconds(long seconds)`	Regola i secondi
`plusNanos(long nanos)` / `minusNanos(long nanos)`	Regola i nanosecondi
`plus(TemporalAmount amount)` / `minus(TemporalAmount amount)`	Period / Duration
`plus(long amountToAdd, TemporalUnit unit)` / `minus(long amountToSubtract, TemporalUnit unit)`	Usando ChronoUnit

- Esempi (con fusi orari e 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);
```

A seconda delle regole dell'ora legale per quella data:

- L'orario locale potrebbe saltare da 02:00 a 03:00 o simile.
- `ZonedDateTime` regola l'offset e l'orario locale secondo le regole della zona, ma rappresenta comunque l'istante corretto sulla timeline.

> **Important**
>
> Per `ZonedDateTime`, l'aritmetica è definita in termini di timeline locale e regole del fuso orario, il che può causare spostamenti di ore durante le transizioni DST.

12.1.8 Tabella riassuntiva

Classe	Metodi shortcut plus/minus	Metodi generici
LocalDate	plusDays, plusWeeks, plusMonths, plusYears (e minus)	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
LocalTime	plusNanos, plusSeconds, plusMinutes, plusHours (e minus)	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
LocalDateTime	Tutte le scorciatoie di LocalDate + LocalTime	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
ZonedDateTime	Come LocalDateTime, ma con consapevolezza della zona	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)

12.2 Metodi `withXxx(...)`

I metodi `with...` restituiscono una copia dell'oggetto con un campo modificato. Non mutano mai l'istanza originale.

Classe	Metodi with... comuni (non esaustivo)	Descrizione
`LocalDate`	withYear(int year)	Stessa data, ma con un anno diverso
LocalDate	LocalDate.withMonth(int month)	Stessa data, mese diverso (1–12)
LocalDate	LocalDate.withDayOfMonth(int dayOfMonth)	Stessa data, giorno del mese diverso
LocalDate	LocalDate.with(TemporalField field, long newValue)	Regolazione generica basata su campo
LocalDate	LocalDate.with(TemporalAdjuster adjuster)	Usa un adjuster (es. firstDayOfMonth())
`LocalTime`	withHour(int hour)	Stessa ora, ora (hour) diversa
LocalTime	LocalTime.withMinute(int minute)	Stessa ora, minuto diverso
LocalTime	LocalTime.withSecond(int second)	Stessa ora, secondo diverso
LocalTime	LocalTime.withNano(int nanoOfSecond)	Stessa ora, nanosecondo diverso
LocalTime	LocalTime.with(TemporalField field, long newValue)	Regolazione generica basata su campo
LocalTime	LocalTime.with(TemporalAdjuster adjuster)	Regola usando un temporal adjuster
`LocalDateTime`	withYear(int year), withMonth(int month), withDayOfMonth(int day)	Cambia solo la parte data
LocalDateTime	withHour(int hour), withMinute(int minute), withSecond(int second)	Cambia solo la parte ora
LocalDateTime	withNano(int nanoOfSecond)	Cambia il nanosecondo
LocalDateTime	with(TemporalField field, long newValue)	Regolazione generica basata su campo
LocalDateTime	with(TemporalAdjuster adjuster)	Regola usando un temporal adjuster
`ZonedDateTime`	tutti i withXxx(...) di LocalDateTime	Cambia i componenti locali di data/ora
ZonedDateTime	withZoneSameInstant(ZoneId zone)	Stesso istante, zona diversa (cambia l'ora locale)
ZonedDateTime	withZoneSameLocal(ZoneId zone)	Stessa data/ora locale, zona diversa (cambia l'istante)

12.3 Conversione e metodi `at...` (collegare data, ora e zona)

Questi metodi sono usati per combinare o convertire tra `LocalDate`, `LocalTime`, `LocalDateTime` e `ZonedDateTime`.

Da	Metodo	Risultato	Descrizione
`LocalDate`	atTime(LocalTime time)	LocalDateTime	Combina questa data con un'ora data
LocalDate	atTime(int hour, int minute)	LocalDateTime	Overload di convenienza con componenti numerici dell'ora
LocalDate	atTime(int hour, int minute, int second)	LocalDateTime	—
LocalDate	atTime(int hour, int minute, int second, int nano)	LocalDateTime	—
LocalDate	atStartOfDay()	LocalDateTime	Questa data all'ora 00:00
LocalDate	atStartOfDay(ZoneId zone)	ZonedDateTime	Questa data all'inizio del giorno in una zona specifica
`LocalTime`	atDate(LocalDate date)	LocalDateTime	Combina questa ora con una data data
`LocalDateTime`	atZone(ZoneId zone)	ZonedDateTime	Aggiunge un fuso orario a una data/ora locale
LocalDateTime	toLocalDate()	LocalDate	Estrae il componente data
LocalDateTime	toLocalTime()	LocalTime	Estrae il componente ora
`ZonedDateTime`	toLocalDate()	LocalDate	Rimuove zona/offset, mantiene la data locale
ZonedDateTime	toLocalTime()	LocalTime	Rimuove zona/offset, mantiene l'ora locale
ZonedDateTime	toLocalDateTime()	LocalDateTime	Rimuove zona/offset, mantiene la data/ora locale

---

12.4 Period, Duration e Instant

Il package `java.time` fornisce tre classi temporali essenziali che rappresentano quantità di tempo o punti sulla timeline:

- **Period** → quantità di data "umane" (anni, mesi, giorni)
- **Duration** → quantità di tempo "macchina" (secondi, nanosecondi)
- **Instant** → un punto sulla timeline UTC

12.5 `Period` — quantità "umane" di data

`Period` rappresenta una quantità di tempo basata su data, come "3 anni, 2 mesi e 5 giorni". Si usa con `LocalDate` e `LocalDateTime` (perché contengono parti di data).

Metodi di creazione

Metodo	Descrizione
Period.ofYears(int years)	Solo anni
Period.ofMonths(int months)	Solo mesi
Period.ofWeeks(int weeks)	Converte settimane in giorni
Period.ofDays(int days)	Solo giorni
Period.of(int years, int months, int days)	Periodo completo
Period.parse(CharSequence text)	Formato ISO-8601: “P1Y2M3D”, “P7D”, “P1W”, …

Proprietà chiave

- Non supporta ore, minuti, secondi, nanosecondi.
- Può essere negativo.
- Immutabile.
- Esempi

```
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”) is allowed and represents a period of 7 days (equivalent to “P7D”).

> **Tip**
>
> Period is calendar-based: adding a period of months/years respects month lengths and leap years.

12.6 `Duration` — quantità “macchina” di tempo

`Duration` rappresenta una quantità di tempo basata su secondi e nanosecondi. Si usa con `LocalTime`, `LocalDateTime`, `ZonedDateTime` e `Instant`.

Metodi di creazione

Metodo	Descrizione
Duration.ofDays(long days)	Converte i giorni in secondi
Duration.ofHours(long hours)	Converte le ore in secondi
Duration.ofMinutes(long minutes)	Converte i minuti in secondi
Duration.ofSeconds(long seconds)	Rappresentazione base in secondi
Duration.ofSeconds(long seconds, long nanoAdjustment)	Secondi più nanos aggiuntivi
Duration.ofMillis(long millis)	Converte i millisecondi in nanos
Duration.ofNanos(long nanos)	Solo nanosecondi
Duration.between(Temporal start, Temporal end)	Calcola la durata tra due istanti
Duration.parse(CharSequence text)	ISO: “PT20H”, “PT15M”, “PT10S”

Caratteristiche chiave

- Supporta ore fino ai nanosecondi, ma non anni/mesi/settimane direttamente.
- Ideale per calcoli temporali a livello “macchina”.
- Immutabile.
- Esempi

```
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) represents exactly 24 hours of machine time. In a zone with DST, 24 hours may not align with “the same local time tomorrow”.

12.7 `Instant` — punto sulla timeline UTC

`Instant` rappresenta un singolo momento nel tempo relativo a UTC, con precisione al nanosecondo.

Contiene:

- Secondi dall’epoca (1970-01-01T00:00Z).
- Un’aggiunta di nanosecondi.

Metodi di creazione

Metodo	Descrizione
Instant.now()	Momento corrente in UTC
Instant.ofEpochSecond(long seconds)	Da secondi dall’epoca
Instant.ofEpochSecond(long seconds, long nanos)	Da secondi più nanos
Instant.ofEpochMilli(long millis)	Da millisecondi dall’epoca
Instant.parse(CharSequence text)	ISO: “2024-01-01T10:15:30Z”

Conversioni

Azione	Metodo
Instant → ora con zona	instant.atZone(zoneId)
ZonedDateTime → Instant	zdt.toInstant()
LocalDateTime → Instant	Non consentito direttamente (serve una zona)

- Esempio

```
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 è sempre UTC, senza informazioni di fuso orario associate. Non può essere combinato con un Period; usa invece Duration.

12.8 Tabella riassuntiva (Period vs Duration vs Instant)

Concetto	Rappresenta	Utile per	Funziona con	Note
Period	Anni, mesi, giorni	Aritmetica di calendario	LocalDate, LocalDateTime	Unità basate su "umano"
Duration	Ore fino ai nanosecondi	Calcoli temporali precisi	LocalTime, LocalDateTime, ZonedDateTime, Instant	Basato su "macchina"
Instant	Punto esatto sulla timeline UTC	Rappresentazione timestamp	Convertibile da/a ZonedDateTime	Non combinabile con Period

Trappole comuni

- `Period.of(1, 0, 0)` non è la stessa cosa di `Duration.ofDays(365)` (anni bisestili!).
- `Duration.ofDays(1)` potrebbe non essere uguale a un "giorno di calendario" pieno in una zona con DST.
- `LocalDateTime` non può essere convertito in un `Instant` senza un fuso orario.
- `Period.parse("P1W")` è valido e produce un periodo di 7 giorni.

12.9 TemporalUnit e TemporalAmount

L'API `java.time` si basa su due interfacce chiave che definiscono come date, orari e durate vengono manipolati:

- `TemporalUnit` → rappresenta un'unità di tempo (ad esempio, DAYS, HOURS, MINUTES).
- `TemporalAmount` → rappresenta una quantità di tempo (ad esempio, `Period`, `Duration`).

Entrambe sono essenziali per capire come funzionano i metodi `plus`, `minus` e `with`.

12.9.1 `TemporalUnit`

`TemporalUnit` rappresenta una singola unità di misura di data/ora. L'implementazione principale usata in Java è:

12.9.2 enum `ChronoUnit`

Questo enum fornisce le unità standard usate nella cronologia ISO-8601:

Categoria	Unità
Unità di data	DAYS, WEEKS, MONTHS, YEARS, DECADES, CENTURIES, MILLENNIA, ERAS
Unità di tempo	NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF_DAYS
Speciale	FOREVER

Un `TemporalUnit` può essere usato direttamente con i metodi `plus()` e `minus()`.

- Esempi usando `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**
>
> Non puoi usare unità basate sul tempo con LocalDate, né unità basate sulla data con LocalTime.

- Esempi:

```
// ❌ UnsupportedTemporalTypeException
LocalDate d = LocalDate.now().plus(5, ChronoUnit.HOURS);

// ❌ UnsupportedTemporalTypeException
LocalTime t = LocalTime.now().plus(1, ChronoUnit.DAYS);
```

12.9.3 `TemporalAmount`

`TemporalAmount` rappresenta una quantità di tempo multi-unità (per esempio, “2 anni, 3 mesi”, o “90 minuti”). È implementato da:

- `Period` → anni, mesi, giorni (basato su data)
- `Duration` → secondi, nanosecondi (basato su tempo)

Entrambi possono essere passati agli oggetti di data/ora per regolarli usando `plus()` e `minus()`.

12.9.4 `Period` come `TemporalAmount`

`Period` rappresenta una quantità “umana”: anni, mesi, giorni.

- Esempi:

```
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
```

Note

- `Period` non può essere usato con `LocalTime` (nessun componente data).
- `Period.ofWeeks(n)` viene convertito internamente in giorni (n × 7).

12.9.5 `Duration` come `TemporalAmount`

`Duration` rappresenta tempo “macchina”: secondi + nanosecondi.

- Esempi:

```
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
```

Note

- `Duration` può essere usata con classi che hanno componenti di ora (`LocalTime` , `LocalDateTime` , `ZonedDateTime` , `Instant`).
- `Duration` non può essere applicata a `LocalDate` → lancerà `UnsupportedTemporalTypeException` .
- `Duration` interagisce con zone e transizioni DST quando applicata a `ZonedDateTime` .

12.9.6 Usare `TemporalAmount` vs `TemporalUnit`

Usare un `TemporalUnit` :

```
LocalDate d1 = LocalDate.now().plus(5, ChronoUnit.DAYS);
```

Usare un `TemporalAmount` :

```
Period p = Period.ofDays(5);
LocalDate d2 = LocalDate.now().plus(p);
```

Entrambi producono lo stesso risultato quando supportati.

Differenze

Aspetto	TemporalUnit	TemporalAmount
Rappresenta	Una singola unità (es., DAYS)	Una quantità strutturata (es. 2Y, 5M, 3D)
Esempi	ChronoUnit.DAYS	Period.of(2,5,3)
Supporta più campi	No	Sì
Utile per	Incrementi semplici	Incrementi complessi
Comune con	Tutte le classi data/ora	Limitato dal tipo

12.9.7 Metodi `between(...)`

Molte classi forniscono un metodo `between` da `ChronoUnit` , `Duration` o `Period` .

Usare `Duration.between` (per classi basate sul tempo)

```
Duration d = Duration.between(
    LocalTime.of(10, 0),
    LocalTime.of(13, 30)
);
// PT3H30M
```

Usare `Period.between` (solo per date)

```
Period p = Period.between(
    LocalDate.of(2025, 3, 1),
    LocalDate.of(2025, 5, 10)
);
// P2M9D
```

Usare `ChronoUnit between`

```
long days = ChronoUnit.DAYS.between(
    LocalDate.of(2025, 3, 1),
    LocalDate.of(2025, 3, 10)
);
// 9
```

> **Important**
>
> ChronoUnit.between(...) always returns a long, while Period.between returns a Period, and Duration.between returns a Duration.

12.9.8 Problemi comuni

- Applicare il `TemporalAmount` sbagliato:

```
// LocalTime.plus(Period.ofDays(1))    // ❌ compile-time error
// LocalDate.plus(Duration.ofHours(1)) // ❌ runtime error: UnsupportedTemporalTypeException
```

- Cambi DST con Duration: aggiungere 24 ore non è sempre “domani” in una zona con cambi DST.
- `Period.ofWeeks(1)` è esattamente 7 giorni; gli effetti DST compaiono quando applicato a tipi consapevoli della zona.
- `Instant.plus(Period)` → runtime `UnsupportedTemporalTypeException`; usa `Duration` se possibile.
- `Instant` non puo esser creata direttamente da `LocalDateTime`; devi applicare prima una time zone: `ldt.atZone(zone).toInstant()`.

12.9.9 Riepilogo

Feature	TemporalUnit	TemporalAmount	ChronoUnit	Period	Duration
Represents	A unit	An amount	enum of units	Y/M/D	S + nanos
Multi-field	No	Yes	No	Yes	No
Works with	plus/minus	plus/minus	date/time	LocalDate/LocalDateTime	Time/time-zone
Human-based	No	Yes	No	Yes	No
Machine-based	Yes	Yes	Yes	No	Yes

---

13. Formattazione e Localizzazione in Java

Indice

- 13.1 Formattazione delle Stringhe
    - 13.1.1 String.format e formatted
        - 13.1.1.1 Flag per numeri in virgola mobile
        - 13.1.1.2 Precisione n
        - 13.1.1.3 Larghezza m
        - 13.1.1.4 Flag di riempimento con zero 0
        - 13.1.1.5 Giustificazione a sinistra Flag -
        - 13.1.1.6 Segno esplicito Flag +
        - 13.1.1.7 Parentesi per i negativi Flag (
        - 13.1.1.8 Combinazione dei flag
        - 13.1.1.9 Effetti del Locale
        - 13.1.1.10 Errori comuni
    - 13.1.2 Valori di testo personalizzati ed escaping
- 13.2 Formattazione dei Numeri
    - 13.2.1 NumberFormat
    - 13.2.2 Localizzazione dei numeri
    - 13.2.3 DecimalFormat e NumberFormat
    - 13.2.4 Struttura del pattern DecimalFormat
    - 13.2.5 Il simbolo 0 (cifra obbligatoria)
    - 13.2.6 Il simbolo # (cifra opzionale)
    - 13.2.7 Combinare 0 e #
    - 13.2.8 Separatori decimali e di raggruppamento
    - 13.2.9 DecimalFormatSymbols: simboli di formattazione specifici del Locale
    - 13.2.10 Pattern speciali di DecimalFormat
    - 13.2.11 Regole ed errori comuni
- 13.3 Parsing dei Numeri
    - 13.3.1 Parsing con DecimalFormat
    - 13.3.2 CompactNumberFormat
- 13.4 Formattazione di Data e Ora
    - 13.4.1 DateTimeFormatter
    - 13.4.2 Simboli standard di data e ora
    - 13.4.3 datetime.format vs formatter.format
    - 13.4.4 Localizzazione delle date
- 13.5 Internazionalizzazione (i18n) e Localizzazione (l10n)
    - 13.5.1 Locales
    - 13.5.2 Categorie di Locale
    - 13.5.3 Esempio reale
- 13.6 Properties e Resource Bundles
    - 13.6.1 Regole di risoluzione dei Resource Bundle
- 13.7 Regole ed errori comuni

Questo capitolo fornisce un trattamento approfondito e pratico della formattazione in Java 21.

13.1 Formattazione delle Stringhe

13.1.1 String.format e formatted

`String.format()` crea stringhe formattate utilizzando segnaposto in stile printf.

È sensibile al locale e restituisce una nuova `String` immutabile.

```
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));
```

Output:

```
The User: Bob | Score: 42
```

Caratteristiche chiave:

- Utilizza specificatori di formato come `%s` (qualsiasi tipo, comunemente valori String), `%d` (valori interi), `%f` (valori in virgola mobile).
- Non modifica le stringhe esistenti.
- Lancia `IllegalFormatException` se gli argomenti non corrispondono al formato.
- È sensibile al locale quando viene fornito un `Locale`.

```
String price = String.format(Locale.GERMANY, "%.2f", 1234.5);
// Output (locale tedesco): 1234,50
```

13.1.1.1 Flag per numeri in virgola mobile

`%f` è usato per formattare numeri in virgola mobile (`float`, `double`, `BigDecimal`) usando la notazione decimale.

```
System.out.printf("%f", 12.345);
```

```
12.345000
```

- **Stampa sempre 6 cifre dopo il punto decimale** per impostazione predefinita.
- Utilizza l'arrotondamento (non il troncamento).
- È sensibile al locale per il separatore decimale.

13.1.1.2 Precisione (.n)

La precisione definisce il numero di cifre stampate **dopo** il punto decimale.

```
System.out.printf("%.2f", 12.345);
```

```
12.35
```

- `%.0f` non stampa cifre decimali.
- Viene applicato l'arrotondamento.
- La precisione è applicata prima del riempimento della larghezza.

13.1.1.3 Larghezza (m)

La larghezza definisce il numero minimo totale di caratteri nell'output.

```
System.out.printf("%8.2f", 12.34);
```

```
   12.34
```

- Per impostazione predefinita **riempie con spazi**.
- Se il valore è più lungo, la larghezza viene ignorata (non viene mai troncato).
- Il riempimento è applicato a sinistra per impostazione predefinita.

13.1.1.4 Flag di riempimento con zero `0`

Il flag `0` sostituisce il riempimento con spazi con zeri.

```
System.out.printf("%08.2f", 12.34);
```

```
00012.34
```

- Richiede una larghezza.
- Gli zeri sono inseriti dopo il segno.
- Ignorato se è presente la giustificazione a sinistra (flag `-`).

13.1.1.5 Giustificazione a sinistra Flag `-`

Il flag `-` allinea il valore a sinistra all'interno della larghezza.

```
System.out.printf("%-8.2f", 12.34);
```

```
12.34
```

- Il riempimento viene spostato a destra.
- Sovrascrive il riempimento con zero.

13.1.1.6 Segno esplicito Flag `+`

Il flag `+` forza la visualizzazione del segno per i numeri positivi.

```
System.out.printf("%+8.2f", 12.34);
```

```
  +12.34
```

- I numeri negativi mostrano già `-`.
- Sovrascrive il flag spazio (che stampa uno spazio iniziale per i valori positivi).

13.1.1.7 Parentesi per i negativi Flag `(`

Il flag `(` formatta i numeri negativi usando le parentesi.

```
System.out.printf("%(8.2f", -12.34);
```

```
 (12.34)
```

- Influenza solo i valori negativi.
- Raramente usato nella pratica.

13.1.1.8 Combinazione dei flag

```
System.out.printf("%+010.2f", 12.34);
```

```
+000012.34
```

Ordine di valutazione (semplificato):

- Viene applicata la precisione.
- Viene gestito il segno.
- Viene applicata la larghezza.
- Viene applicato il riempimento (spazi o zeri).

13.1.1.9 Effetti del Locale

```
System.out.printf(Locale.FRANCE, "%,.2f", 12345.67);
```

```
12 345,67
```

I separatori decimali e di raggruppamento dipendono dal `Locale` attivo.

13.1.1.10 Errori comuni

- `%f` usa 6 cifre decimali per impostazione predefinita se non viene specificata la precisione.
- La larghezza non tronca mai l'output, ma aggiunge solo riempimento se necessario.
- Il flag `0` è ignorato quando è presente `-`.
- `+` sovrascrive il flag spazio.
- Raggruppamento e separatori dipendono dal Locale.

13.1.2 Valori di testo personalizzati ed escaping

Alcuni caratteri hanno un significato speciale nelle stringhe di formato e devono essere sottoposti a escaping.

- `%%` → segno percentuale letterale.
- `\n`, `\t` → escape standard Java.

```
String msg = String.format("Completion: %d%%%nStatus: OK", 100);
System.out.println(msg);
```

Output:

```
Completion: 100%
Status: OK
```

> **Note**
>
> Un singolo % senza uno specificatore valido causa una IllegalFormatException a runtime.

13.2 Formattazione dei Numeri

13.2.1 NumberFormat

`NumberFormat` è una classe astratta utilizzata per formattare e fare il parsing dei numeri in modo sensibile al locale.

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(nf.format(1234567.89));
```

> **Important**
>
> - I metodi factory determinano lo stile di formattazione (generale, intero, valuta, percentuale, compatto, ...).
> - La formattazione dipende dal `Locale` fornito.
> - `NumberFormat` (e `DecimalFormat`) non sono thread-safe.

13.2.2 Localizzazione dei numeri

La localizzazione dei numeri influisce sui separatori decimali, sui separatori di raggruppamento e sui simboli di valuta.

```
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 e NumberFormat

`DecimalFormat` è una sottoclasse concreta di `NumberFormat` che fornisce un controllo fine sulla formattazione numerica tramite pattern.

`NumberFormat` definisce una formattazione sensibile al locale tramite metodi factory, mentre `DecimalFormat` consente un controllo esplicito basato su pattern.

```
NumberFormat nf = NumberFormat.getInstance(Locale.US);
DecimalFormat df = (DecimalFormat) nf;
```

Oppure direttamente:

```
DecimalFormat df = new DecimalFormat("#,##0.00");
```

> **Note**
>
> - `DecimalFormat` è mutabile (è possibile cambiare pattern, simboli, ecc.).
> - `DecimalFormat` non è thread-safe.
> - La formattazione è sensibile al locale tramite `DecimalFormatSymbols`.

13.2.4 Struttura del pattern DecimalFormat

Un pattern può contenere una sottostruttura positiva e una negativa opzionale, separate da `;`.

```
#,##0.00;(#,##0.00)
```

> **Note**
>
> - La prima parte → numeri positivi.
> - La seconda parte → numeri negativi.
> - Se la parte negativa è omessa, i numeri negativi usano automaticamente un `-` iniziale.

13.2.5 Il simbolo `0` (cifra obbligatoria)

Il simbolo `0` forza la visualizzazione di una cifra, riempiendo con zeri se necessario.

```
DecimalFormat df = new DecimalFormat("0000.00");
System.out.println(df.format(12.3));
```

```
0012.30
```

- Controlla il numero minimo di cifre.
- Riempie con zeri se il numero ha meno cifre.
- Utile per output a larghezza fissa o allineato.

13.2.6 Il simbolo `#` (cifra opzionale)

Il simbolo `#` visualizza una cifra solo se esiste.

```
DecimalFormat df = new DecimalFormat("####.##");
System.out.println(df.format(12.3));
```

```
12.3
```

- Sopprime gli zeri iniziali.
- Sopprime gli zeri finali non necessari.
- Adatto a una formattazione “user-friendly”.

13.2.7 Combinare `0` e `#`

I pattern combinano spesso entrambi i simboli per maggiore flessibilità.

```
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
```

Spiegazione del pattern:

```
#,##0 . ##
 ^  ^    ^
 |  |    |
 |  |    └─ cifre frazionarie opzionali (#)
 |  └───── cifra intera obbligatoria (0)
 └──────── pattern di raggruppamento (,)
```

- È garantita almeno una cifra intera (lo `0`).
- Le cifre sono raggruppate per migliaia usando il separatore di raggruppamento.
- Le cifre frazionarie sono opzionali (fino a due).

13.2.8 Separatori decimali e di raggruppamento

Nei pattern:

- `.` → separatore decimale.
- `,` → separatore di raggruppamento.

I simboli effettivamente utilizzati a runtime dipendono dal `Locale` (ad esempio, virgola vs punto).

13.2.9 DecimalFormatSymbols: simboli di formattazione specifici del Locale

```
DecimalFormatSymbols symbols =
        DecimalFormatSymbols.getInstance(Locale.FRANCE);

DecimalFormat df =
        new DecimalFormat("#,##0.00", symbols);

System.out.println(df.format(1234.5));
```

```
1 234,50
```

- Controlla i separatori decimali e di raggruppamento.
- Controlla il segno meno e il simbolo di valuta.
- Controlla le stringhe NaN e Infinity.

13.2.10 Pattern speciali di DecimalFormat

```
0.###E0   notazione scientifica
###%      percentuale
¤#,##0.00 valuta (¤ è il simbolo di valuta)
```

13.2.11 Regole ed errori comuni

- `DecimalFormat` è una sottoclasse di `NumberFormat`.
- `0` forza le cifre, `#` no.
- I pattern controllano la formattazione, non la modalità di arrotondamento (usare `setRoundingMode()`).
- Il raggruppamento funziona solo se il separatore (di solito `,`) è presente nel pattern.
- Il parsing può riuscire parzialmente senza errore se sono presenti caratteri finali dopo un numero valido.
- `DecimalFormat` è mutabile e non thread-safe.

13.3 Parsing dei Numeri

Il parsing converte testo localizzato in valori numerici. Per impostazione predefinita, il parsing è permissivo.

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRANCE);
Number n = nf.parse("12 345,67abc"); // estrae 12345.67
```

- Il parsing si ferma al primo carattere non valido.
- Il testo finale viene ignorato se non controllato esplicitamente.

13.3.1 Parsing con DecimalFormat

`DecimalFormat` può anche effettuare il parsing dei numeri. Il parsing è permissivo per impostazione predefinita.

```
DecimalFormat df = new DecimalFormat("#,##0.##");
Number n = df.parse("1,234.56abc");
```

- Il parsing si ferma al primo carattere non valido.
- Il testo finale viene ignorato se presente.

Per forzare un parsing rigoroso:

```
df.setParseStrict(true);
```

13.3.2 CompactNumberFormat

La formattazione compatta abbrevia i numeri grandi per la leggibilità umana.

- Supporta stili SHORT e LONG.
- Usa abbreviazioni dipendenti dal locale (ad esempio K, M, “million”).

```java
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 Formattazione di Data e Ora

13.4.1 DateTimeFormatter

Java 21 si basa su `java.time` e `DateTimeFormatter` per la formattazione moderna di data e ora.

```java
DateTimeFormatter f =
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
System.out.println(LocalDateTime.now().format(f));
```

Proprietà principali:

- Immutabile.
- Thread-safe.
- Sensibile al locale.

13.4.2 Simboli standard di data e ora

```
y   anno
M   numero del mese (o nome con più lettere)
d   giorno del mese
E   nome del giorno
H   ora (0-23)
h   ora (1-12)
m   minuto
s   secondo
a   indicatore AM/PM
z   fuso orario
```

13.4.3 datetime.format vs formatter.format

Entrambi i metodi sono funzionalmente identici:

```java
date.format(formatter);
formatter.format(date);
```

- `date.format(formatter)` → preferito per leggibilità (prima il dato, poi la formattazione).
- `formatter.format(date)` → utile in codice funzionale o con formatter riutilizzabili.

13.4.4 Localizzazione delle date

Gli stili localizzati adattano l’output delle date alle norme culturali.

```
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));
```

Output possibile:

```
mercoledì 17 dicembre 2025
17/12/25
```

13.5 Internazionalizzazione (i18n) e Localizzazione (l10n)

13.5.1 Locales

Un `Locale` definisce lingua, paese e una variante opzionale.

```
Locale l1 = Locale.US;
Locale l2 = Locale.of("fr", "FR");
Locale l3 = new Locale.Builder()
        .setLanguage("en")
        .setRegion("US")
        .build();
```

Formati di Locale:

- `en` (it, fr, ecc.): codice lingua minuscolo.
- `en_US` (fr_CA, it_IT, ecc.): codice lingua minuscolo + underscore + codice paese maiuscolo.

13.5.2 Categorie di Locale

Le categorie di Locale separano la formattazione dalla lingua dell'interfaccia utente.

`Locale.Category` consente a Java di usare Locale predefiniti diversi per scopi diversi.

Esistono due categorie:

Category	Usata per
FORMAT	Numeri, date, valuta, altra formattazione
DISPLAY	Testo leggibile (UI, nomi, messaggi)

13.5.3 Esempio reale

Un utente francese che vive in Germania potrebbe volere:

- Numeri e date → formato tedesco.
- Lingua dell'interfaccia → francese.

Prima di Java 7, questo non era possibile.

```
Locale.setDefault(Locale.Category.FORMAT, Locale.GERMANY);
Locale.setDefault(Locale.Category.DISPLAY, Locale.FRANCE);
```

Effetti di esempio:

Aspetto	Risultato (esempio)
Numeri	1.234,56
Date	31.12.2025
Valuta	€
Testo UI	Francese
Nomi dei mesi	décembre
Nomi dei paesi	Allemagne

13.6 Properties e Resource Bundles

I resource bundle esternalizzano il testo e consentono la localizzazione senza modifiche al codice.

```
ResourceBundle rb =
        ResourceBundle.getBundle("messages", Locale.GERMAN);

String msg = rb.getString("welcome");
```

13.6.1 Regole di risoluzione dei Resource Bundle

Java cerca i bundle seguendo un ordine di fallback rigoroso. Ad esempio, con nome base `messages` e locale `de_DE`:

- messages_de_DE.properties
- messages_de.properties
- messages.properties

Se nessuno viene trovato → `MissingResourceException`.

> **Note**
>
> I file `.properties` tradizionali sono specificati come ISO-8859-1; i caratteri non ASCII devono essere codificati come escape Unicode (ad esempio `\u00E9` per é), a meno di usare meccanismi di caricamento alternativi.

13.7 Regole ed errori comuni

- `DateTimeFormatter` è immutabile e thread-safe.
- `NumberFormat` / `DecimalFormat` sono mutabili e non thread-safe.
- Cambiare il `Locale` influisce su come i valori sono formattati e interpretati, non sui valori numerici o temporali sottostanti.
- Il parsing con `NumberFormat` o `DecimalFormat` può riuscire parzialmente senza eccezioni se dopo un numero valido è presente testo aggiuntivo.
- `java.time` sostituisce la maggior parte degli usi delle vecchie API `java.util.Date` / `Calendar` nel codice moderno e nell'esame.

Object-Oriented Fundamentals

14. Metodi, Attributi e Variabili

Indice

- 14.1 Metodi
  - 14.1.1 Componenti obbligatori di un metodo
    - 14.1.1.1 Modificatori di accesso
    - 14.1.1.2 Tipo di ritorno
    - 14.1.1.3 Nome del metodo
    - 14.1.1.4 Firma del metodo
    - 14.1.1.5 Corpo del metodo
  - 14.1.2 Modificatori opzionali
  - 14.1.3 Dichiarare i metodi
- 14.2 Java è un linguaggio Pass-by-Value
- 14.3 Overloading dei metodi
  - 14.3.1 Chiamare metodi overloaded
    - 14.3.1.1 Vince la corrispondenza esatta
    - 14.3.1.2 Se non esiste una corrispondenza esatta Java sceglie il tipo compatibile più specifico
    - 14.3.1.3 L'allargamento dei primitivi batte il boxing
    - 14.3.1.4 Il boxing batte i varargs
    - 14.3.1.5 Per i riferimenti Java sceglie il tipo di riferimento più specifico
    - 14.3.1.6 Quando non esiste un più specifico non ambiguo la chiamata è un errore di compilazione
    - 14.3.1.7 Overload misti primitivi + wrapper
    - 14.3.1.8 Quando i primitivi si mescolano con i tipi di riferimento
    - 14.3.1.9 Quando vince Object
    - 14.3.1.10 Tabella riassuntiva risoluzione overload
- 14.4 Variabili locali e di istanza
  - 14.4.1 Variabili di istanza
  - 14.4.2 Variabili locali
    - 14.4.2.1 Variabili locali effettivamente final
    - 14.4.2.2 Parametri come effettivamente final
- 14.5 Varargs liste di argomenti a lunghezza variabile
- 14.6 Metodi statici variabili statiche e inizializzatori statici
  - 14.6.1 Variabili statiche variabili di classe
  - 14.6.2 Metodi statici
  - 14.6.3 Blocchi di inizializzazione statica
  - 14.6.4 Ordine di inizializzazione statico vs istanza
  - 14.6.5 Accesso ai membri statici
    - 14.6.5.1 Utilizzare nome della classe
    - 14.6.5.2 Utilizzare riferimento di istanza
  - 14.6.6 Static ed ereditarietà
  - 14.6.7 Errori comuni

Questo capitolo introduce concetti fondamentali di Programmazione Orientata agli Oggetti (OOP) in Java, iniziando con **metodi**, **passaggio dei parametri**, **overloading**, **variabili locali vs. di istanza** e

varargs.

14.1 Metodi

I `metodi` rappresentano le **operazioni/comportamenti** che possono essere eseguiti da un particolare tipo di dato (una **classe**).

Descrivono *cosa può fare l'oggetto* e come interagisce con il suo stato interno e con il mondo esterno.

Una `dichiarazione di metodo` è composta da componenti **obbligatori** e **opzionali**.

14.1.1 Componenti obbligatori di un metodo

14.1.1.1 Modificatori di accesso

I `Modificatori di accesso` controllano la *visibilità*, non il comportamento.

(Fare riferimento al Paragrafo "**Access Modifiers**" nel Capitolo: Mattoni di base del linguaggio Java)

14.1.1.2 Tipo di ritorno

Appare **immediatamente prima** del nome del metodo.

- Se il metodo restituisce un valore → il tipo di ritorno specifica il tipo del valore.
- Se il metodo *non* restituisce un valore → la keyword `void` **deve** essere usata.
- Un metodo con tipo di ritorno non-void **deve** contenere almeno una istruzione `return valore;`.
- Un metodo `void` può:
  - omettere una istruzione return
  - includere `return;` (senza **nessun** valore)

14.1.1.3 Nome del metodo

Segue le stesse regole degli identificatori (Fare riferimento al Capitolo: Regole di naming Java).

14.1.1.4 Firma del metodo

La **firma del metodo** in Java include:

- il *nome del metodo*
- la *lista dei tipi dei parametri* (tipi + ordine)

> **Note**
>
> I **nomi dei parametri NON fanno parte della firma**, contano solo `tipi` e `ordine`.

- Esempio di firme distinte:

```
void process(int x)
void process(int x, int y)
void process(int x, String y)
```

- Esempio di *stessa* firma (overloading illegale):

```
// ❌ stessa firma: differiscono solo i nomi dei parametri
void m(int a)
void m(int b)
```

14.1.1.5 Corpo del metodo

Un blocco `{ }` contenente **zero o più istruzioni**.

Se il metodo è `abstract`, il corpo deve essere omesso.

14.1.2 Modificatori opzionali

I modificatori opzionali dei metodi includono:

- `static`
- `abstract`
- `final`
- `default` (metodi di interfaccia)
- `synchronized`
- `native`
- `strictfp`

Regole:

- I modificatori opzionali possono apparire in **qualsiasi ordine**.
- Tutti i modificatori devono apparire **prima del tipo di ritorno**.
- Esempio:

```
public static final int compute() {
    return 10;
}
```

14.1.3 Dichiarare i metodi

```
public final synchronized String formatValue(int x, double y) throws IOException {
    return "Result: " + x + ", " + y;
}
```

Scomposizione:

Part	Significato
`public`	modificatore di accesso
`final`	non può essere sovrascritto
`synchronized`	modificatore di controllo dei thread
`String`	tipo di ritorno
`formatValue`	nome del metodo
`(int x, double y)`	lista dei parametri
`throws IOException`	lista delle eccezioni
`method body`	implementazione

14.2 Java è un linguaggio "Pass-by-Value"

Java usa **solo pass-by-value**, senza eccezioni.

Questo significa:

- Per i **tipi primitivi** → il metodo riceve una *copia del valore*.
- Per i **tipi riferimento** → il metodo riceve una *copia del riferimento*, il che significa:
-   - il riferimento stesso non può essere cambiato dal metodo
-   - l'*oggetto puntato* **può** essere modificato tramite quel riferimento
- Esempio:

```
void modify(int a, StringBuilder b) {
    a = 50;           // modifica la *copia* locale → nessun effetto all'esterno
    b.append("!");    // modifica l'*oggetto puntato* → visibile all'esterno
}
```

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

    int num1 = 11;
    methodTryModif(num1);
    System.out.println(num1);

}

public static void methodTryModif(int num1){
    num1 = 10;  // questa nuova assegnazione influisce solo sul parametro del metodo che, acci
}
```

14.3 Overloading dei metodi

L'overloading dei metodi significa **stesso nome del metodo**, **firma diversa**.

Due metodi sono considerati `overloaded` se differiscono per:

- numero di parametri
- tipi dei parametri
- ordine dei parametri

L'overloading **NON dipende da**:

- tipo di ritorno
- modificatore di accesso
- eccezioni
- Esempio:

```
void print(int x)
void print(double x)
void print(int x, int y)
```

Metodo overloaded illegale:

```
// ❌ Il tipo di ritorno non conta nell'overloading
int compute(int x)
double compute(int x)
```

14.3.1 Chiamare metodi overloaded

Quando sono disponibili più metodi overloaded, Java applica la **risoluzione dell'overload** per decidere quale metodo chiamare.

Il compilatore seleziona il metodo i cui tipi di parametro sono i **più specifici** e **compatibili** con gli argomenti forniti.

La risoluzione dell'overload avviene **a compile-time** (a differenza dell' `overriding`, che è basato sul **run-time**).

Java segue queste regole:

14.3.1.1 Vince la corrispondenza esatta

Se un argomento corrisponde esattamente a un parametro del metodo, quel metodo viene scelto.

```java
void call(int x)    { System.out.println("int"); }
void call(long x)   { System.out.println("long"); }

call(5); // stampa: int (corrispondenza esatta per int)
```

14.3.1.2 — Se non esiste una corrispondenza esatta, Java sceglie il tipo compatibile *più specifico*

Java preferisce:

- **widening** rispetto all'autoboxing
- **autoboxing** rispetto ai varargs
- **riferimento più ampio** solo se un tipo più specifico non è disponibile
- Esempio con primitivi numerici:

```java
void test(long x)   { System.out.println("long"); }
void test(float x)  { System.out.println("float"); }

test(5);  // letterale int: può essere allargato a long o float
          // ma long è più specifico di float per i tipi interi
          // Output: long
```

14.3.1.3 — L'allargamento dei primitivi batte il boxing

Se un argomento primitivo può essere sia allargato sia autoboxato, Java sceglie l'allargamento.

```java
void m(int x)       { System.out.println("int"); }
void m(Integer x)   { System.out.println("Integer"); }

byte b = 10;
m(b);               // byte → int (widening) vince
                    // Output: int
```

14.3.1.4 — Il boxing batte i varargs

```java
void show(Integer x)    { System.out.println("Integer"); }
void show(int... x)     { System.out.println("varargs"); }

show(5);                // int → Integer (boxing) preferito
                        // Output: Integer
```

14.3.1.5 — Per i riferimenti, Java sceglie il tipo di riferimento più specifico

```java
void ref(Object o)      { System.out.println("Object"); }
void ref(String s)      { System.out.println("String"); }

ref("abc");             // "abc" è una String → più specifica di Object
                        // Output: String
```

Più specifico significa *più in basso nella gerarchia di ereditarietà*.

14.3.1.6 — Quando non esiste un "più specifico" non ambiguo, la chiamata genera un errore di compilazione

Esempio con classi sorelle:

```java
void check(Number n)      { System.out.println("Number"); }
void check(String s)      { System.out.println("String"); }

check(null);    // Sia String che Number accettano null
                // String è più specifica perché è una classe concreta
                // Output: String
```

Ma se competono due classi non correlate:

```
void run(String s)   { }
void run(Integer i)  { }

run(null);  // ❌ Errore a compile-time: chiamata di metodo ambigua
```

14.3.1.7 — Overload misti primitivi + wrapper

Java valuta `widening`, `boxing` e `varargs` in questo ordine:

`widening → boxing → varargs`

- Esempio:

```
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  (widening)
           // boxing e varargs ignorati
           // Output: long
```

14.3.1.8 — Quando i primitivi si mescolano con i tipi reference

```
void fun(Object o)     { System.out.println("Object"); }
void fun(int x)        { System.out.println("int"); }

fun(10);                 // vince la corrispondenza primitiva esatta
                         // Output: int

Integer i = 10;
fun(i);                  // riferimento accettato → Object
                         // Output: Object
```

14.3.1.9 — Quando vince Object

```
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 Tabella riassuntiva (Risoluzione dell'overload)

Situation	**Rule**
Exact match	Sempre scelto
Primitive widening vs boxing	Vince il widening
Boxing vs varargs	Vince il boxing
Most specific reference type	Vince
Unrelated reference types	Ambiguo → errore di compilazione
Mixed primitive + wrapper	Widening → boxing → varargs

14.4 Variabili locali e di istanza

14.4.1 Variabili di istanza

Le variabili di istanza sono:

- dichiarate come membri di una classe
- create quando un oggetto è istanziato
- accessibili da tutti i metodi dell'istanza

Modificatori possibili per le variabili di istanza:

- modificatori di accesso (`public` , `protected` , `private` , `package-private`)
- `final`
- `volatile`
- `transient`
- Esempio:

```
public class Person {
    private String name;          // variabile di istanza
    protected final int age = 0; // final significa che non può essere riassegnata
}
```

14.4.2 Variabili locali

Le variabili locali:

- sono dichiarate *all'interno* di un metodo, costruttore o blocco
- non hanno **valori di default** → devono essere inizializzate esplicitamente prima dell'uso
- unico modificatore consentito: **final**
- Esempio:

```
void calculate() {
    int x;         // dichiarata
    x = 10;        // deve essere inizializzata prima dell'uso

    final int y = 5;  // legale
}
```

Due casi speciali:

14.4.2.1 Variabili locali effettivamente final

Una variabile locale è *effettivamente final* se viene **assegnata una sola volta**, anche senza `final` .

Le variabili effettivamente final possono essere usate in:

- espressioni lambda
- classi locali/anonime

14.4.2.2 Parametri come effettivamente final

I parametri di metodo si comportano come variabili locali e seguono le stesse regole.

14.5 Varargs (Liste di argomenti a lunghezza variabile)

I varargs permettono a un metodo di accettare **zero o più** parametri dello stesso tipo.

Sintassi:

```
void printNames(String... names)
```

Regole:

- Un metodo può avere **un solo** parametro varargs.
- Deve essere l'**ultimo** parametro nella lista.
- I varargs sono trattati come un **array** all'interno del metodo.
- Esempio:

```java
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**
>
> Varargs e array partecipano all'overloading dei metodi. La risoluzione dell'overload può diventare ambigua.

14.6 Metodi statici, variabili statiche e inizializzatori statici

In Java, la keyword `static` marca elementi che **appartengono alla classe stessa**, non alle singole istanze. Questo significa:

- Sono **caricati una sola volta** in memoria quando la classe è caricata per la prima volta dalla JVM.
- Sono condivisi tra **tutte le istanze**.
- Vi si puo accedere **senza creare un oggetto** della classe.

I membri statici sono memorizzati nella **method area** della JVM (memoria a livello di classe), mentre i membri di istanza vivono nello **heap**.

14.6.1 Variabili statiche (Variabili di classe)

Una **variabile statica** è una variabile definita a livello di classe e condivisa da tutte le istanze.

Caratteristiche:

- Create quando la classe è caricata.
- Esistono **anche se nessuna istanza** della classe è creata.
- Tutti gli oggetti vedono lo **stesso valore**.
- Possono essere marcate `final`, `volatile` o `transient`.
- Esempio:

```java
public class Counter {
    static int count = 0;     // condivisa da tutte le istanze
    int id;                   // variabile di istanza

    public Counter() {
        count++;
        id = count;           // ogni istanza ottiene un id unico
    }
}
```

14.6.2 Metodi statici

Un **metodo statico** appartiene alla classe, non a una istanza dell'oggetto.

Regole:

- Possono essere chiamati usando il nome della classe: `ClassName.method()`.
- Non possono accedere direttamente a variabili o metodi di istanza, ma solo tramite un'istanza della classe.
- Non possono usare `this` o `super`.
- Sono comunemente usati per:
-
  - metodi di utilità (es. `Math.sqrt()`)
-
  - factory methods

- - comportamenti globali che non dipendono dallo stato di istanza
- Esempio:

```
public class MathUtil {

    static int square(int x) {         // metodo statico
        return x * x;
    }

    void instanceMethod() {
        // System.out.println(count);   // OK: accesso a variabile statica
        // square(5);                   // OK: metodo statico accessibile
    }
}
```

Errori comuni:

```
// ❌ Errore di compilazione: metodo di istanza non accessibile direttamente in contesto stati
static void go() {
    run();         // run() è un metodo di istanza!
}

void run() { }
```

14.6.3 Blocchi di inizializzazione statica

I blocchi di inizializzazione statica permettono di eseguire codice **una sola volta**, quando la classe è caricata.

Sintassi:

```
static {
    // logica di inizializzazione
}
```

Utilizzo:

- inizializzazione di variabili statiche complesse
- esecuzione di setup a livello di classe
- esecuzione di codice che deve essere eseguito esattamente una volta
- Esempio:

```
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**
>
> I blocchi di inizializzazione statica vengono eseguiti **una sola volta**, nell'ordine in cui appaiono, prima di `main()` e prima che qualsiasi metodo statico sia chiamato.

14.6.4 Ordine di inizializzazione (Statico vs. Istanza)

(Fare riferimento al Capitolo: Class Loading, Inizializzazione, e Costruzione degli Oggetti)

14.6.5 Accesso ai membri statici

14.6.5.1 Utilizzare nome della classe

```
Math.sqrt(16);
MyClass.staticMethod();
```

14.6.5.2 Utilizzare riferimento di istanza

```
MyClass obj = new MyClass();
obj.staticMethod();
```

14.6.6 Static ed ereditarietà

I metodi statici:

- possono essere **nascosti**, non sovrascritti
- il binding è a **compile-time**, non a runtime
- sono accessi in base al **tipo del riferimento**, non al tipo dell'oggetto
- Esempio:

```
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();   // stampa "A" — binding statico!
```

> **Note**
>
> Regola chiave: i metodi statici usano il **tipo del riferimento**, non il tipo dell'oggetto.

14.6.7 Errori comuni

- Tentare di riferirsi a una variabile/metodo di istanza da un contesto statico.
- Supporre che i metodi statici siano sovrascritti → sono **nascosti**.
- Chiamare un metodo statico da un riferimento di istanza (legale ma confondente).
- Confondere l'ordine di inizializzazione degli elementi statici rispetto a quelli di istanza.
- Dimenticare che le variabili statiche sono condivise tra tutti gli oggetti.
- Non sapere che gli inizializzatori statici vengono eseguiti *una sola volta*, in ordine di dichiarazione.

◀ 13. Formattazione e Localizzazione in Java | ▲ Index | 15. Caricamento delle Classi, Inizializzazione e Costruzione degli Oggetti ▶

15. Caricamento delle Classi, Inizializzazione e Costruzione degli Oggetti

Indice

- 15.1 Aree di Memoria Java Rilevanti per l'Inizializzazione di Classi e Oggetti
- 15.2 Caricamento delle Classi con Ereditarietà
  - 15.2.1 Ordine di Caricamento delle Classi
  - 15.2.2 Cosa Succede Durante il Caricamento di una Classe
- 15.3 Creazione degli Oggetti con Ereditarietà
  - 15.3.1 Ordine Completo di Creazione delle Istanze
- 15.4 Esempio Completo: Inizializzazione Statica + di Istanza nell'Ereditarietà
- 15.5 Diagramma di Visualizzazione
- 15.6 Regole Chiave
- 15.7 Tabella Riassuntiva

In Java, comprendere **come vengono caricate le classi**, **come vengono inizializzati i membri statici e di istanza**, e **come vengono eseguiti i costruttori — specialmente con l'ereditarietà** — è fondamentale per padroneggiare il linguaggio.

Questo capitolo fornisce una spiegazione unificata e chiara su:

- Come una classe viene caricata in memoria
- Come vengono eseguite le variabili statiche e i blocchi statici
- Come vengono creati gli oggetti passo dopo passo
- Come vengono eseguiti i costruttori in una catena ereditaria
- Come le diverse aree di memoria (Heap, Stack, Method Area) partecipano al processo

15.1 Aree di Memoria Java rilevanti per l'Inizializzazione di Classi e Oggetti

Prima di comprendere l'ordine di inizializzazione, è utile ricordare le tre principali aree di memoria coinvolte:

- **Method Area (nota anche come Class Area)** — memorizza i metadati delle classi, le variabili statiche e i blocchi di inizializzazione statica.
- **Heap** — memorizza tutti gli oggetti e i campi di istanza.
- **Stack** — memorizza le chiamate ai metodi, le variabili locali e i riferimenti.

> **Note**
>
> I membri statici appartengono alla **classe** e vengono creati **una sola volta** nella Method Area.
>
> I membri di istanza appartengono a **ogni oggetto** e vivono nell'**Heap**.

15.2 Caricamento delle Classi (con Ereditarietà)

Quando un programma Java si avvia, la JVM carica le classi *su richiesta*.

Quando una classe viene referenziata per la prima volta (ad esempio tramite `new` o accedendo a un membro statico), **l'intera catena ereditaria deve essere caricata prima in memoria**.

15.2.1 Ordine di Caricamento delle Classi

Data una gerarchia di classi:

```
class A { ... }
class B extends A { ... }
class C extends B { ... }
```

Se il codice esegue:

```
public static void main(String[] args) {
    new C();
}
```

Allora il caricamento delle classi procede in questo ordine rigoroso:

- Carica la classe A
- Inizializza le variabili statiche di A (default → esplicite)
- Esegue i blocchi di inizializzazione statica di A (dall'alto verso il basso)
- Carica la classe B e ripete la stessa logica
- Carica la classe C e ripete la stessa logica

15.2.2 Cosa Succede Durante il Caricamento di una Classe

- **Passo 1: Le variabili statiche vengono allocate** (prima con valori di default).
- **Passo 2: Vengono eseguite le inizializzazioni statiche esplicite**.
- **Passo 3: Vengono eseguiti i blocchi di inizializzazione statica** nell'ordine in cui compaiono nel codice.

> **Note**
>
> Dopo questi passaggi, la classe è completamente pronta e può essere utilizzata (istanziata o referenziata).

> **Warning**
>
> L'accesso a un campo statico provoca l'inizializzazione esclusivamente della classe o dell'interfaccia che lo dichiara direttamente.
>
> Questo vale anche se il campo viene referenziato tramite il nome di una sottoclasse, di una sottointerfaccia o di una classe che implementa l'interfaccia.

15.3 Creazione degli Oggetti (con Ereditarietà)

Quando viene usata la parola chiave `new`, **la creazione dell'istanza segue una sequenza rigorosa e prevedibile** che coinvolge tutte le classi genitrici.

15.3.1 Ordine Completo di Creazione delle Istanze

- **1. Viene allocata memoria sull'Heap per il nuovo oggetto** (gli attributi ricevono valori di default).
- **2. La catena dei costruttori viene eseguita (non ancora i corpi) dal genitore al figlio** — si parte dalla cima della gerarchia e procede verso le subclass.
- **3. Le variabili di istanza ricevono le inizializzazioni esplicite**.
- **4. Vengono eseguiti i blocchi di inizializzazione di istanza**.
- **5. Viene eseguito il corpo del costruttore**: per ogni classe nella catena ereditaria, i passaggi 3–5 (inizializzazione dei campi, blocchi di istanza, corpo del costruttore) si applicano dal genitore al figlio.

15.4 Esempio Completo: Inizializzazione Statica + di Istanza nell'Ereditarietà

Consideriamo la seguente gerarchia a tre livelli:

```
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();
    }
}
```

Output

```
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 Diagramma di Visualizzazione

```
            CARICAMENTO DELLE CLASSI (dall'alto verso il basso)

                A  --->  B  --->  C
                |         |         |
      variabili statiche + blocchi statici eseguiti in ordine

-------------------------------------------------------

            CREAZIONE DELL'OGGETTO (dal genitore al figlio)

 new C()
    |
    +--> allocazione memoria per C (valori di default)
    +--> chiamata al costruttore B()
            |
            +--> chiamata al costruttore A()
                    |
                    +--> inizializza variabili di istanza di A
                    +--> esegue blocchi di istanza di A
                    +--> esegue costruttore A
            +--> inizializza variabili di istanza di B
            +--> esegue blocchi di istanza di B
            +--> esegue costruttore B
    +--> inizializza variabili di istanza di C
    +--> esegue blocchi di istanza di C
    +--> esegue costruttore C
```

15.6 Regole Chiave

- L'inizializzazione statica avviene **una sola volta** per classe.
- Gli inizializzatori statici vengono eseguiti in ordine genitore → figlio.
- L'inizializzazione di istanza avviene ogni volta che viene creato un oggetto.
- Per ogni classe nella catena ereditaria, i campi di istanza e i blocchi di istanza vengono eseguiti prima del corpo del costruttore di quella classe.
- Nel complesso, sia l'inizializzazione dei campi/blocchi di istanza sia i costruttori vengono eseguiti dal genitore al figlio.
- I costruttori chiamano sempre il costruttore del genitore (esplicitamente o implicitamente).

15.7 Tabella Riassuntiva

STATIC (Livello Classe)	INSTANCE (Livello Oggetto)
Una sola volta	Avviene a ogni `new`
Eseguito genitore → figlio	Inizializzazione di istanza e costruttori genitore → figlio
variabili statiche (default → esplicite)	variabili di istanza (default → esplicite)
blocchi statici	blocchi di istanza + costruttore

◀ 14. Metodi, Attributi e Variabili | ▲ Index | 16. Ereditarietà in Java ▶

16. Ereditarietà in Java

Indice

- 16.1 Definizione Generale di Ereditarietà
- 16.2 Ereditarietà Singola e java.lang.Object
- 16.3 Ereditarietà Transitiva
- 16.4 Cosa Viene Ereditato, Breve Promemoria
- 16.5 Modificatori di Classe che Influenzano l'Ereditarietà
- 16.6 Riferimenti this e super
  - 16.6.1 Il Riferimento this
  - 16.6.2 Il Riferimento super
- 16.7 Dichiarare Costruttori in una catena ereditaria
- 16.8 Costruttore no-arg di Default
- 16.9 Usare this e Constructor Overloading
- 16.10 Chiamare il Costruttore del Parent usando super
- 16.11 Suggerimenti e Trappole sul Costruttore di Default
- 16.12 super si Riferisce Sempre al Parent più diretto
- 16.13 Ereditare Membri
  - 16.13.1 Method Overriding
    - 16.13.1.1 Definizione e Ruolo nell'Ereditarietà
    - 16.13.1.2 Usare super per chiamare l'Implementazione del Parent
    - 16.13.1.3 Regole di Overriding Instance Methods
    - 16.13.1.4 Nascondere Static Methods Method Hiding
  - 16.13.2 Abstract Classes
    - 16.13.2.1 Definizione e Scopo
    - 16.13.2.2 Regole per le Abstract Classes
  - 16.13.3 Creare Oggetti Immutabili
    - 16.13.3.1 Cos'è un Oggetto Immutable
    - 16.13.3.2 Linee Guida per Progettare Classi Immutable

L' `Inheritance` (Ereditarietà) è uno dei pilastri fondamentali dell'Object-Oriented Programming.

Essa permette a una classe `figlia` (child), la **subclass**, di acquisire lo stato e il comportamento di un'altra classe `genitrice` (parent), la **superclass**, creando relazioni gerarchiche che promuovono riuso del codice, specializzazione e polimorfismo.

16.1 Definizione Generale di Ereditarietà

L'ereditarietà consente a una classe di estenderne un'altra, ottenendone automaticamente i suoi `attributi` e i suoi `metodi` accessibili.

La classe che estende può aggiungere nuove funzionalità o ridefinire (fare `override`) i comportamenti esistenti, creando versioni più specializzate della propria classe parent.

> **Note**
>
> L'Ereditarietà esprime una relazione *"is-a"* (è-un): un Cane **is a** (è-un) Animale.

16.2 Ereditarietà Singola e java.lang.Object

Java supporta la **single inheritance**, il che significa che ogni classe può estendere **una sola** superclasse diretta.

Tutte le classi ereditano in ultima analisi da `java.lang.Object`, che si trova al vertice della gerarchia.

Questo garantisce che tutti gli oggetti Java condividano un comportamento minimo comune (ad esempio i metodi `toString()`, `equals()`, `hashCode()`).

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

// All classes implicitly extend Object
System.out.println(new Dog() instanceof Object); // true
```

16.3 Ereditarietà Transitiva

L' `Inheritance` è **transitiva**.

Se la classe `C` estende `B` e `B` estende `A`, allora `C` eredita effettivamente i membri accessibili sia da `B` *sia* da `A`.

```java
class A { }
class B extends A { }
class C extends B { } // C inherits from both A and B
```

16.4 Cosa Viene Ereditato, Breve Promemoria

Una subclass eredita tutti i membri **accessibili** della classe genitrice.

Tuttavia, nello specifico, questo dipende dagli `access modifiers`.

- **public** → sempre ereditato
- **protected** → ereditato se accessibile tramite regole di package o subclass
- **default (package-private)** → ereditato solo nello stesso package
- **private** → **NON** ereditato

> **Note**
>
> (Fare riferimento al Paragrafo “**Access Modifiers**” nel Capitolo: Mattoni di base del linguaggio Java)

16.5 Modificatori di Classe che influenzano l’Ereditarietà

Alcuni modificatori a livello di classe determinano se una classe possa essere estesa.

Modifier	Meaning	Effect on Inheritance
`final`	La classe non può essere estesa	Inheritance STOP
`abstract`	La classe non può essere istanziata	Deve essere estesa
`sealed`	Permette solo un elenco fisso di subclass	Restringe l'inheritance
`non-sealed`	Subclass di una sealed class che riapre l'inheritance	Inheritance permesso
`static`	Si applica solo alle nested classes	Si comporta come una top-level class all'interno della sua classe contenitore

> **Note**
>
> Una classe `static` in Java può esistere solo come **static nested class**.

16.6 Riferimenti `this` e `super`

16.6.1 Il Riferimento `this`

Il riferimento `this` si riferisce all'istanza corrente dell'oggetto e permette di disambiguare l'accesso ai membri correnti ed ereditati.

Java utilizza una regola di **granular scope**:

- Se una variabile di metodo/locale ha lo stesso nome di un `instance field`, quella locale "oscura" l'attributo di istanza.
- È necessario usare `this.fieldName` per accedere quindi all'attributo di istanza.

```
public class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Se i nomi differiscono, `this` è opzionale.

```
public class Person {
    String name;

    public Person(String n) {
        name = n;
    }
}
```

> **Warning**
>
> `this` NON può essere usato all'interno di metodi statici perché, in quel contesto, non esiste alcuna istanza.

16.6.2 Il Riferimento `super`

Il riferimento `super` dà accesso ai membri della classe genitrice (parent) diretta.

Utile quando:

- Il parent (genitore) e il child (figlio) definiscono un attributo/metodo con lo stesso nome; vedi sezione: [Ereditare Membri]

- Parent e child definiscono un attributo con lo stesso nome → `variable hiding` (due copie)
- Parent e child definiscono un metodo con la stessa signature → `method overriding`
- Si vuole chiamare esplicitamente l'implementazione ereditata

```
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` NON può essere usato dentro contesti static.

16.7 Dichiarare Costruttori in una catena ereditaria

Un `costruttore` inizializza un oggetto appena creato.

I costruttori non vengono **mai ereditati**, ma ogni costruttore di subclass deve assicurare che la classe parent sia inizializzata.

I `costruttori` sono metodi speciali con un nome che corrisponde al nome della classe e che non dichiarano alcun return type.

Una classe può definire più costruttori (constructor overloading), ciascuno con una `signature` unica.

Si può dichiarare esplicitamente un `no-arg constructor` o un qualsiasi costruttore specifico oppure, se non lo si fa, Java creerà implicitamente un `default no-arg constructor`.

Se si dichiara esplicitamente un costruttore, il compilatore Java non includerà alcun `default no-arg constructor`: questa regola si applica indipendentemente a ogni classe nella gerarchia.

Una classe parent continua ad avere il proprio costruttore di default a meno che non ne definisca anch'essa uno.

16.8 Costruttore `no-arg` di Default

Se una classe non dichiara alcun costruttore, Java inserisce automaticamente un **default no-argument constructor**.

Questo costruttore invocherà il costruttore `super()` del genitore diretto, implicitamente: il compilatore Java inserisce implicitamente una chiamata al no-arg constructor `super()`.

```
class Parent { }

class Child extends Parent {
    // Compiler inserts:
    // Child() { super(); }
}
```

16.9 Usare `this()` e Constructor Overloading

this() invoca un altro costruttore nella stessa classe.

Regole:

- Deve essere la **prima** istruzione nel costruttore
- Non può essere combinato con `super()`
- È consentita una sola chiamata a `this()` in un costruttore
- Usato per centralizzare l'inizializzazione

```
class Car {
    int year;
    String model;

    Car() {
        this(2020, "Unknown");
    }

    Car(int year, String model) {
        this.year = year;
        this.model = model;
    }
}
```

16.10 Chiamare il Costruttore del Parent usando `super()`

Ogni costruttore deve chiamare un costruttore della superclasse, esplicitamente o implicitamente.

La chiamata a `super()` deve apparire come **prima** istruzione nel costruttore (a meno che non sia sostituito da `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 Suggerimenti e Trappole sul Costruttore di Default

- **Se la classe genitore non ha un no-arg constructor, la classe figlia DEVE invocare lo specifico `super(args)` esplicitamente.**
- Se la classe figlia non definisce alcun costruttore, Java non crea automaticamente un costruttore di default per questa.
- Se ci si dimentica di chiamare esplicitamente un `parent constructor` esistente, il compilatore inserisce `super()` — il quale potrebbe non esistere.

```
class Parent {
    Parent(int x) { }
}

class Child extends Parent {
    // ERROR → compiler inserts super(), but Parent() does not exist
    Child() { }
}
```

16.12 `super()` si Riferisce Sempre al Parent più diretto

Anche in lunghe catene ereditarie, `super()` invoca sempre (e soltanto) il costruttore della classe genitrice **immediata**.

```
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 Ereditare Membri

In Java, l'**accesso ai campi** e le **chiamate a metodi statici** vengono risolti a compile-time, mentre le **chiamate ai metodi di istanza** vengono risolte a runtime.

La distinzione fondamentale è:

- La `variabile` o il `metodo statico` utilizzato dipende dal **tipo dichiarato del riferimento**.
- Il `metodo di istanza` eseguito dipende dal **tipo reale dell'oggetto a runtime**.

Esempio: `Accesso ai Campi` (Non Polimorfico)

I campi vengono risolti in base al **tipo dichiarato del riferimento**, non al tipo reale dell'oggetto.

```
class Parent {
    String name = "Parent";
}

class Child extends Parent {
    String name = "Child";
}

Parent p = new Child();
System.out.println(p.name);    // Output: Parent
```

Spiegazione:

- Il riferimento `p` è dichiarato di tipo `Parent`.
- L'accesso ai campi è determinato a compile-time.
- Pertanto viene utilizzato `Parent.name`, anche se l'oggetto è di tipo `Child`.

> **Important**
>
> - I campi **non sono polimorfici**.

Esempio: `Metodi Statici` (Non Polimorfici)

Anche i metodi statici vengono risolti utilizzando il **tipo dichiarato del riferimento**.

```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
```

Spiegazione:

- I metodi statici sono collegati (binding) a compile-time.
- Il metodo scelto dipende dal tipo del riferimento (`Parent`), non dal tipo reale dell'oggetto.

> **Important**
>
> - Questo meccanismo è chiamato **method hiding**, non overriding.

Esempio: `Metodi di Istanza` (Polimorfici)

I metodi di istanza vengono risolti a runtime in base al **tipo reale dell'oggetto**.

```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
```

Spiegazione:

- Il tipo del riferimento è `Parent` .
- L'oggetto reale è di tipo `Child` .
- Java utilizza il dynamic dispatch.
- Pertanto viene eseguito `Child.print()` .

> **Important**
>
> - I metodi di istanza sono **polimorfici**.

16.13.1 Method Overriding

Il `method overriding` è un concetto fondamentale dell'ereditarietà: permette a una classe figlia di fornire una **nuova implementazione** per un metodo già definito in una sua classe parent.

A runtime, la versione del metodo eseguita dipende dal **tipo reale dell'oggetto**, non dal particolare `reference type` .

Questo comportamento è chiamato **dynamic dispatch** ed è ciò che rende possibile il polimorfismo in Java.

16.13.1.1 Definizione e Ruolo nell'Ereditarietà

Un metodo in una subclass fa **override** di un metodo di una sua superclass se:

- il metodo della superclass è `metodo d'istanza` (non statico).
- il metodo della subclass ha lo **stesso nome**, la **stessa lista di parametri** e un **return type che è dello stesso tipo** o di un **sottotipo** del return type nel metodo ereditato.
- Quando il tipo di ritorno del metodo sovrascritto (cioè il metodo nella classe base/superclasse) è un tipo **primitivo**, il tipo di ritorno del metodo che lo sovrascrive (cioè il metodo nella sottoclasse) deve corrispondere esattamente al tipo di ritorno del metodo sovrascritto.
- entrambi i metodi sono accessibili (non privati) e il metodo della subclass non è meno visibile di quello della superclass.
- Il metodo in overriding **non può dichiarare nuove o più ampie checked exceptions**.

L'Overriding è usato per specializzare il comportamento: una subclass può adattare o rifinire il comportamento della classe parent, pur potendo essere usata tramite un reference del tipo genitore.

```
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)
    }
}
```

Non è possibile sovrascrivere un metodo di istanza con un metodo statico, né sovrascrivere un metodo statico con un metodo di istanza.

Tuttavia, una sottointerfaccia può ridichiarare un metodo statico di una superinterfaccia come metodo `default`.

- Esempio :

```
class Alpha {
    static void a() { }
    void b() { }
    static void c() { }
    void d() { }
}

class Beta extends Alpha {
    void a() { }        // NON COMPILA (impossibile sovrascrivere un metodo statico con un met
    static void b() { } // NON COMPILA (impossibile sovrascrivere un metodo di istanza con un
    static void c() { } // VALIDO, c() in Alpha è nascosto
    void d() { }        // VALIDO, d() in Alpha è sovrascritto
}
```

16.13.1.2 Usare super per chiamare l'Implementazione del Parent

Quando una subclass fa override di un metodo, può comunque accedere all'implementazione "originaria" della superclass, tramite il riferimento `super`.

Questo è utile se si vuole riusare o estendere il comportamento definito nella 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.");
    }
}
```

Se sia la classe parent sia la classe child dichiarano un membro (attributo o metodo) con lo stesso nome, il child può accedere a entrambi:

- la versione in overriding (default)
- la versione del parent tramite `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 Regole di Overriding (Instance Methods)

- **Stessa firma (signature)**: stesso nome di metodo, stessi tipi e ordine dei parametri.
- **return type coovariante**: il metodo in overriding può restituire (ritornare) lo stesso tipo del parent, o un **subtype** del return type del parent.
- **Accessibilità**: il metodo in overriding non può essere meno accessibile del metodo originario (ad esempio, non si può passare da `public` a `protected` o `private`). Può soltanto mantenere la stessa visibilità o aumentarla.
- **Checked exceptions**: il metodo in overriding non può dichiarare nuove o più ampie `checked exceptions` rispetto al `parent method`; può dichiararne meno, dichiarare checked exceptions più specifiche o, eventualmente, rimuoverle completamente.
- **Unchecked exceptions**: possono essere aggiunte o rimosse senza restrizioni.
- **final methods**: non possono partecipare all' `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 Nascondere Static Methods (Method Hiding)

I metodi statici non partecipano all' `overriding`; risultano invece, eventualmente, nascosti (**hidden**).

Se una subclass definisce uno static method con la stessa firma di uno static method della classe parent, il metodo statico della subclass **nasconde** quello della classe genitrice.

Se uno dei metodi invece è marcato come `static` e l'altro no, il codice non compilerà.

La selezione del metodo per i metodi statici avviene a **compile time** ed è basata sul `reference type`: non sull' `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

- metodi statici **final** non possono essere `hidden` (nascosti); metodi d'istanza dichiarati **final** non possono essere `overriden`.
- Se si prova a ridefinirli in una subclass, il codice non compilerà.

16.13.2 Abstract Classes

16.13.2.1 Definizione e Scopo

Una **abstract class** è una classe che non può essere istanziata direttamente ed è destinata a essere estesa.

Può contenere:

- metodi abstract (dichiarati senza body);
- metodi concreti (con implementazione);
- attributi, costruttori, membri statici, e anche static initializers.

Le abstract classes sono usate quando si vuole definire un comportamento comune (e un contratto) di **base**, ma lasciare alcuni dettagli da implementare alle subclasses concrete.

16.13.2.2 Regole per le Abstract Classes

- Una classe con almeno un metodo astratto **deve** essere dichiarata `abstract`.
- Una `abstract class` **non può** essere istanziata direttamente.
- I metodi astratti non hanno body e terminano con un punto e virgola.
- Gli **abstract methods non possono essere `final`, `static` o `private`**, perché devono essere ridefinibili `overridable`.
- La prima subclass concreta (non-abstract) nella gerarchia, deve implementare tutti gli `abstract methods` ereditati, altrimenti deve essere dichiarata anch'essa `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**
>
> - Sebbene una `abstract class` non possa essere istanziata, i suoi costruttori vengono comunque chiamati quando si creano istanze di classi figlie concrete.
> - Il flusso delle istanziazioni, nella `catena ereditaria`, parte sempre dal top della gerarchia e si muove verso il basso.

16.13.3 Creare Oggetti Immutabili

16.13.3.1 Cos'è un Oggetto `Immutable`

Un oggetto è **immutable** se, dopo che è stato creato, il suo stato **non può cambiare**.

Tutti gli attributi che ne rappresentano lo stato, rimangono costanti per l'intero ciclo di vita di quell'oggetto.

Gli `immutable objects` sono semplici da comprendere, intrinsecamente `thread safe` (se progettati correttamente), e ampiamente usati nella Java Standard Library (ad esempio `String`, wrapper classes come `Integer`, e molte classi in `java.time`).

16.13.3.2 Linee Guida per Progettare Classi Immutable

- Dichiarare una classe **final** cosicché non possa essere estesa (oppure rendere tutti i costruttori privati e fornire factory methods protetti).
- Rendere tutti gli attributi che ne rappresentano lo stato **private** e **final**.
- Non fornire alcun `mutator` (setter) methods.
- Inizializzare tutti gli attributi nei costruttori (o nei factory methods) e non esporli mai in modo `mutabile`.
- Se un attributo si riferisce ad un oggetto mutabile, fare **defensive copies** (copie difensive) in fase di costruzione e quando lo si restituisce tramite `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);
    }
}
```

In questo esempio:

- `Person` è **final**: non può essere estesa e il suo comportamento non può essere alterato tramite `inheritance`.
- Tutti gli attributi sono `private` e `final`, definiti una sola volta nel costruttore.
- La lista degli `hobbies` viene copiata difensivamente nella fase di costruzione e wrappata come `unmodifiable` (non modificabile) nel `metodo getter`, cosicché alcun codice esterno ne possa modificare lo stato interno.

Progettare `immutable objects` è particolarmente importante in contesti multithread e quando si passano oggetti attraverso i diversi layers di una applicazione.

◀ 15. Caricamento delle Classi, Inizializzazione e Costruzione degli Oggetti | ▲ Index | 17. Oltre le Classi ▶

17. Oltre le Classi

Indice

- 17.1 Interfacce
  - 17.1.1 Cosa Possono Contenere le Interfacce
  - 17.1.2 Implementare un'Interfaccia
  - 17.1.3 Ereditarietà Multipla
  - 17.1.4 Ereditarietà delle Interfacce e Conflitti
  - 17.1.5 Metodi default
  - 17.1.6 Metodi static
  - 17.1.7 Metodi private nelle interfacce
- 17.2 Tipi sealed, non-sealed e final
  - 17.2.1 Regole
- 17.3 Enum
  - 17.3.1 Definizione di Enum Semplice
  - 17.3.2 Enum Complesse con Stato e Comportamento
  - 17.3.3 Metodi delle Enum
  - 17.3.4 Regole
- 17.4 Record (Java 16+)
  - 17.4.1 Riepilogo delle Regole di Base per i Record
  - 17.4.2 Costruttore Lungo
  - 17.4.3 Costruttore Compatto
  - 17.4.4 Pattern Matching per i Record
  - 17.4.5 Nested Record Patterns e Matching dei Record con var e Generics
    - 17.4.5.1 Nested Record Pattern di Base
    - 17.4.5.2 Nested Record Patterns con var
    - 17.4.5.3 Nested Record Patterns e Generics
    - 17.4.5.4 Errori Comuni con i Nested Record Patterns
- 17.5 Classi Annidate in Java
  - 17.5.1 Static Nested Classes
    - 17.5.1.1 Sintassi e Regole di Accesso
    - 17.5.1.2 Errori Comuni
  - 17.5.2 Inner Classes (Non-Static Nested Classes)
    - 17.5.2.1 Sintassi e Regole di Accesso
    - 17.5.2.2 Errori Comuni
  - 17.5.3 Classi Locali
    - 17.5.3.1 Caratteristiche
    - 17.5.3.2 Errori Comuni
  - 17.5.4 Classi Anonime
    - 17.5.4.1 Sintassi e Utilizzo
    - 17.5.4.2 Classe Anonima che Estende una Classe
  - 17.5.5 Confronto dei Tipi di Classi Annidate
- 17.6 Nesting delle Interfacce in Java
  - 17.6.1 Dove può essere dichiarata un'interfaccia

- 17.6.2 Interfacce annidate (Nested Interfaces)
  - 17.6.2.1 Interfaccia annidata in una Classe
  - 17.6.2.2 Interfaccia annidata in una un’altra Interfaccia
- 17.6.3 Regole di Accesso
- 17.6.4 Nested Types nelle Interfacce
- 17.6.5 Riassunto Essenziale

Questo capitolo presenta diversi meccanismi avanzati di tipo (type) oltre quello, già visto, della Classe: **interface**, **enum**, **classi sealed / non-sealed**, **record** e **classi annidate**.

17.1 Interfacce

Un’**interfaccia** in Java è un tipo di riferimento che definisce un contratto di metodi che una classe accetta di implementare.

Una `interface` è implicitamente `abstract` e non può essere marcata come `final`: come per le classi top-level, un’interfaccia può dichiarare visibilità come `public` o `default` (package-private).

Una classe Java può implementare un numero qualsiasi di interfacce tramite la keyword `implements`.

Un `interface` può a sua volta estendere più interfacce usando la keyword `extends`.

Le interfacce abilitano astrazione, accoppiamento lasco e ereditarietà multipla di tipo.

17.1.1 Cosa Possono Contenere le Interfacce

- **Metodi astratti** (implicitamente `public` e `abstract`)
- **Metodi concreti**
  - **Metodi default** (includono codice e sono implicitamente `public`)
  - **Metodi static** (dichiarati come `static`, includono codice e sono implicitamente `public`)
  - **Metodi private** (Java 9+) per riuso interno
- **Costanti** → implicitamente `public static final` e inizializzate alla dichiarazione

```
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**
>
> Poiché i metodi astratti delle interfacce sono implicitamente `public`, **non puoi** ridurre il livello di accesso su un metodo di implementazione.

17.1.2 Implementare un’Interfaccia

```
class BasicCalc implements Calculator {
    public int add(int a, int b) { return a + b; }
}
```

> **Note**
>
> **Ogni** metodo astratto deve essere implementato a meno che la classe non sia astratta essa stessa.

17.1.3 Ereditarietà Multipla

Una classe può implementare più interfacce.

```
interface A { void a(); }
interface B { void b(); }

class C implements A, B {
    public void a() {}
    public void b() {}
}
```

17.1.4 Ereditarietà delle Interfacce e Conflitti

Se due interfacce forniscono metodi `default` con la stessa signature, la classe che implementa deve fare override del metodo.

```
interface X { default void run() { } }
interface Y { default void run() { } }

class Z implements X, Y {
    public void run() { } // mandatory
}
```

Se vuoi comunque accedere a una particolare implementazione del metodo `default` ereditato, puoi usare la seguente sintassi:

```
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 Metodi `Default`

Un metodo `default` (dichiarato con la parola chiave `default`) è un metodo che definisce un implementazione e può essere sovrascritto da una classe che implementa l interfaccia.

- Un metodo default contiene codice ed è implicitamente `public` ;
- Un metodo default non può essere `abstract`, `static` o `final` ;
- Un'interfaccia può ridichiarare un metodo default e fornire una implementazione alternativa ;
- Una sottointerfaccia può ridichiarare un metodo statico di una superinterfaccia come metodo `default` ;
- Come abbiamo visto appena sopra, se due interfacce forniscono metodi default con la stessa firma, la classe che implementa deve sovrascrivere il metodo ;
- Una classe che implementa può naturalmente fare affidamento sull implementazione fornita dal metodo `default` senza sovrascriverlo ;
- Il metodo `default` può essere invocato su un istanza della classe che implementa e NON come metodo `static` dell interfaccia che lo contiene ;
- Una classe (o un interfaccia) può invocare esplicitamente un metodo default di un interfaccia che è direttamente menzionata nella sua clausola `implements` (o `extends`) utilizzando la sintassi `InterfaceName.super.methodName()` ; ciò è tipicamente utilizzato per risolvere ambiguità tra più metodi default ereditati ;
- Questa sintassi può essere utilizzata solo se l interfaccia è esplicitamente menzionata nella clausola `implements` (o `extends`) ; non può essere utilizzata per invocare un metodo default proveniente

da un interfaccia ereditata indirettamente ;

- La sintassi `InterfaceName.super.methodName()` si applica esclusivamente ai metodi `default` e non può essere utilizzata per metodi astratti, metodi statici, metodi privati di interfaccia o campi.

Example: Uso valido

```
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: Uso errato

```
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**
>
> - Una sottointerfaccia può ridichiarare un metodo statico di una superinterfaccia come metodo `default` ;

Esempio:

```
interface Parent {
    static void p() { }
}

interface Child extends Parent {
    default void p() { } // VALID, static method redeclared as default
}
```

> **Note**
>
> - Un interfaccia è autorizzata a ridichiarare un metodo `default` ereditato da una superinterfaccia e a trasformarlo in un metodo `abstract` .
>
> Quando ciò accade, l implementazione `default` proveniente dalla superinterfaccia viene effettivamente rimossa nella sottointerfaccia. Di conseguenza, qualsiasi classe che implementa la sottointerfaccia NON erediterà l implementazione `default` originale e dovrà fornire una propria implementazione.

Esempio:

```
interface Parent {
    default void greet() {
        System.out.println("Hello from Parent");
    }
}

interface Child extends Parent {
    void greet();    // ridichiarato come abstract
}

class Demo implements Child {
    public void greet() {    // obbligatorio
        System.out.println("Hello from Demo");
    }
}
```

Spiegazione:

- `Parent` fornisce un implementazione di default di `greet()`.
- `Child` ridichiara `greet()` senza `default`, rendendolo nuovamente astratto.
- `Demo` non può ereditare l implementazione di default di `Parent`.
- Pertanto, `Demo` deve implementare esplicitamente `greet()`.

17.1.6 Metodi `static`

- Un'interfaccia può fornire `static methods` (tramite la keyword `static`) che sono implicitamente `public`;
- I metodi static devono includere un corpo del metodo e sono accessibili usando il nome dell'interfaccia;
- I metodi static non possono essere `abstract` o `final`;

17.1.7 Metodi `private` nelle interfacce

Tra tutti i metodi concreti che un'interfaccia può implementare, abbiamo anche:

- **Metodi `private`**: visibili solo all'interno dell'interfaccia dichiarante e che possono essere invocati solo da un contesto `non-static` (metodi `default` o altri `non-static private methods`).
- **Metodi `private static`**: visibili solo all'interno dell'interfaccia dichiarante e che possono essere invocati da qualsiasi metodo dell'interfaccia contenitore.

17.2 Tipi `sealed`, `non-sealed` e `final`

Le classi e le interfacce `sealed` (Java 17+) restringono quali altre classi (o interfacce) possono estenderle o implementarle.

Un `sealed type` è dichiarato mettendo il modificatore `sealed` subito prima della keyword class (o interface), e aggiungendo, dopo il nome del Tipo, la keyword `permits` seguita dalla lista dei tipi che possono estenderlo (o implementarlo).

```
public sealed class Shape permits Circle, Rectangle { }

final class Circle extends Shape { }

non-sealed class Rectangle extends Shape { }
```

17.2.1 Regole

- Un tipo sealed deve dichiarare tutti i sottotipi permessi.
- Un sottotipo permesso deve essere **final**, **sealed** o **non-sealed**; poiché le interfacce non possono essere final, possono essere marcate solo `sealed` o `non-sealed` quando estendono un'interfaccia sealed.
- Se una sealed class appartiene a un `named` module, allora tutte le classi elencate nella sua permits clause devono anch esse appartenere a quello stesso `module`.

- Se una sealed class appartiene a un `unnamed module`, allora tutte le classi elencate nella sua permits clause devono essere dichiarate nello `stesso package`.

17.3 Enum

Le **enum** definiscono un insieme fisso di valori costanti.

Le `enum` possono dichiarare `attributi`, `costruttori` e `metodi` come le classi regolari ma **non possono essere estese**.

La lista dei valori dell'enum deve terminare con un punto e virgola `(;)` nel caso di `Enum Complesse`, ma questo non è obbligatorio per `Enum Semplici`.

17.3.1 Definizione di Enum `Semplice`

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN } // punto e virgola omesso
```

17.3.2 Enum `Complesse` con Stato e Comportamento

```
enum Level {
    LOW(1), MEDIUM(5), HIGH(10); // punto e virgola obbligatorio

    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 Metodi delle Enum

- `values()` – restituisce un array di tutti i valori costanti che possono essere usati, per esempio, in un ciclo `for-each`
- `valueOf(String)` – restituisce la costante per nome
- `ordinal()` – indice (int) della costante

17.3.4 Regole

- **I costruttori delle enum sono implicitamente** `private`;
- Le enum possono contenere metodi `static` e `instance`;
- Le enum possono implementare `interfaces`;
- Le enum non possono essere estese.

17.4 Record (Java 16+)

Un **record** è una classe speciale progettata per modellare dati immutabili: sono infatti implicitamente **final**.

Non puoi estendere un record, ma è permesso implementare un'interfaccia `regolare` o `sealed`.

Fornisce automaticamente:

- **campi private final** per ogni componente;
- **costruttore** con parametri nello stesso ordine della dichiarazione del record;
- **getters** (con nome degli attributi);
- **`equals()`**, **`hashCode()`**, **`toString()`**: è inoltre permesso fare override di questi metodi;

- I **Record** possono includere `nested classes`, `interfaces`, `records`, `enums` e `annotations`.

```
public record Point(int x, int y) { }

var element = new Point(11, 22);

System.out.println(element.x);
System.out.println(element.y);
```

Se ti serve validazione o trasformazione aggiuntiva dei campi forniti, puoi definire un `costruttore lungo` o un `costruttore compatto`.

17.4.1 Riepilogo delle Regole di Base per i Record

Un record può essere dichiarato in tre posizioni:

- Come **record top-level** (direttamente in un package)
- Come **record member** (come membro, all'interno di una classe o interfaccia)
- Come **record local** (all'interno di un metodo)

Tutte le classi record `member` e `local` sono implicitamente `static`.

- Un record member può dichiarare `static` in modo ridondante.
- Un record local non deve dichiarare `static` esplicitamente.

Ogni classe record è implicitamente `final`.

- Dichiarare `final` esplicitamente è consentito ma ridondante.
- Un record non può essere dichiarato `abstract`, `sealed` o `non-sealed`.

La superclasse diretta di ogni record è `java.lang.Record`.

- Un record non può dichiarare una clausola `extends`.
- Un record non può estendere nessun'altra classe.

La serializzazione dei record è diversa rispetto alle classi serializzabili ordinarie.

- Durante la deserializzazione viene invocato il costruttore canonico.

Il corpo di un record può contenere:

- Costruttori
- Metodi
- Campi statici
- Blocchi di inizializzazione statici

Il corpo di un record NON deve contenere:

- Dichiarazioni di campi di istanza
- Blocchi di inizializzazione di istanza
- Metodi `abstract`
- Metodi `native`

17.4.2 Costruttore Lungo

```
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;
    }
}
```

Puoi anche definire costruttori in overload, purché alla fine deleghino a quello canonico usando `this(...)`:

```
public record Point(int x, int y) {

    // Overloaded constructor (NOT canonical)
    public Point(int value) {
        this(value, value); // deve invocare, come prima istruzione, un altro costruttore over
    }
}
```

> **Note**
>
> - Il compilatore non inserirà un costruttore se ne fornisci manualmente uno con la stessa lista di parametri nell'ordine definito;
> - In questo caso, devi impostare esplicitamente ogni campo manualmente;

17.4.3 Costruttore Compatto

Puoi definire un `costruttore compatto` che imposta implicitamente tutti i campi, permettendoti di eseguire validazioni e trasformazioni su campi specifici.

Java eseguirà il costruttore completo, impostando tutti i campi, dopo che il costruttore compatto è terminato.

```
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**
>
> - Se provi a modificare un attributo di Record dentro un Costruttore Compatto, il tuo codice non compilerà

17.4.4 Pattern Matching per i Record

Quando usi pattern matching con `instanceof` o con `switch`, un record pattern deve specificare:

- Il tipo del record;
- Un pattern per ogni campo del record (corrispondendo al numero corretto di componenti, e tipi compatibili);

Esempio 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 e Matching dei Record con `var` e Generics

I nested record patterns permettono di destrutturare record che contengono altri record o tipi complessi, estraendo valori ricorsivamente direttamente nel pattern stesso.

Combinano la potenza della destrutturazione dei `record` con il pattern matching, dandoti un modo conciso ed espressivo per navigare strutture dati gerarchiche.

17.4.5.1 Nested Record Pattern di Base

Se un record contiene un altro record, puoi destrutturare entrambi in una volta:

```
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");
    }
}
```

Nell'esempio sopra, il pattern `Person` include un pattern `Address` annidato.

Entrambi sono matchati strutturalmente.

17.4.5.2 Nested Record Patterns con `var`

Invece di specificare tipi esatti per ogni campo, puoi usare `var` dentro il pattern per lasciare al compilatore l'inferenza del tipo.

```
    switch (obj) {
        case Person(var name, Address(var city, var country)) -> System.out.println(name + " -
    }
```

`var` nei pattern funziona come `var` nelle variabili locali: significa "inferisci il tipo".

> **Warning**
>
> - Ti serve ancora il tipo del record contenitore (Person, Address);
> - solo i tipi dei campi possono essere sostituiti con `var`.

17.4.5.3 Nested Record Patterns e Generics

I record patterns funzionano anche con record generici.

```
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");
    }
}
```

In questo esempio:

- Il pattern richiede esattamente `Box<String>`, non `Box<Integer>`.
- Dentro il pattern, `var v` cattura il valore generico unboxed.

17.4.5.4 Errori Comuni con i Nested Record Patterns

Struttura record non corrispondente

```
// ❌ ERROR: pattern does not match record structure
case Person(var n, var city) -> ...
```

`Person` ha 2 campi, ma uno di questi è un record. Devi destrutturare correttamente.

Numero errato di componenti

```
// ❌ ERROR: Address has 2 components, not 1
case Person(var n, Address(var onlyCity)) -> ...
```

Mismatch generico

```
// ❌ ERROR: expecting Box<String> but found Box<Integer>
case Wrapper(Box<Integer>(var v)) -> ...
```

Posizionamento illegale di `var`

```
// ❌ var cannot replace the record type itself
case var(Person(var n, var a)) -> ...
```

> **Note**
>
> - `var` non può sostituire l'intero pattern, solo i singoli componenti.

17.5 Classi Annidate in Java

Java supporta diversi tipi di **classi annidate** — classi dichiarate dentro un'altra classe.

Sono uno strumento fondamentale per incapsulamento, organizzazione del codice, pattern di event-handling e rappresentazione di gerarchie logiche.

Una classe annidata appartiene sempre a una **classe contenitore** e ha regole speciali di accessibilità e istanziazione a seconda della sua categoria.

Java definisce quattro tipi di classi annidate:

- **Static Nested Classes** – dichiarate con `static` dentro un'altra classe.
- **Inner Classes** (non-static **nested** classes).
- **Local Classes** – dichiarate dentro un blocco (metodo, costruttore o initializer).
- **Anonymous Classes** – classi senza nome create inline, di solito per fare override di un metodo o implementare un'interfaccia.

> **Warning**
>
> - `static` si applica solo alle classi **membro nested**
> - Le classi `Top-level` → non possono essere static
> - Le classi `Local` (dichiarate nei metodi) → non possono essere static
> - Le classi `Anonymous` → non possono essere static
> - Una classe `static nested` non può accedere ai membri di istanza senza un riferimento esplicito a un oggetto esterno.

17.5.1 Static Nested Classes

Una **static nested class** si comporta come una classe top-level con namespace dentro la sua classe contenitore.
Non **può** accedere ai membri d'istanza della classe esterna ma **può** accedere ai membri statici.
Non mantiene un riferimento a un'istanza della classe contenitore. Una classe annidata `static` può contenere variabili membro non statiche.

17.5.1.1 Sintassi e Regole di Accesso

- Dichiarata usando `static class` dentro un'altra classe.
- Può accedere solo ai membri **static** della classe esterna.
- Non ha un riferimento implicito all'istanza contenitore.
- Può essere istanziata senza un'istanza esterna.

- Può contenere variabili membro non statiche

```
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 Errori Comuni

- Le static nested classes **non possono accedere alle variabili d'istanza**:

```
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)

Una **inner class** è associata a un'istanza della classe esterna e può accedere a **tutti i membri** della classe esterna, inclusi quelli **private**.

17.5.2.1 Sintassi e Regole di Accesso

- Dichiarata senza `static`.
- Ha un riferimento implicito all'istanza contenitore.
- Può accedere sia ai membri statici sia ai membri d'istanza della classe esterna.
- Poiché non è statica, deve essere creata tramite un'istanza della classe contenitore.

```
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();
    }
}
```

All'interno di una classe interna `non-static`, è possibile fare riferimento all'oggetto esterno (enclosing object) utilizzando `OuterClass.this`.
L'espressione `InnerClass.this`, equivalente a `this`, si riferisce invece all'oggetto `Inner` corrente.

- Esempio:

```
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 Errori Comuni

- Le inner classes **non possono dichiarare membri statici** eccetto **static final constants**.

```
class Outer {
    class Inner {
        // static int x = 10;     // ❌ Compile error
        static final int OK = 10; // ✓ Allowed (constant)
    }
}
```

> **Warning**
>
> - Istanziare una inner class SENZA un'istanza esterna è illegale.

17.5.3 Classi Locali

Una **classe locale** è una classe annidata definita dentro un blocco — più comunemente un metodo.

Non ha modificatori di accesso ed è visibile solo dentro il blocco in cui è dichiarata.

17.5.3.1 Caratteristiche

- Dichiarata dentro un metodo, costruttore o initializer.
- Può accedere ai membri della classe esterna.
- Può accedere a variabili locali se sono **effectively final**.
- Non può dichiarare membri statici (eccetto 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();
    }
}
```

Una classe locale, proprio come una classe interna membro, possiede un riferimento implicito all'istanza esterna tramite `OuterClass.this`.
Dispone inoltre di `LocalClass.this`, equivalente a `this`, che è valido all'interno del corpo della classe locale.

- Esempio:

```
class Outer {
    int x = 10;

    void method() {
        class Local {
            void print() {
                System.out.println(Outer.this.x);   // ✓ valido

                System.out.println(Local.this);     // ✓ valido
            }
        }
    }
}
```

17.5.3.2 Errori Comuni

- `base` deve essere effectively final; cambiarla rompe la compilazione.

```
void compute() {
    int base = 5;
    base++; // ❌ Now base is NOT effectively final
    class Local {}
}
```

17.5.4 Classi Anonime

Una **classe anonima** è una classe one-off creata inline, di solito per implementare un'interfaccia o fare override di un metodo senza nominare una nuova classe.

17.5.4.1 Sintassi e Utilizzo

- Creata usando `new` + tipo + body.
- Non può avere costruttori (nessun nome).
- Spesso usata per event handling, callbacks, comparators.

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonymous running");
    }
};
```

17.5.4.2 Classe Anonima che Estende una Classe

```
Button b = new Button("Click");
b.onClick(new ClickHandler() {
    @Override
    public void handle() {
        System.out.println("Handled!");
    }
});
```

17.5.5 Confronto dei Tipi di Classi Annidate

Una tabella rapida che riassume tutti i tipi di classi annidate.

Tipo	Ha un'Istanza Esterna?	Può Accedere ai Membri d'Istanza Esterna?	Può Avere Membri Statici?	Uso Tipico
Static Nested	No	No	Sì	Namespacing, helpers
Inner Class	Sì	Sì	No (eccetto costanti)	Comportamento legato all'oggetto
Local Class	Sì	Sì	No	Classi temporanee con scope
Anonymous Class	Sì	Sì	No	Personalizzazione inline

17.6 Nesting delle Interfacce in Java

In Java, un'interfaccia può essere dichiarata in diverse posizioni e seguire regole specifiche riguardo al nesting e ai membri consentiti.

17.6.1 Dove può essere dichiarata un'interfaccia

Un'interfaccia può essere:

- **Top-level** (direttamente in un package)
- **Nested member interface** (dichiarata all'interno di una classe o di un'altra interfaccia)
- **Local interface** ❌ (non consentita)
- **Anonymous interface** ❌ (non esiste come dichiarazione, solo implementazioni anonime)

In Java **non è permesso dichiarare un'interfaccia locale** (cioè dentro un metodo o blocco).
Le interfacce possono essere solo `top-level` o `member`.

17.6.2 Interfacce annidate (Nested Interfaces)

Una Nested Interface può essere dichiarata dentro:

17.6.2.1 Interfaccia annidata in una Classe

- È implicitamente `static`
- Non può essere dichiarata `non-static`
- Può essere dichiarata `public`, `protected`, `private` o `package-private`
- Esempio:

```
class Outer {
    interface InnerInterface {
        void test();
    }
}
```

La parola chiave `static` è implicita:

```
class Outer {
    static interface InnerInterface {   // consentito ma ridondante
        void test();
    }
}
```

17.6.2.2 Interfaccia annidata in una un'altra Interfaccia

- È implicitamente `public` e `static`
- Non può essere `private` o `protected`

```
interface A {
    interface B {
        void test();
    }
}
```

17.6.3 Regole di Accesso

Una `nested interface`:

- Non ha riferimento implicito a un'istanza della classe esterna
- Non può accedere direttamente ai membri di istanza della classe esterna
- **Può accedere solo ai membri `static` della classe esterna**

17.6.4 Nested Types nelle Interfacce

Un'interfaccia può contenere:

- Classi annidate (implicitamente `public static`)

- Record annidati (implicitamente `public static`)
- Enum annidati (implicitamente `public static`)
- Altre interfacce annidate (implicitamente `public static`)

17.6.5 Riassunto Essenziale

- Le interfacce nested sono sempre `static`
- Non esistono interfacce locali
- I campi sono sempre `public static final`
- I metodi sono implicitamente `public abstract` (salvo default/static/private)
- Possono contenere altri tipi nested

◀ 16. Ereditarietà in Java | ▲ Index | 18. Generics in Java ▶

18. Generics in Java

Indice

- 18.1 Basi dei Tipi Generici
- 18.2 Perché Esistono i Generics
- 18.3 Metodi Generici
- 18.4 Type Erasure
  - 18.4.1 Come Funziona la Type Erasure
  - 18.4.2 Erasure dei Parametri di Tipo Senza Bound
  - 18.4.3 Erasure dei Parametri di Tipo con Bound
  - 18.4.4 Bound Multipli: Il Primo Bound Determina l'Erasure
  - 18.4.5 Perché Solo il Primo Bound Diventa il Tipo a Runtime
  - 18.4.6 Un Esempio Più Complesso
  - 18.4.7 Override e Generics
    - 18.4.7.1 Come il compilatore valida un override
    - 18.4.7.2 Parametri generici e override
    - 18.4.7.3 Override valido — Eliminazione della specificità generica
    - 18.4.7.4 Override non valido — Aggiunta di specificità generica
    - 18.4.7.5 Override valido — Parametrizzazione identica
    - 18.4.7.6 Override non valido — Modifica dell'argomento generico
    - 18.4.7.7 Perché esiste questa regola
    - 18.4.7.8 Modello mentale
    - 18.4.7.9 Ritorni covarianti e Generics
    - 18.4.7.10 Regole riassuntive
  - 18.4.8 Overloading di un Metodo Generico — Perché Alcuni Overload Sono Impossibili
  - 18.4.9 Overloading di un Metodo Generico Ereditato da una Classe Parent
  - 18.4.10 Restituire Tipi Generici — Regole e Restrizioni
  - 18.4.11 Riepilogo delle Regole di Erasure
- 18.5 Bound sui Parametri di Tipo
  - 18.5.1 Upper Bounds: extends
  - 18.5.2 Bound Multipli
  - 18.5.3 Wildcard: ?, ? extends, ? super
    - 18.5.3.1 Wildcard Non Limitata
    - 18.5.3.2 Wildcard con Upper Bound extends
    - 18.5.3.3 Wildcard con Lower Bound super
- 18.6 Generics ed Ereditarietà
- 18.7 Type Inference (Operatore Diamond)
- 18.8 Raw Types (Compatibilità Legacy)
- 18.9 Array Generici (Non Permessi)
- 18.10 Bounded Type Inference
- 18.11 Wildcard vs Parametri di Tipo
- 18.12 Regola PECS (Producer Extends, Consumer Super)
- 18.13 Errori Comuni
- 18.14 Tabella Riassuntiva delle Wildcard
- 18.15 Riepilogo dei Concetti
- 18.16 Esempio Completo

Java `Generics` permettono di creare classi, interfacce e metodi che lavorano con tipi specificati dall'utente, garantendo che vengano usati solo oggetti del tipo corretto.

Tutti i controlli di tipo vengono eseguiti dal compilatore a compile-time.

Durante la compilazione, il compilatore verifica i tipi e poi rimuove le informazioni generiche (processo identificato come **type erasure**), sostituendole con i tipi reali o con Object quando necessario.

Il bytecode risultante non contiene generics: contiene solo i tipi concreti e, se serve, cast inseriti automaticamente dal compilatore.

In questo modo, gli errori di tipo vengono intercettati prima dell'esecuzione, rendendo il codice più sicuro, leggibile e riutilizzabile.

I Generics si applicano a:

- `Classi`
- `Interfacce`
- `Metodi` (metodi generici)
- `Costruttori`

18.1 Basi dei Tipi Generici

Una classe o interfaccia generica introduce uno o più **parametri di tipo**, racchiusi tra parentesi angolari.

```
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(); // nessun cast necessario
```

Sono permessi più parametri di tipo:

```
class Pair<K, V> {
    K key;
    V value;
}
```

18.2 Perché Esistono i Generics

```
List list = new ArrayList();           // pre-generics
list.add("hi");

Integer x = (Integer) list.get(0);     // ClassCastException a runtime
```

Con i generics:

```
List<String> list = new ArrayList<>();
list.add("hi");

String x = list.get(0);                // type-safe, nessun cast
```

18.3 Metodi Generici

Un **metodo generico** introduce i propri parametri di tipo, indipendenti dalla classe.

```
class Util {

    static <T> T pick(T a, T b) { return a; }

}

String s = Util.<String>pick("A", "B"); // esplicito
String t = Util.pick("A", "B");         // l'inferenza funziona
```

18.4 Type Erasure

La `Type erasure` è il processo attraverso cui il compilatore Java rimuove tutte le informazioni sui tipi generici prima di generare il bytecode.

Questo garantisce compatibilità con le JVM precedenti a Java 5.

A `compile time`, i generics sono completamente controllati: bound sui tipi, varianza, overloading di metodi generici, ecc.

Tuttavia, a runtime, tutte le informazioni generiche scompaiono.

18.4.1 Come Funziona la Type Erasure

- Sostituire tutte le variabili di tipo (come `T`) con il loro tipo erasure.
- Inserire cast dove necessario.
- Rimuovere tutti gli argomenti di tipo generico (es. `List<String>` → `List`).

18.4.2 Erasure dei Parametri di Tipo Senza Bound

Se una variabile di tipo non ha bound:

```
class Box<T> {
    T value;
    T get() { return value; }
}
```

L'erasure di `T` è `Object`.

```
class Box {
    Object value;
    Object get() { return value; }
}
```

18.4.3 Erasure dei Parametri di Tipo con Bound

Se il parametro di tipo ha bound:

```
class TaskRunner<T extends Runnable> {

    void run(T task) { task.run(); }

}
```

Allora l'erasure di `T` è il primo bound trovato dal compilatore: in questo specifico caso `Runnable`.

```
class TaskRunner {
    void run(Runnable task) { task.run(); }
}
```

18.4.4 Bound Multipli: Il Primo Bound Determina l'Erasure

Java permette bound multipli:

```
<T extends Runnable & Serializable & Cloneable>
```

> **Important**
>
> L'erasure di `T` è sempre il **primo bound**, che deve essere una classe o interfaccia.

Poiché `Runnable` è il primo bound, il compilatore effettua l'erasure di `T` a `Runnable`.

- Esempio con Bound Multipli (Completamente Espanso)

```
public static <T extends Runnable & Serializable & Cloneable>
void runAll(List<T> list) {
    for (T t : list) {
        t.run();
    }
}
```

Versione con Erasure:

```
public static void runAll(List list) {
    for (Object obj : list) {
        Runnable t = (Runnable) obj;    // cast inserito dal compilatore
        t.run();
    }
}
```

Cosa succede agli altri bound (Serializable, Cloneable)?

- Sono applicati solo a compile time.
- NON compaiono nel bytecode.
- Nessuna interfaccia aggiuntiva viene associata al tipo con erasure.

18.4.5 Perché Solo il Primo Bound Diventa il Tipo a Runtime?

Perché la JVM deve operare usando un singolo tipo di riferimento concreto per ogni variabile o parametro.

Le istruzioni bytecode a runtime come `invokevirtual` richiedono una singola classe o interfaccia, non un tipo composto come "Runnable & Serializable & Cloneable".

> **Note**
>
> Java seleziona il **primo bound** come tipo a runtime, e usa i bound restanti solo per la **validazione a compile-time**.

18.4.6 Un Esempio Più Complesso

```
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();
    }
}
```

Versione con Erasure:

```
class Demo {
    void test(A value) {
        value.a();
        // value.b();   // ❌ non disponibile dopo l'erasure: il tipo è A, non B
    }
}
```

> **Note**
>
> Il compilatore può inserire cast aggiuntivi o metodi bridge in scenari di ereditarietà più complessi, ma l'erasure usa sempre solo il primo bound (A in questo caso).

18.4.7 Override e Generics

Quando i generics interagiscono con l'ereditarietà, è fondamentale comprendere chiaramente due regole:

> **Important**
>
> **L'override viene verificato dopo la type erasure.**
>
> **La compatibilità dei tipi viene verificata prima della type erasure.**

Questi due passaggi spiegano perché alcuni metodi effettuano correttamente l'override mentre altri producono errori di compilazione.

18.4.7.1 Come il compilatore valida un override

Quando una sottoclasse dichiara un metodo che *potrebbe* effettuare l'override di un metodo della superclasse, il compilatore esegue due controlli:

1. **Prima della erasure**
    - Il metodo deve essere compatibile a livello di tipo con quello della classe padre:
        - Stesso nome del metodo
        - Stessi tipi dei parametri (inclusi gli argomenti generici)
        - Tipo di ritorno compatibile (covarianza ammessa)
2. **Dopo la erasure**
    - Le firme erase devono coincidere esattamente.
        - Entrambe le condizioni devono essere soddisfatte.

18.4.7.2 Parametri generici e override

Gli argomenti di tipo generico fanno parte della firma del metodo **a compile-time**, ma scompaiono dopo la erasure.

Per questo motivo:

- È consentito **eliminare l'informazione generica nel metodo che effettua override**
- Non è consentito **aggiungere nuova specificità generica**
- Se entrambi i metodi dichiarano tipi parametrizzati, devono coincidere esattamente

18.4.7.3 Override valido — Eliminazione della specificità generica

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    @Override
    void process(Set data) {}   // ✓ consentito (raw type)
}
```

Spiegazione:

- Prima della erasure: `Set` è assignment-compatible con `Set<Integer>`

- Dopo la erasure: entrambi diventano `Set`

✓ Override valido.

18.4.7.4 Override non valido — Aggiunta di specificità generica

```
class Parent {
    void process(Set data) {}
}

class Child extends Parent {
    void process(Set<Integer> data) {}    // ✗ errore di compilazione
}
```

Spiegazione:

- Prima della erasure: `Set<Integer>` NON è assignment-compatible con `Set`
- Il compilatore lo rifiuta prima ancora di considerare la erasure

18.4.7.5 Override valido — Parametrizzazione identica

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    @Override
    void process(Set<Integer> data) {}    // ✓ corrispondenza esatta
}
```

Entrambi i controlli sono soddisfatti:

- Compatibilità prima della erasure
- Firma identica dopo la erasure

18.4.7.6 Override non valido — Modifica dell'argomento generico

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    void process(Set<String> data) {}    // ✗ errore di compilazione
}
```

Spiegazione:

- Prima della erasure: `Set<String>` non è compatibile con `Set<Integer>`
- Dopo la erasure: entrambi diventerebbero `Set`
- Collisione + incompatibilità → errore di compilazione

18.4.7.7 Perché esiste questa regola

Java deve garantire:

- **Type safety a compile-time**
- **Polimorfismo a runtime dopo la erasure**

Poiché i generics scompaiono a runtime, la JVM vede solo le firme erase. Il compilatore deve quindi garantire compatibilità prima della erasure e coerenza dopo la erasure.

18.4.7.8 Modello mentale

Considera l'override con generics come un controllo in due fasi:

```
Fase 1 → I tipi a livello di sorgente sono compatibili?
Fase 2 → Le firme erase coincidono?
```

Se una delle due fasi fallisce → errore di compilazione.

18.4.7.9 Ritorni Covarianti e Generics

Un metodo che effettua l’override (cioè un metodo dichiarato in una sottoclasse) è autorizzato a restituire un **sottotipo** del tipo di ritorno dichiarato nel metodo sovrascritto (cioè nel metodo della superclasse).

Questa è nota come **regola dei ritorni covarianti**.

Il primo passo nella validazione di un override consiste quindi nel:

- Verificare se il tipo di ritorno del metodo nella sottoclasse è un sottotipo del tipo di ritorno dichiarato nella superclasse.

> **Important**
>
> - Se il metodo sovrascritto restituisce `List`, il metodo nella sottoclasse può restituire `ArrayList`.
> - Non può **restituire** `Object`, poiché `Object` è un supertipo e non un sottotipo.

Quando entrano in gioco i generics, la validazione del tipo di ritorno diventa più delicata.

Occorre valutare le relazioni di sottotipizzazione utilizzando le regole della gerarchia dei tipi generici.

Si assuma che `S` sia un sottotipo di `T`.

Esistono due importanti gerarchie generiche da ricordare.

Gerarchia 1 (wildcard con limite superiore):

`A<S>` è un sottotipo di `A<? extends S>` che a sua volta è un sottotipo di `A<? extends T>`

- Esempio:

Poiché `Integer` è un sottotipo di `Number`:

- `List<Integer> <<< List<? extends Integer>`
- `List<? extends Integer> <<< List<? extends Number>`

Pertanto, se un metodo sovrascritto restituisce:

`List<? extends Integer>`

il metodo che effettua l’override può restituire:

- `List<Integer>`

ma non può restituire:

- `List<Number>`
- `List<? extends Number>`

Gerarchia 2 (wildcard con limite inferiore):

`A<T>` è un sottotipo di `A<? super T>` che a sua volta è un sottotipo di `A<? super S>`

- Esempio:
- `List<Number> <<< List<? super Number>`
- `List<? super Number> <<< List<? super Integer>`

Pertanto, se un metodo sovrascritto restituisce:

`List<? super Number>`

il metodo che effettua l’override può restituire:

- `List<Number>`

ma non può restituire:

- `List<Integer>`

- `List<? super Integer>`

Un punto fondamentale da ricordare:

Anche se `Integer` è un sottotipo di `Number`,
`List<Integer>` **non** è un sottotipo di `List<Number>`.

I tipi generici in Java sono invarianti, salvo l'uso di wildcard.

Queste regole spiegano perché alcuni metodi che sembrano compatibili vengono rifiutati dal compilatore. La specificità generica deve rispettare le gerarchie formali di sottotipizzazione prima che la validazione dell'override passi ai controlli basati sull'erasure (vedi 18.4.7.10).

18.4.7.10 Regole riassuntive

- L'override è validato **dopo la erasure**
- La compatibilità è validata **prima della erasure**
- È possibile eliminare informazione generica nella sottoclasse
- Non è possibile aggiungere nuova specificità generica
- Se entrambi i metodi sono parametrizzati, gli argomenti devono coincidere esattamente
- Dopo la erasure, le firme devono essere identiche
- I tipi di ritorno covarianti richiedono che il tipo di ritorno del metodo che effettua l'override sia un vero sottotipo.
- Con i generics, le relazioni di sottotipizzazione devono rispettare le regole della gerarchia dei wildcard.
- Le relazioni logiche apparenti tra argomenti di tipo (ad esempio `Integer` e `Number`) non si traducono automaticamente in relazioni di sottotipizzazione tra tipi parametrizzati.

Questo spiega perché alcuni metodi che *sembrano* semplici overload vengono rifiutati: dopo la erasure entrano in collisione e, se non costituiscono un override valido, il compilatore li blocca.

18.4.8 Overloading di un Metodo Generico — Perché Alcuni Overload Sono Impossibili

Quando Java compila codice generico, applica la type erasure: i parametri di tipo come T vengono rimossi, e il compilatore li sostituisce con il loro tipo erasure (di solito Object o il primo bound).

Per questo motivo, due metodi che sembrano diversi a livello di sorgente possono diventare identici dopo l'erasure.

Se le `signature` con erasure sono uguali, Java non può distinguerli, quindi il codice non compila.

- Esempio: Due Metodi che Collassano sulla Stessa `Signature`

```
public class Demo {
    public void testInput(List<Object> inputParam) {}

    // public void testInput(List<String> inputParam) {}   // ❌ Errore di compilazione: dopo
}
```

Spiegazione

```
List<Object> e List<String> vengono entrambi cancellati a List.
```

A runtime entrambi i metodi apparirebbero come:

```
void testInput(List inputParam)
```

Java non permette due metodi con signature identiche nella stessa classe, quindi l'overload viene rifiutato a compile time.

18.4.9 Overloading di un Metodo Generico Ereditato da una Classe Parent

La stessa regola si applica quando una subclass tenta di introdurre un metodo che, dopo erasure, ha la stessa signature di uno nella superclass.

```java
public class SubDemo extends Demo {
    public void testInput(List<Integer> inputParam) {}
    // ❌ Errore di compilazione: erasure → testInput(List), uguale al parent
}
```

Ancora una volta, il compilatore rifiuta l'overload perché le signature con erasure collidono.

Quando l'Overloading Funziona

L'erasure rimuove solo i parametri generici, non la classe reale usata come parametro del metodo.

Quindi, se due parametri differiscono nel tipo raw (non generico), l'overload è legale.

```java
public class Demo {
    public void testInput(List<Object> inputParam) {}
    public void testInput(ArrayList<String> inputParam) {}  // ✓ Compila
}
```

Perché funziona

Anche se `ArrayList<String>` diventa `ArrayList`, e `List<Object>` diventa `List`, queste sono classi diverse (ArrayList vs List), quindi le signature restano distinte:

```java
void testInput(List inputParam)
void testInput(ArrayList inputParam)
```

Nessuna collisione → overloading legale.

18.4.10 Restituire Tipi Generici — Regole e Restrizioni

Quando si restituisce un valore da un metodo, Java segue una regola rigida:

Il tipo di ritorno di un metodo in overriding deve essere un sottotipo del tipo di ritorno del parent, e qualsiasi argomento generico deve rimanere type-compatible (anche se viene cancellato a runtime).

Questo spesso confonde i programmatori, perché i generics nei tipi di ritorno causano conflitti simili a quelli dei parametri.

Punti Chiave:

- La **covarianza del tipo di ritorno si applica solo al tipo raw**, non agli argomenti generici.
- Gli argomenti generici devono restare compatibili dopo l'erasure (devono coincidere).
- **Due metodi non possono differire solo per il parametro generico nel tipo di ritorno**.

Esempio: sostituzione Illegale del Tipo di Ritorno a Causa di Incompatibilità Generica

```java
class A {
    List<String> getData() { return null; }
}

class B extends A {
    // List<Integer> non è un tipo di ritorno covariante di List<String>
    // ❌ Errore di compilazione
    List<Integer> getData() { return null; }
}
```

Spiegazione:

Anche se i generics vengono cancellati, Java impone comunque type safety a livello di sorgente:

```
List<Integer> non è un sottotipo di List<String>.
```

Entrambi diventano List, ma Java rifiuta l'override che rompe la compatibilità di tipo.

- Esempio: Tipo di Ritorno Covariante Legale

```java
class A {
    Collection<String> getData() { return null; }
}

class B extends A {
    List<String> getData() { return null; }  // ✓ List è sottotipo di Collection
}
```

Questo è permesso perché:

- I tipi raw sono covarianti (List estende Collection).
- Gli argomenti generici coincidono (String vs String).
- Esempio: Overload Illegale Basato Solo sul Tipo di Ritorno

```java
class Demo {
    List<String> getList() { return null; }

    // List<Integer> getList() { return null; }
    // ❌ Errore di compilazione: il tipo di ritorno da solo non distingue i metodi
}
```

Java non usa il tipo di ritorno per distinguere metodi in overload.

18.4.11 Riepilogo delle Regole di Erasure

- `T senza bound` → erasure a Object.
- `T extends X` → erasure a X.
- `T extends X & Y & Z` → erasure a X.
- Tutti i parametri generici vengono cancellati nelle signature dei metodi.
- Vengono inseriti cast per preservare la tipizzazione a compile-time.
- Possono essere generati metodi bridge per preservare il polimorfismo.

18.5 Bound sui Parametri di Tipo

Questa sezione introduce i **vincoli sui parametri di tipo e i wildcard** nei generics di Java.
I vincoli limitano l'insieme dei tipi che possono essere utilizzati con un parametro di tipo generico o con un wildcard.

Sono utilizzati per imporre **vincoli di tipo** e per esprimere relazioni tra tipi nel codice generico.

I vincoli compaiono principalmente in due forme:

- **Vincoli sui parametri di tipo** usando `extends`
- **Vincoli sui wildcard** usando `?`, `? extends` e `? super`

Questi meccanismi permettono alle API generiche di specificare quali tipi sono accettabili e quali operazioni sono sicure dal punto di vista del sistema di tipi.

Regole

- `T extends Tipo` → il parametro di tipo deve essere `Tipo` o una sottoclasse.
- `T extends Classe & Interface1 & Interface2` → sono consentiti vincoli multipli.
- Nei vincoli multipli, la classe deve apparire per prima.
- `?` rappresenta un tipo sconosciuto.
- `? extends Tipo` → accetta tipi che sono `Tipo` o sottoclassi.
- `? super Tipo` → accetta tipi che sono `Tipo` o superclassi.
- `? extends` consente **lettura (estrazione)** ma proibisce l'inserimento.
- `? super` consente **scrittura (inserimento)** ma la lettura restituisce `Object`.

Tabella riassuntiva

Sintassi	Significato	Compatibilità di assegnazione	Lettura	Scrittura
`<T extends Number>`	Il parametro di tipo deve essere `Number` o una sottoclasse	Vincolo nella dichiarazione generica	`T`	`T`
`<T extends Classe & Interface>`	Vincoli multipli	Vincolo nella dichiarazione generica	`T`	`T`
`List<?>`	Tipo di elemento sconosciuto	Qualsiasi `List<T>`	`Object`	❌
`List<? extends Number>`	Sottotipo sconosciuto di `Number`	`List<Integer>`, `List<Double>`, ecc.	`Number`	❌
`List<? super Integer>`	`Integer` o supertipo	`List<Integer>`, `List<Number>`, `List<Object>`	`Object`	`Integer`

18.5.1 Upper Bounds: extends

`<T extends Number>` significa che **T deve essere Number o una sottoclasse**.

```
class Stats<T extends Number> {
    T num;
    Stats(T num) { this.num = num; }
}
```

18.5.2 Bound Multipli

Sintassi: `T extends Classe & Interface1 & Interface2 ...`

La classe deve apparire per prima.

```
class C<T extends Number & Comparable<T>> { }
```

18.5.3 Wildcard: `?`, `? extends`, `? super`

18.5.3.1 Wildcard Non Limitata `?`

Usare quando si vuole accettare una lista di tipo sconosciuto:

```
void printAll(List<?> list) { ... }
```

18.5.3.2 Wildcard con Upper Bound `? extends`

```
List<? extends Number> nums = List.of(1, 2, 3);
Number n = nums.get(0);    // OK
// nums.add(5);            // ❌ impossibile aggiungere: sicurezza di tipo
```

> **Non è possibile aggiungere elementi (eccetto null) a ? extends** perché non si conosce il sottotipo esatto.

18.5.3.3 Wildcard con Lower Bound `? super`

`<? super Integer>` significa che **il tipo deve essere Integer o una sua superclasse**.

```
List<? super Integer> list = new ArrayList<Number>();
list.add(10);     // OK
Object o = list.get(0); // restituisce Object (supertipo comune minimo)
```

> **Important**
>
> - `Super` accetta **inserimento**
> - `extends` accetta **estrazione**.

18.6 Generics ed Ereditarietà

> I generics NON partecipano all'ereditarietà.
> Un `List<String>` non è sottotipo di `List<Object>`; i tipi parametrizzati sono invarianti.

```
List<String> ls = new ArrayList<>();
List<Object> lo = ls;       // ❌ errore di compilazione
```

Invece:

```
List<? extends Object> ok = ls;    // funziona
```

18.7 Type Inference (Operatore Diamond)

```
Map<String, List<Integer>> map = new HashMap<>();
```

Il compilatore deduce gli argomenti generici dall'assegnazione.

18.8 Raw Types (Compatibilità Legacy)

Un **raw type** disabilita i generics, reintroducendo comportamenti non sicuri.

```
List raw = new ArrayList();
raw.add("x");
raw.add(10);    // permesso, ma non sicuro
```

> I raw types dovrebbero essere evitati.

18.9 Array Generici (Non Permessi)

Non puoi creare array di tipi parametrizzati:

```
List<String>[] arr = new List<String>[10];    // ❌ errore di compilazione
```

Perché gli array applicano type safety a runtime mentre i generics si basano solo su controlli a 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"); // ❌ non è un Number
```

18.11 Wildcard vs Parametri di Tipo

Usa le **wildcard** quando ti serve flessibilità nei parametri. Usa i **parametri di tipo** quando il metodo deve restituire o mantenere informazioni di tipo.

- Esempio — wildcard troppo debole:

```
List<?> copy(List<?> list) {
    return list;  // perde informazioni di tipo
}
```

Meglio:

```
<T> List<T> copy(List<T> list) {
    return list;
}
```

18.12 Regola PECS (Producer Extends, Consumer Super)

Usa **? extends** quando il parametro **produce** valori. Usa **? super** quando il parametro **consuma** valori.

```
List<? extends Number> listExtends = List.of(1, 2, 3);
List<? super Integer> listSuper = new ArrayList<Number>();

// ? extends → lettura sicura
Number n = listExtends.get(0);

// ? super → scrittura sicura
listSuper.add(10);
```

18.13 Errori Comuni

- Ordinare liste con wildcard: `List<? extends Number>` non può accettare inserimenti.
- Fraintendere che `List<Object>` NON è un supertype di `List<String>`.
- Dimenticare che gli array generici sono illegali.
- Pensare che i tipi generici siano preservati a runtime (vengono cancellati).
- Provare a fare overload di metodi usando solo parametri di tipo diversi.

18.14 Tabella Riassuntiva delle Wildcard

Sintassi	Significato
`?`	tipo sconosciuto (sola lettura eccetto metodi Object)
`? extends T`	leggere T in sicurezza, non si può aggiungere (eccetto null)
`? super T`	si può aggiungere T, la lettura restituisce Object

18.15 Riepilogo dei Concetti

```
Generics = type safety a compile-time
Bound = limitano i tipi legali
Wildcard = flessibilità nei parametri
Type Inference = il compilatore deduce i tipi
Type Erasure = i generics scompaiono a runtime
Bridge Methods = mantengono il polimorfismo
```

18.16 Esempio Completo

```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); }

    // metodo generico con wildcard
    static double sum(List<? extends Number> list) {
        double total = 0;
        for (Number n : list) total += n.doubleValue();
        return total;
    }
}
```

◀ 17. Oltre le Classi | ▲ Index | 19. Eccezioni e Gestione degli Errori ▶

19. Eccezioni e Gestione degli Errori

Indice

- [19.1 Gerarchia e tipi di eccezioni]
  - [19.1.1 Throwable]
  - [19.1.2 Error (unchecked)]
  - [19.1.3 Eccezioni Checked (`Exception`)]
  - [19.1.4 Eccezioni Unchecked (`RuntimeException`)]
- [19.2 Dichiarare e lanciare eccezioni]
  - [19.2.1 Dichiarare eccezioni con throws]
  - [19.2.2 Lanciare eccezioni]
- [19.3 Override dei metodi e regole sulle eccezioni]
- [19.4 Gestione delle eccezioni: try, catch, finally]
  - [19.4.1 Sintassi base try-catch]
  - [19.4.2 Blocchi catch multipli]
  - [19.4.3 Multi-catch Java 7]
  - [19.4.4 Blocco finally]
- [19.5 Gestione automatica delle risorse try-with-resources]
  - [19.5.1 Sintassi base]
  - [19.5.2 Dichiarare risorse multiple]
  - [19.5.3 Scope delle risorse]
- [19.6 Eccezioni soppresse]
- [19.7 Riepilogo sulle eccezioni]

Le `Exceptions` sono il meccanismo strutturato di Java per gestire condizioni anomale a runtime.

Permettono ai programmi di separare il flusso di esecuzione normale dalla logica di gestione degli errori, migliorando robustezza, leggibilità e correttezza.

19.1 Gerarchia e tipi di eccezioni

Tutte le eccezioni derivano da `Throwable`.

La gerarchia definisce quali condizioni sono recuperabili, quali devono essere dichiarate e quali rappresentano errori fatali del sistema.

```
java.lang.Object
└── java.lang.Throwable
    ├── java.lang.Error
    └── java.lang.Exception
        └── java.lang.RuntimeException
```

19.1.1 Throwable

- Classe base per tutti gli errori e le eccezioni
- Supporta messaggio, causa e stack trace
- Solo `Throwable` (e le sue sottoclassi) può essere lanciato o catturato

19.1.2 Error (unchecked)

- Rappresenta gravi problemi della JVM o del sistema
- Non è pensato per essere catturato o gestito

- Esempi: `OutOfMemoryError` , `StackOverflowError`

> **Note**
>
> - Gli Error indicano condizioni dalle quali l'applicazione generalmente non è attesa a riprendersi.

19.1.3 Eccezioni Checked (`Exception`)

- Sottoclassi di `Exception` **escludendo** `RuntimeException`
- Rappresentano condizioni che le applicazioni potrebbero voler gestire
- Devono essere **catturate** oppure **dichiarate**
- Esempi: `IOException` , `SQLException`

19.1.4 Eccezioni Unchecked (`RuntimeException`)

- Sottoclassi di `RuntimeException`
- Non è richiesto dichiararle o catturarle
- Solitamente rappresentano errori di programmazione
- Esempi: `NullPointerException` , `IllegalArgumentException`

19.2 Dichiarare e lanciare eccezioni

19.2.1 Dichiarare eccezioni con throws

Un metodo dichiara eccezioni **checked** usando la clausola `throws` . Questa fa parte del contratto API del metodo.

```
void readFile(Path p) throws IOException {
    Files.readString(p);
}
```

> **Note**
>
> - Solo le **eccezioni checked** devono essere dichiarate.
> - Le eccezioni unchecked possono essere dichiarate, ma di solito vengono omesse.

19.2.2 Lanciare eccezioni

Le eccezioni vengono create con `new` e lanciate esplicitamente usando `throw` .

```
if (value < 0) {
    throw new IllegalArgumentException("value must be >= 0");
}
```

- `throw` lancia esattamente un'istanza di eccezione
- `throws` dichiara le possibili eccezioni nella firma del metodo

19.3 Override dei metodi e regole sulle eccezioni

Quando si fa override di un metodo, le regole sulle eccezioni sono applicate in modo rigoroso.

- Un metodo in override può lanciare **meno** o **più specifiche** eccezioni checked
- Può lanciare qualsiasi eccezione unchecked
- Non può lanciare **nuove o più generiche** eccezioni checked

```java
class Parent {
    void work() throws IOException {}
}

class Child extends Parent {
    @Override
    void work() throws FileNotFoundException {} // OK (sottoclasse)
}
```

Note

- Cambiare solo le eccezioni **unchecked** non rompe mai il contratto di override.

Important

Ricorda: i `costruttori` seguono una regola diversa.

Un `Costruttore` deve dichiarare tutte le eccezioni checked dichiarate nel costruttore di base (o le superclassi di quelle eccezioni checked).

Può anche dichiarare eccezioni checked aggiuntive. Questo comportamento è l'opposto di quello dell'overriding dei metodi.

Un metodo che override non può lanciare alcuna eccezione checked diversa da quelle dichiarate dal metodo overridato. Può lanciare solo sottoclassi di tali eccezioni.

19.4 Gestione delle eccezioni: try, catch, finally

19.4.1 Sintassi base try-catch

```java
try {
    riskyOperation();
} catch (IOException e) {
    handle(e);
}
```

- Un blocco `try` deve essere seguito da almeno un `catch` oppure da un `finally`
- I catch vengono controllati dall'alto verso il basso

19.4.2 Blocchi catch multipli

Blocchi catch multipli permettono gestioni diverse per tipi di eccezione differenti.

```java
try {
    process();
} catch (FileNotFoundException e) {
    recover();
} catch (IOException e) {
    log();
}
```

Note

- Le eccezioni più specifiche devono venire prima di quelle più generali, altrimenti la compilazione fallisce.
- Se si mette un catch per una superclasse (es. `IOException`) prima di uno per una sottoclasse (es. `FileNotFoundException`), il catch della sottoclasse diventa irraggiungibile.

19.4.3 Multi-catch (Java 7+)

```
try {
    process();
} catch (IOException | SQLException e) {
    log(e);
}
```

- I tipi di eccezione devono essere non correlati (nessuna relazione parent/child)
- La variabile catturata è implicitamente `final`

19.4.4 Blocco finally

Il blocco `finally` viene eseguito indipendentemente dal fatto che venga lanciata un'eccezione, tranne in casi estremi di terminazione della JVM.

```
try {
    open();
} finally {
    close();
}
```

- Usato per logica di cleanup
- Viene eseguito anche se si usa `return` nel blocco try e/o catch

Note

- Un blocco `finally` può sovrascrivere un valore di ritorno o "inghiottire" un'eccezione. Questo è generalmente sconsigliato perché rende il flusso di controllo più difficile da comprendere.

Important

- Quando sia un blocco `catch` sia un blocco `finally` lanciano un'eccezione, l'eccezione lanciata nel blocco `finally` è quella che viene propagata dal metodo.
- L'eccezione lanciata nel blocco `catch` viene persa e **non** viene aggiunta alla lista delle eccezioni soppresse.

```
try {
    throw new RuntimeException("try");
} catch (RuntimeException e) {
    throw new RuntimeException("catch");
} finally {
    throw new RuntimeException("finally");
}
```

In questo caso, viene lanciata solo l'eccezione `"finally"`.

19.5 Gestione automatica delle risorse (try-with-resources)

Il `try-with-resources` fornisce la chiusura automatica delle risorse che implementano `AutoCloseable`.

Elimina la necessità di cleanup esplicito con `finally` nella maggior parte dei casi.

19.5.1 Sintassi base

```
try (BufferedReader br = Files.newBufferedReader(path)) {
    return br.readLine();
}
```

- Le risorse vengono chiuse automaticamente
- La chiusura avviene anche se viene lanciata un'eccezione
- Le risorse vengono chiuse prima dell'esecuzione dei blocchi `catch` o del `finally`

```
try (Resource a = new Resource()) {
    a.read();
} finally {
    a.close();  // ❌ Compile-time error: a è out of scope qui
}
```

19.5.2 Dichiarare risorse multiple

```
try (InputStream in = Files.newInputStream(p);
        OutputStream out = Files.newOutputStream(q)) {
    in.transferTo(out);
}
```

- Le risorse vengono chiuse in **ordine inverso** rispetto alla dichiarazione

19.5.3 Scope delle risorse

- Le risorse sono visibili solo all'interno del blocco `try`
- Sono implicitamente `final`
- Da Java 9, si possono dichiarare risorse in anticipo, fuori dal try-with-resources, purché siano dichiarate `final` o siano effettivamente final.

```
final var firstWriter = Files.newBufferedWriter(filePath);

try (firstWriter; var secondWriter = Files.newBufferedWriter(filePath)) {
    // CODE
}
```

Note

- Tentare di riassegnare una variabile risorsa causa un errore di compilazione.

```
Resource a = new Resource();
try(a){ // since Java 9
  ...
}finally{
   a.close(); // questo codice compila ma la risorsa puntata dal reference 'a', è stata chiusa
}
```

19.6 Eccezioni soppresse

Quando sia il blocco `try` sia il metodo `close()` della risorsa lanciano eccezioni, Java conserva l'eccezione principale e **sopprime** le altre.

```
try (BadResource r = new BadResource()) {
    throw new RuntimeException("main");
}
```

Se anche `close()` lancia un'eccezione, questa diventa **soppressa**.

```java
catch (Exception e) {
    for (Throwable t : e.getSuppressed()) {
        System.out.println(t);
    }
}
```

- L'eccezione principale viene lanciata
- Le eccezioni secondarie sono accessibili tramite `getSuppressed()`

Important

- Le eccezioni soppresse vengono generate solo dal blocco `finally` **implicito** creato dal `try-with-resources`.
- Al contrario, le eccezioni lanciate in un blocco `finally` **esplicito** non vengono soppresse: sostituiscono qualsiasi eccezione precedente e diventano l'unica eccezione propagata.

19.7 Riepilogo sulle eccezioni

- Le eccezioni checked devono essere catturate o dichiarate
- I metodi in override non possono ampliare le eccezioni checked
- Usa il multi-catch per logica di gestione condivisa
- Preferisci try-with-resources al cleanup con finally
- Le risorse si chiudono in ordine inverso
- Le eccezioni soppresse preservano il contesto completo del fallimento

---

◀ 18. Generics in Java | ▲ Index | 20. Programmazione Funzionale in Java ▶

Module 05

Functional Programming

20. Programmazione Funzionale in Java

Indice

- [20.1 Interfacce Funzionali]
  - [20.1.1 Regole per le Interfacce Funzionali]
  - [20.1.2 Interfacce Funzionali Comuni (java.util.function)]
  - [20.1.3 Metodi di Comodità nelle Interfacce Funzionali]
  - [20.1.4 Interfacce Funzionali Primitive]
  - [20.1.5 Riepilogo]
- [20.2 Espressioni Lambda]
  - [20.2.1 Sintassi delle Espressioni Lambda]
  - [20.2.2 Esempi di Sintassi Lambda]
  - [20.2.3 Regole per le Espressioni Lambda]
  - [20.2.4 Inferenza di Tipo]
  - [20.2.5 Restrizioni nei Corpi delle Lambda]
  - [20.2.6 Regole sul Tipo di Ritorno]
  - [20.2.7 Lambda vs Classi Anonime]
  - [20.2.8 Errori Comuni nelle Lambda]
- [20.3 Riferimenti a Metodi]
  - [20.3.1 Riferimento a un Metodo Statico]
  - [20.3.2 Riferimento a un Metodo d'Istanza di un Oggetto Specifico]
  - [20.3.3 Riferimento a un Metodo d'Istanza di un Oggetto Arbitrario di un Dato Tipo]
  - [20.3.4 Riferimento a un Costruttore]
  - [20.3.5 Tabella Riassuntiva dei Tipi di Method Reference]
  - [20.3.6 Errori Comuni]

La `programmazione funzionale` è un paradigma di programmazione che si concentra sul descrivere **cosa** deve essere fatto, piuttosto che **come** deve essere fatto.

A partire da Java 8, il linguaggio ha aggiunto diverse funzionalità che abilitano uno stile di programmazione "funzionale": `lambda expressions`, `functional interfaces` e `method references`.

Queste funzionalità permettono agli sviluppatori di scrivere codice più espressivo, conciso e riutilizzabile, soprattutto quando si lavora con collezioni, API di concorrenza e sistemi event-driven.

20.1 Interfacce Funzionali

In Java, un'**interfaccia funzionale** è un'interfaccia che contiene **esattamente un solo** metodo astratto.

Le interfacce funzionali abilitano **Lambda Expressions** e **Method References**, formando il nucleo del modello di programmazione funzionale di Java.

> **Note**
>
> Java tratta automaticamente come interfaccia funzionale qualsiasi interfaccia con un solo metodo astratto. L'annotazione `@FunctionalInterface` è opzionale ma consigliata.

20.1.1 Regole per le Interfacce Funzionali

- **Esattamente un metodo astratto** (SAM = Single Abstract Method).

- Le interfacce possono dichiarare un numero qualsiasi di metodi **default**, **static** o **private**.
- Possono fare override dei metodi di `Object` (`toString()`, `equals(Object)`, `hashCode()`) senza influenzare il conteggio SAM.
- Il metodo funzionale può provenire da una **superinterfaccia**.

Esempio:

```
@FunctionalInterface
interface Adder {
    int add(int a, int b);   // single abstract method
    static void info() {}
    default void log() {}
}
```

20.1.2 Interfacce Funzionali Comuni (java.util.function)

Di seguito un riepilogo delle interfacce funzionali più importanti.

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 (stessi tipi)
`BinaryOperator<T>`	T	apply(T,T)	2 (stessi tipi)
`Predicate<T>`	boolean	test(T)	1
`BiPredicate<T,U>`	boolean	test(T,U)	2

- Esempi

```
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 Metodi di Comodità nelle Interfacce Funzionali

Molte interfacce funzionali includono metodi di supporto che consentono chaining e composizione.

Interface	Method	Description
Function	andThen()	applica la funzione, poi l'altra specificata in questo metodo addizionale
Function	compose()	applica la funzione specificata nel metodo addizionale, poi la funzione
Function	identity()	restituisce una funzione x -> x
Predicate	and()	AND logico
Predicate	or()	OR logico
Predicate	negate()	NOT logico
Consumer	andThen()	concatena consumer
BinaryOperator	minBy()	minimo basato su comparator
BinaryOperator	maxBy()	massimo basato su comparator

- Esempi

```
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 Interfacce Funzionali Primitive

Java fornisce versioni specializzate delle interfacce funzionali per i tipi primitivi, per evitare overhead di 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

- Esempio

```
IntSupplier dice = () -> (int)(Math.random() * 6) + 1;

IntPredicate even = x -> x % 2 == 0;

IntUnaryOperator doubleIt = x -> x * 2;
```

20.1.5 Riepilogo

- Le interfacce funzionali contengono esattamente un metodo astratto (SAM).
- Sono alla base di Lambda e Method References.
- Java offre molte FI built-in in `java.util.function`.
- Le varianti primitive migliorano le performance rimuovendo il boxing.

20.2 Espressioni Lambda

Una lambda expression è un modo compatto di scrivere una funzione.

Le lambda expressions offrono un modo conciso per definire implementazioni di interfacce funzionali.

Una lambda è essenzialmente un piccolo blocco di codice che accetta parametri e restituisce un valore, senza richiedere una dichiarazione completa di metodo.

Rappresentano il comportamento come dato e sono un elemento chiave del modello di programmazione funzionale in Java.

20.2.1 Sintassi delle Espressioni Lambda

La sintassi generale è:

`(parameters) -> expression`
oppure
`(parameters) -> { statements }`

> **Important**
>
> Un'espressione lambda non introduce un nuovo scope per le variabili. Di conseguenza, i nomi di variabili già esistenti nel contesto circostante non possono essere ridefiniti come parametri dell'espressione lambda.

20.2.2 Esempi di Sintassi Lambda

Zero parametri

```
Runnable r = () -> System.out.println("Hello");
```

Un parametro (parentesi opzionali)

```
Consumer<String> c = s -> System.out.println(s);
```

Più parametri

```
BinaryOperator<Integer> add = (a, b) -> a + b;
```

Con block body

```
Function<Integer, String> f = (x) -> {
    int doubled = x * 2;
    return "Value: " + doubled;
};
```

20.2.3 Regole per le Espressioni Lambda

- I tipi dei parametri possono essere omessi (type inference).
- Se un parametro ha un tipo, allora **tutti** i parametri devono specificare il tipo.
- Un singolo parametro non richiede parentesi.
- Più parametri richiedono le parentesi.
- Se il corpo è una singola espressione (senza `{ }`), `return` non è consentito; l'espressione stessa è il valore di ritorno.
- Se il corpo usa `{ }` (un blocco), `return` deve comparire se viene restituito un valore.
- Le lambda possono essere assegnate solo a interfacce funzionali (tipi SAM).

20.2.4 Inferenza di Tipo

Il compilatore deduce il tipo della lambda dal contesto dell'interfaccia funzionale target.

```
Predicate<String> p = s -> s.isEmpty();  // s inferito come String
```

Se il compilatore non riesce a inferire il tipo, devi specificarlo esplicitamente.

```
BiFunction<Integer, Integer, Integer> f = (Integer a, Integer b) -> a * b;
```

20.2.5 Restrizioni nei Corpi delle Lambda

Le lambda possono catturare solo variabili locali che sono final o effectively final (non riassegnate).

```
int x = 10;
Runnable r = () -> {
    // x++;   // ❌ errore di compilazione — x deve essere effectively final
    System.out.println(x);
};
```

Possono invece modificare lo stato di un oggetto (solo i riferimenti devono essere effectively final).

```
var list = new ArrayList<>();
Runnable r2 = () -> list.add("OK");  // consentito
```

20.2.6 Regole sul Tipo di Ritorno

Se il corpo è un'espressione: l'espressione è il valore di ritorno.

```
Function<Integer, Integer> f = x -> x * 2;
```

Se il corpo è un blocco: devi includere `return`.

```
Function<Integer, Integer> g = x -> {
    return x * 2;
};
```

20.2.7 Lambda vs Classi Anonime

- Le lambda NON creano un nuovo scope: condividono lo scope contenitore.
- `this` dentro una lambda si riferisce all'oggetto contenitore, non alla lambda.

```
class Test {
    void run() {
        Runnable r = () -> System.out.println(this.toString());
    }
}
```

Nelle classi anonime, `this` si riferisce all'istanza della classe anonima.

20.2.8 Errori Comuni nelle Lambda

Tipi di ritorno incoerenti

```
x -> { if (x > 0) return 1; }  // ❌ manca return per il caso negativo
```

Mescolare parametri tipizzati e non tipizzati

```
(a, int b) -> a + b   // ❌ illegale
```

Restituire un valore da una lambda con target void

```
Runnable r = () -> 5;  // ❌ Runnable.run() restituisce void
```

Risoluzione di overload ambigua

```
void m(IntFunction<Integer> f) {}
void m(Function<Integer, Integer> f) {}

m(x -> x + 1);  // ❌ ambiguo
```

20.3 Riferimenti a Metodi

I riferimenti a metodi (method references) forniscono una sintassi abbreviata per usare un metodo esistente come implementazione di un'interfaccia funzionale.

Sono equivalenti alle lambda expressions, ma più concisi, leggibili e spesso preferibili quando il metodo target esiste già.

Esistono quattro categorie di method references in Java:

- Riferimento a un metodo statico (`ClassName::staticMethod`)
- Riferimento a un metodo d'istanza di un oggetto specifico (`instance::method`)
- Riferimento a un metodo d'istanza di un oggetto arbitrario di un dato tipo (`ClassName::instanceMethod`)
- Riferimento a un costruttore (`ClassName::new`)

20.3.1 Riferimento a un Metodo Statico

Un method reference statico sostituisce una lambda che invoca un metodo statico.

```
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
```

Sia `f1` che `f2` si comportano in modo identico.

20.3.2 Riferimento a un Metodo d'Istanza di un Oggetto Specifico

Usato quando hai già un'istanza di un oggetto e vuoi riferirti a uno dei suoi metodi.

```
String prefix = "Hello, ";

UnaryOperator<String> op1 = s -> prefix.concat(s);
UnaryOperator<String> op2 = prefix::concat;   // method reference

System.out.println(op2.apply("World"));
```

Il riferimento `prefix::concat` lega `concat` a **quell'oggetto specifico**.

20.3.3 Riferimento a un Metodo d'Istanza di un Oggetto Arbitrario di un Dato Tipo

Questa è la forma più "insidiosa".

Il primo parametro dell'interfaccia funzionale diventa il receiver del metodo (`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**
>
> Questa forma applica il metodo al *primo argomento* della lambda.

20.3.4 Riferimento a un Costruttore

I constructor references sostituiscono lambda che invocano `new`.

```
Supplier<ArrayList<String>> sup1 = () -> new ArrayList<>();
Supplier<ArrayList<String>> sup2 = ArrayList::new; // method reference

Function<Integer, ArrayList<String>> sup3 = ArrayList::new;
// invoca il costruttore ArrayList(int capacity)
```

20.3.5 Tabella Riassuntiva dei Tipi di Method Reference

La tabella seguente riassume tutte le categorie di method reference.

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 Errori Comuni

- Un method reference deve combaciare *esattamente* con la signature dell'interfaccia funzionale.
- Gli overload possono rendere i method references ambigui.
- Il riferimento a metodo d'istanza (`Class::method`) sposta il receiver al parametro 1.
- Un constructor reference fallisce se non esiste un costruttore compatibile.

```
// ❌ Ambiguo: quale println()? (println(int), println(String)...)
Consumer<String> c = System.out::println; // OK solo perché il parametro FI è String

// ❌ Costruttore non compatibile: interfaccia funzionale errata
Supplier<Integer> s = Integer::new;         // ✓ OK: invoca Integer()
Function<String, Long> f = Integer::new;    // ❌ ERRORE: il costruttore restituisce Integer,
```

In caso di dubbio, riscrivi il method reference come una lambda: se la lambda funziona ma il method reference no, il problema è quasi sempre il matching della signature.

---

21. Java Optional e Streams

Indice

- 21.1 Optional (Optional OptionalInt OptionalLong OptionalDouble)
  - 21.1.1 Creare Optional
  - 21.1.2 Leggere valori in sicurezza
  - 21.1.3 Trasformare Optional
  - 21.1.4 Optional e Stream
  - 21.1.5 Optional per tipi primitivi
  - 21.1.6 Trappole comuni
- 21.2 Che Cos’è uno Stream (E cosa Non è)
- 21.3 Architettura della Pipeline Stream
  - 21.3.1 Sorgenti di Stream
  - 21.3.2 Operazioni Intermedie
  - 21.3.2.1 Tabella delle operazioni intermedie comuni
  - 21.3.3 Operazioni Terminali
    - 21.3.3.1 Tabella delle operazioni terminali
- 21.4 Valutazione Pigra e Short-Circuiting
- 21.5 Operazioni Stateless vs Stateful
  - 21.5.1 Operazioni Stateless
  - 21.5.2 Operazioni Stateful
- 21.6 Ordinamento degli Stream e Determinismo
- 21.7 Stream Paralleli
- 21.8 Operazioni di Riduzione
  - 21.8.1 `reduce()` : combinare uno stream in un singolo oggetto
    - 21.8.1.1 Modello mentale corretto
  - 21.8.2 `collect()`
  - 21.8.3 Perché `collect()` è diverso da `reduce()`
- 21.9 Trappole Comuni degli Stream
- 21.10 Stream Primitivi
  - 21.10.1 Perché gli stream primitivi sono importanti
  - 21.10.2 Metodi comuni di creazione
  - 21.10.3 Metodi di mapping specializzati per primitivi
  - 21.10.4 Tabella di mapping tra `Stream<T>` e stream primitivi
  - 21.10.5 Operazioni terminali e i loro tipi di risultato
  - 21.10.6 Trappole e gotcha comuni
- 21.11 Collector (collect(), Collector e i Metodi Factory di Collectors)
  - 21.11.1 collect() vs Collector
  - 21.11.2 Collector core
  - 21.11.3 Collector di raggruppamento
  - 21.11.4 partitioningBy
  - 21.11.5 toMap e regole di merge
  - 21.11.6 collectingAndThen
  - 21.11.7 Come i collector si relazionano agli stream paralleli

---

21.1 Optional (Optional, OptionalInt, OptionalLong, OptionalDouble)

`Optional<T>` è un oggetto contenitore che può contenere, o meno, un valore non-null.

È stato pensato per rendere esplicita l'"assenza di un valore" e per ridurre il rischio di `NullPointerException` forzando i chiamanti a gestire il caso di "assenza".

> **Note**
>
> - `Optional` è inteso principalmente per **tipi di ritorno**.
> - È generalmente sconsigliato per attributi, parametri di metodo e contesti di serializzazione (a meno che un contratto API specifico lo richieda).

21.1.1 Creare Optional

Ci sono tre metodi factory principali per creare Optional.

- `Optional.of(value)` → value deve essere non-null; altrimenti viene lanciata `NullPointerException`
- `Optional.ofNullable(value)` → restituisce empty se value è null
- `Optional.empty()` → un Optional esplicitamente vuoto

```
Optional<String> a = Optional.of("x");
Optional<String> b = Optional.ofNullable(null); // Optional.empty
Optional<String> c = Optional.empty();
```

21.1.2 Leggere valori in sicurezza

Gli Optional forniscono molteplici modi per accedere al valore incapsulato.

- `isPresent()` / `isEmpty()` → test di presenza
- `get()` → restituisce il valore o lancia `NoSuchElementException` se non presente (sconsigliato)
- `orElse(defaultValue)` → restituisce valore o default (default valutato immediatamente)
- `orElseGet(supplier)` → restituisce valore o risultato del supplier (supplier valutato lazily)
- `orElseThrow()` → restituisce valore o lancia `NoSuchElementException`
- `orElseThrow(exceptionSupplier)` → restituisce valore o lancia eccezione personalizzata

```
Optional<String> opt = Optional.of("java");

String v1 = opt.orElse("default");
String v2 = opt.orElseGet(() -> "computed");
String v3 = opt.orElseThrow(); // ok perché opt è presente
```

> **Note**
>
> - Una trappola comune: `orElse(...)` valuta il suo argomento anche se l'Optional è presente.
> - Usa `orElseGet(...)` quando il default è laborioso da calcolare.

21.1.3 Trasformare Optional

Gli Optional supportano trasformazioni funzionali simili agli stream, ma con semantica "0 o 1 elemento".

- `map(fn)` → trasforma il valore se presente
- `flatMap(fn)` → trasforma in un Optional "flattened", senza annidamento
- `filter(predicate)` → mantiene il valore solo se il predicate è true

```
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"));
```

Output:

```
5
Alice
```

Note

- `map` incapsula il risultato in un Optional.
- Se la tua funzione di mapping restituisce già un Optional, usa `flatMap` per evitare l'annidamento `Optional<Optional<T>>`.

21.1.4 Optional e Stream

Un pattern di pipeline molto comune è fare `map` verso un Optional e poi rimuovere gli "assenti".

Da Java 9, `Optional` fornisce `stream()` per convertire "presente → un elemento" e "vuoto → zero elementi".

```
        Stream<String> words = Stream.of("a", "bb", "ccc");

        words.map(w -> w.length() > 1 ? Optional.of(w.length()) : Optional.<Integer>empty()).f
                .forEach(System.out::println);
```

Output:

```
2
3
```

Note

Prima di Java 9, questo pattern richiedeva `filter(Optional::isPresent)` più `map(Optional::get)`.

21.1.5 Optional per tipi primitivi

Gli stream primitivi usano optional primitivi per evitare boxing: `OptionalInt`, `OptionalLong`, `OptionalDouble`.

Essi rispecchiano la API principale di Optional con `getter` primitivi come `getAsInt()`.

- `OptionalInt.getAsInt()` / `OptionalLong.getAsLong()` / `OptionalDouble.getAsDouble()`
- `orElse(...)` / `orElseGet(...)` / `orElseThrow(...)`

```
OptionalInt m = IntStream.of(3, 1, 2).min(); // OptionalInt[1]
int value = m.orElse(0); // 1
```

21.1.6 Trappole comuni

- Non usare `get()` senza controllare la presenza; si preferisca `orElseThrow` o trasformazioni

- Evita di restituire `null` invece di `Optional.empty()`; un riferimento Optional in sé non dovrebbe essere null
- Ricorda: `average()` sugli stream primitivi restituisce sempre `OptionalDouble` (anche per `IntStream` e `LongStream`)
- Usa `orElseGet` quando calcolare il default è costoso in termini di calcolo (Performances)

21.2 Che Cos'è uno Stream (E cosa non è)

Uno `Stream Java` rappresenta una sequenza di elementi (una pipeline) che supporta operazioni in stile funzionale.

Gli stream sono progettati per l'elaborazione dei dati, non per l'archiviazione degli stessi.

Caratteristiche chiave:

- Uno stream non memorizza dati
- Uno stream è Lazy — non succede nulla su di esso, finché non viene invocata un'operazione terminale
- Uno stream può essere consumato soltanto una volta
- Gli stream incoraggiano operazioni senza effetti collaterali

> **Note**
>
> Gli stream sono concettualmente simili a query di database: descrivono cosa calcolare, non come iterare.

21.3 Architettura della Pipeline Stream

Ogni pipeline di stream consiste di tre fasi distinte:

- 1 **Sorgente**
- 2 Zero o più **Operazioni Intermedie**
- 3 Esattamente una **Operazione Terminale**

21.3.1 Sorgenti di Stream

Sorgenti comuni di stream includono:

- Collezioni: `collection.stream()`
- Array: `Arrays.stream(array)`
- Canali I/O e file
- Stream infiniti: `Stream.iterate`, `Stream.generate`

```
List<String> names = List.of("Ana", "Bob", "Carla");

Stream<String> s = names.stream();
```

21.3.2 Operazioni Intermedie

Operazioni intermedie:

- Restituiscono un nuovo stream
- Sono evalualte "Lazily"
- Non innescano l'esecuzione

21.3.2.1 Tabella delle operazioni intermedie comuni:

Method	Common input Params	Return value	Desctiption
`filter`	Predicate	`Stream<T>`	filtra lo stream secondo una corrispondenza del predicate
`map`	Function	`Stream<R>`	trasforma uno stream attraverso un mapping uno a uno input/output
`flatMap`	Function	`Stream<R>`	appiattisce stream annidati in un singolo stream
`sorted`	(none) or Comparator	`Stream<T>`	ordina per ordine naturale o per il Comparator fornito
`distinct`	(none)	`Stream<T>`	rimuove elementi duplicati
`limit` / `skip`	long	`Stream<T>`	limita la dimensione o salta elementi
`peek`	Consumer	`Stream<T>`	esegue un'azione con side-effect per ogni elemento (debugging)

- Esempio:

```
List<String> names = List.of("Ana", "Bob", "Carla", "Mario");

names.stream().filter(n -> n.length() > 3).map(String::toUpperCase).forEach(System.out
```

Output:

```
CARLA
MARIO
```

> **Note**
>
> Le operazioni intermedie descrivono solo il calcolo. Nessun elemento è ancora elaborato.

21.3.3 Operazioni Terminali

Operazioni terminali:

- Innescano l'esecuzione
- Consumano lo stream
- Producono un risultato o un effetto collaterale

21.3.3.1 Tabella delle operazioni terminali:

Method	Return value	behaviour for infinite streams
`forEach`	**void**	non termina
`collect`	varia	non termina
`reduce`	varia	non termina
`findFirst` / `findAny`	`Optional<T>`	termina
`anyMatch` / `allMatch` / `noneMatch`	**boolean**	può terminare presto (short-circuit)
`min` / `max`	`Optional<T>`	non termina
`count`	**long**	non termina

21.4 Valutazione Pigra e Short-Circuiting

```
var newNames = new ArrayList<String>();

newNames.add("Bob");
newNames.add("Dan");

// Gli stream sono valutati pigramente: questo non attraversa ancora i dati,
// crea solo una descrizione della pipeline legata alla sorgente.
var stream = newNames.stream();

newNames.add("Erin");

// L'operazione terminale innesca la valutazione. Lo stream vede la sorgente aggiornata,
// quindi il count include "Erin".
stream.count(); // 3
```

> **Note**
>
> Uno stream è legato alla sua *sorgente* (`newNames`), e la pipeline non viene eseguita finché non viene invocata un'operazione terminale.
> Per questa ragione, se **modifichi la collezione prima dell'operazione terminale**, l'operazione terminale "vede" i nuovi elementi (qui, `Erin`).
> In generale, tuttavia, **modificare la sorgente mentre una pipeline stream è in uso è una cattiva pratica** e può portare a comportamento non deterministico (o `ConcurrentModificationException` con alcune sorgenti/operazioni). La regola pratica è: *costruisci la sorgente, poi crea ed esegui lo stream senza mutarla*.

Gli stream processano elementi **uno alla volta**, scorrendo "verticalmente" attraverso la pipeline piuttosto che stadio-per-stadio.

Sotto modifichiamo l'esempio per usare un'operazione terminale **short-circuiting**: `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);
```

Ordine di esecuzione:

> **Note**
>
> Viene processato solo il numero minimo di elementi richiesti dall'operazione terminale.

```
filter a
filter bb
map bb
BB
```

`findFirst()` è soddisfatto non appena trova il **primo** elemento che passa con successo attraverso la pipeline (qui `"bb"`), quindi:

- `"ccc"` non viene mai processato (né `filter` né `map`);
- la valutazione pigra evita lavoro non necessario rispetto a un'operazione terminale che consuma tutti gli elementi (come `forEach` o `count`).

> **Important**
>
> `allMatch`, `noneMatch`, `anyMatch`, `findFirst` e `findAny` sono **operazioni terminali a corto circuito** (*short-circuiting terminal operations*).
>
> Ciò significa che il predicato fornito **non viene necessariamente valutato per ogni elemento dello stream**.
> L'operazione può interrompersi non appena il risultato finale può essere determinato.
>
> Per esempio, con `allMatch`, se il predicato restituisce `false` per il **primo elemento**, il risultato complessivo dell'operazione è già noto come `false`. Poiché `allMatch` richiede che tutti gli elementi soddisfino il predicato, trovare un singolo elemento che non lo soddisfa è sufficiente per determinare il risultato.
>
> Di conseguenza, quando un tale elemento viene incontrato, **gli elementi rimanenti dello stream non devono essere testati**, e l'elaborazione dello stream termina immediatamente.

21.5 Operazioni Stateless vs Stateful

21.5.1 Operazioni Stateless

Operazioni come `map` e `filter` processano ogni elemento indipendentemente.

21.5.2 Operazioni Stateful

Operazioni come `distinct`, `sorted` e `limit` richiedono il mantenimento di stato interno.

> **Note**
>
> Le operazioni stateful possono impattare severamente le prestazioni degli stream paralleli.

21.6 Ordinamento degli Stream e Determinismo

Gli stream possono essere:

- Ordinati (es. `List.stream()`)
- Non ordinati (es. `HashSet.stream()`)

Alcune operazioni rispettano l'ordine di encounter:

- `forEachOrdered`

- `findFirst`

> **Note**
>
> Negli stream paralleli, `forEach` non garantisce ordine.

21.7 Stream Paralleli

Gli stream paralleli dividono il lavoro tra thread usando il ForkJoinPool.commonPool().

```
int sum =
IntStream.range(1, 1_000_000)
    .parallel()
        .sum();
```

Regole per stream paralleli sicuri:

- Nessun effetto collaterale
- Nessuno stato condiviso mutabile
- Solo operazioni associative

> **Note**
>
> Gli stream paralleli possono essere più lenti per carichi di lavoro leggeri.

21.8 Operazioni di Riduzione

21.8.1 `reduce()` : combinare uno stream in un singolo oggetto

Ci sono tre firme di metodo per questa operazione:

- 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 riduzione richiede:

- **Identity**: valore iniziale per ogni riduzione parziale; deve essere un elemento neutro; Esempio: 0 per la somma, 1 per la moltiplicazione, collezione vuota per la raccolta;
- **Accumulator**: incorpora un elemento dello stream in un risultato parziale;
- (Opzionale) **Combiner**: unisce due risultati parziali; Usato solo quando lo stream è parallelo; Ignorato per stream sequenziali

> **Note**
>
> L'accumulator deve essere associativo e stateless.

21.8.1.1 Modello mentale corretto

- Accumulator: risultato + elemento
- Combiner: risultato + risultato

Esempio 1: Uso corretto (somma delle lunghezze)

```
int totalLength =
    Stream.of("a", "bb", "ccc")
          .parallel()
          .reduce(
              0,                          // identity
              (sum, s) -> sum + s.length(), // accumulator
              (left, right) -> left + right // combiner
          );
```

Cosa succede in parallelo

Supponi che lo stream sia diviso:

- Thread 1: "a", "bb" → 0 + 1 + 2 = 3
- Thread 2: "ccc" → 0 + 3 = 3

in seguito il combiner unisce i risultati parziali:

```
3 + 3 = 6
```

Esempio 2: Combiner ignorato negli stream sequenziali

```
int result =
    Stream.of("a", "bb", "ccc")
          .reduce(
              0,
              (sum, s) -> sum + s.length(),
              (x, y) -> {
                  throw new RuntimeException("Never called");
              }
          );
```

Esempio 3: Combiner scorretto

```
int result =
    Stream.of(1, 2, 3, 4)
          .parallel()
          .reduce(
              0,
              (a, b) -> a - b,    // accumulator
              (x, y) -> x - y     // combiner
          );
```

Perché questo è sbagliato

La sottrazione non è associativa.

Possibile esecuzione:

- Thread 1: 0 - 1 - 2 = -3
- Thread 2: 0 - 3 - 4 = -7

Combiner:

```
-3 - (-7) = 4
```

Il risultato sequenziale sarebbe:

```
(((0 - 1) - 2) - 3) - 4 = -10
```

> **Warning**
>
> ❌ I risultati paralleli e sequenziali differiscono → riduzione illegale

21.8.2 `collect()`

`collect` è una riduzione mutabile ottimizzata per raggruppamento e aggregazione.

È lo strumento standard della Stream API per la “riduzione mutabile”: accumuli elementi in un contenitore mutabile (come una List, Set, Map, StringBuilder, oggetto risultato custom), e poi, opzionalmente, si uniscono i contenitori parziali quando si esegue in parallelo.

La forma generale è:

- `public <R> R **collect**(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)`;

E una versione comune usata è:

- `public <R, A> R **collect**(Collector<? super T, A, R> collector)`

dove Collectors.* fornisce collector pre-costruiti (grouping, mapping, joining, counting, ecc.).

Significato:

- **supplier**: crea un nuovo contenitore risultato vuoto (es. `new ArrayList<>()`)
- **accumulator**: aggiunge un elemento in quel contenitore (es. `list::add`)
- **combiner**: unisce due contenitori (es. `list1.addAll(list2)`)

21.8.3 Perché `collect()` è diverso da `reduce()`

- **Intento**: mutazione vs immutabilità
  - `reduce()` è progettato per riduzione in stile immutabile: combinare valori in un nuovo valore (es. somma, min, max).
  - `collect()` è progettato per contenitori mutabili: costruire una List, Map, StringBuilder, ecc.
- **Correttezza** in parallelo
  - `reduce()` richiede che l’operazione sia:
    - associativa
    - stateless
    - compatibile con regole di identity/combiner
  - `collect()` è costruito per supportare il parallelismo in sicurezza mediante:
    - creazione di un contenitore per thread (supplier)
    - accumulo locale (accumulator)
    - merge alla fine (combiner)
- **Prestazioni**
  - `collect()` può essere ottimizzato perché il runtime dello stream sa che stai costruendo contenitori:
    - può evitare copie non necessarie
    - può pre-dimensionare o usare implementazioni specializzate (a seconda del collector)
    - è l’approccio idiomatico e atteso
    - usare reduce() per costruire una collezione spesso crea oggetti extra o forza mutazione non sicura.
- Esempio: “collect in una List” nel modo corretto

```
List<String> longNames =
    names.stream()
         .filter(s -> s.length() > 3)
            .collect(Collectors.toList());
```

- Esempio: groupingBy con spiegazione

```
Map<Integer, List<String>> byLength =
    names.stream()
        .collect(Collectors.groupingBy(String::length));
```

Cosa succede concettualmente:

- Il collector crea una `Map<Integer, List<String>>` vuota
- Per ogni name:
  - calcola la chiave (String::length)
  - lo mette nella lista bucket corretta
- In parallelo:
  - ogni thread costruisce le proprie mappe parziali
  - il combiner unisce le mappe unendo le liste per chiave

21.9 Trappole comuni degli Stream

- Riutilizzare uno stream consumato → `IllegalStateException`
- Modificare variabili esterne dentro le lambda
- Assumere ordine di esecuzione negli stream paralleli
- Usare `peek` per logica invece che per debugging

21.10 Stream Primitivi

Java fornisce tre tipi di stream specializzati per evitare overhead di boxing e per abilitare operazioni focalizzate sui numeri:

- `IntStream` per `int`
- `LongStream` per `long`
- `DoubleStream` per `double`

Gli stream primitivi sono comunque stream (pipeline lazy, operazioni intermedie + terminali, single-use), ma non sono **generici** e usano interfacce funzionali specializzate per primitivi (es. `IntPredicate`, `LongUnaryOperator`, `DoubleConsumer`).

> **Note**
>
> Usa stream primitivi quando i dati sono naturalmente numerici o quando le prestazioni contano: evitano overhead di boxing/unboxing e forniscono operazioni terminali numeriche aggiuntive.

21.10.1 Perché gli stream primitivi sono importanti

- Prestazioni: evitare l'allocazione di oggetti wrapper e boxing/unboxing ripetuti in pipeline grandi
- Convenienza: riduzioni numeriche integrate come `sum()`, `average()`, `summaryStatistics()`
- Trappole comuni: capire quando i risultati sono primitivi vs `OptionalInt` / `OptionalLong` / `OptionalDouble`

21.10.2 Metodi comuni di creazione

I seguenti sono i modi usati più frequentemente per creare stream primitivi. Molte domande di certificazione iniziano identificando il tipo di stream creato da un metodo factory.

Sources
IntStream.of(int...)
IntStream.range(int startInclusive, int endExclusive)
IntStream.rangeClosed(int startInclusive, int endInclusive)
IntStream.iterate(int seed, IntUnaryOperator f) // infinito a meno che limitato
IntStream.iterate(int seed, IntPredicate hasNext, IntUnaryOperator f)
IntStream.generate(IntSupplier s) // infinito a meno che limitato
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

- Solo `IntStream` e `LongStream` forniscono `range()` e `rangeClosed()`.
- Non esiste `DoubleStream.range` perché contare con double ha problemi di arrotondamento.

21.10.3 Metodi di mapping specializzati per primitivi

Gli stream primitivi forniscono operazioni di mapping **solo per primitivi** che evitano 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 Tabella di mapping tra `Stream<T>` e stream primitivi

Questa tabella riassume le principali conversioni tra stream di oggetti e stream primitivi.

La colonna “From” ti dice quali metodi sono disponibili e il tipo di stream target risultante.

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**
>
> - Non esiste un'operazione **`unboxed()`**.
> - Per passare da wrapper a primitivi devi partire da `Stream<T>` e usare `mapToInt` / `mapToLong` / `mapToDouble`.

21.10.5 Operazioni terminali e i loro tipi di risultato

Gli stream primitivi hanno diverse operazioni terminali che sono uniche o hanno tipi di ritorno specifici per primitivi.

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**
>
> - Anche per `IntStream` e `LongStream`, **`average()`** restituisce `OptionalDouble` (non `OptionalInt` o `OptionalLong`).

- Esempio 1: `Stream<String>` → `IntStream` → operazioni terminali primitive.

```
List<String> words = List.of("a", "bb", "ccc");

int totalLength = words.stream()
    .mapToInt(String::length) // IntStream
        .sum(); // int

// totalLength = 1 + 2 + 3 = 6
```

- Esempio 2: `IntStream` → boxed `Stream<Integer>` (boxing introdotto).

```
Stream<Integer> boxed = IntStream.rangeClosed(1, 3) // 1,2,3
    .boxed(); // Stream<Integer>
```

- Esempio 3: stream primitivo → stream di oggetti via `mapToObj`.

```
Stream<String> labels = IntStream.range(1, 4) // 1,2,3
    .mapToObj(i -> "N=" + i); // Stream<String>
```

21.10.6 Trappole e gotcha comuni

- Non confondere `Stream<Integer>` con `IntStream`: i loro metodi di mapping e interfacce funzionali differiscono
- `IntStream.sum()` restituisce `int` ma `IntStream.count()` restituisce `long`
- `average()` restituisce sempre `OptionalDouble` per tutti i tipi di stream primitivi
- Usare `boxed()` reintroduce boxing; fallo solo se l'API downstream richiede oggetti (es. raccogliere in `List<Integer>`)
- Fai attenzione alle conversioni di narrowing: `LongStream.mapToInt` e `DoubleStream.mapToInt` possono troncare i valori

---

21.11 Collector (collect(), Collector e i Metodi Factory di Collectors)

Un `Collector` descrive come accumulare elementi di stream in un risultato finale.

L'operazione terminale `collect(...)` esegue questa ricetta.

La classe utility `Collectors` fornisce collector pronti per compiti comuni di aggregazione.

21.11.1 collect() vs Collector

Ci sono due modi principali per raccogliere:

- `collect(Collector)` → la forma comune usando `Collectors.*`
- `collect(supplier, accumulator, combiner)` → riduzione mutabile esplicita (più low-level)

```
List<String> list =
Stream.of("a", "b")
    .collect(Collectors.toList());

StringBuilder sb =
Stream.of("a", "b")
    .collect(StringBuilder::new, StringBuilder::append, StringBuilder::append);
```

> **Note**
>
> Usa `collect(supplier, accumulator, combiner)` quando ti serve un contenitore mutabile custom e non vuoi implementare un `Collector` completo.

21.11.2 Collector core

Questi sono i collector usati più frequentemente e quelli più probabilmente presenti in domande d'esame.

- `toList()` → `List<T>` (nessuna garanzia su mutabilità/implementazione)
- `toSet()` → `Set<T>`
- `toCollection(supplier)` → tipo di collezione specifico (es. `TreeSet`)
- `joining(delim, prefix, suffix)` → `String` da elementi `CharSequence`
- `counting()` → conteggio `Long`
- `summingInt` / `summingLong` / `summingDouble` → somme numeriche
- `averagingInt` / `averagingLong` / `averagingDouble` → medie numeriche
- `minBy(comparator)` / `maxBy(comparator)` → `Optional<T>`
- `mapping(mapper, downstream)` → trasforma poi raccoglie con downstream
- `filtering(predicate, downstream)` → filtra dentro il collector (Java 9+)

21.11.3 Collector di raggruppamento

`groupingBy` classifica elementi in bucket con chiave data da una funzione classifier.

Produce una `Map<K, V>` dove `V` dipende dal collector downstream.

```
Map<Integer, List<String>> byLen =
Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length));
System.out.println("byLen: " + byLen.toString());
```

Output:

```
byLen: {1=[a], 2=[bb, dd], 3=[ccc]}
```

Con un collector downstream controlli cosa contiene ogni bucket:

```
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());
```

Output:

```
countByLen: {1=1, 2=2, 3=1}
setByLen: {1=[a], 2=[bb, dd], 3=[ccc]}
```

Warning

Fai attenzione al tipo del valore della mappa risultante. Esempio: `groupingBy(..., counting())` produce `Map<K, Long>` (non `int`).

21.11.4 partitioningBy

`partitioningBy` divide lo stream in esattamente due gruppi usando un `Predicate` booleano. Restituisce sempre una mappa con chiavi `true` e `false`.

```
Map<Boolean, List<String>> parts =
Stream.of("a", "bb", "ccc")
    .collect(Collectors.partitioningBy(s -> s.length() > 1));
System.out.println("parts: " + parts.toString());
```

Output:

```
parts: {false=[a], true=[bb, ccc]}
```

Note

`partitioningBy` crea sempre due bucket, mentre `groupingBy` può crearne molti. Entrambi supportano collector downstream.

21.11.5 toMap e regole di merge

`toMap` lancia un'eccezione su chiavi duplicate a meno che tu non fornisca una funzione di merge.

```
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)); //
```

Output:

```
m2: {2=aa,bb,cc}
```

21.11.6 collectingAndThen

`collectingAndThen(downstream, finisher)` ti permette di applicare una trasformazione finale dopo la raccolta (es. rendere la lista non modificabile).

```
List<String> unmodifiable =
Stream.of("a", "b", "c")
    .collect(Collectors.collectingAndThen(Collectors.toList(), List::copyOf));
```

21.11.7 Come i collector si relazionano agli stream paralleli

I collector sono progettati per funzionare con stream paralleli usando supplier/accumulator/combiner internamente. In parallelo, ogni worker costruisce un contenitore di risultato parziale e poi unisce i contenitori.

- L'accumulator muta un contenitore per-thread (nessuno stato condiviso mutabile)
- Il combiner unisce i contenitori (richiesto per esecuzione parallela)
- Alcuni collector sono "concurrent" o hanno caratteristiche che influenzano prestazioni e ordinamento

Note

preferisci `collect(Collectors.toList())` rispetto a usare `reduce` per costruire collezioni. `reduce` è per riduzioni in stile immutabile; `collect` è per contenitori mutabili.

◀ 20. Programmazione Funzionale in Java | ▲ Index | 22. Introduzione al Framework delle Collezioni ▶

Collections Framework

22. Introduzione al Framework delle Collezioni

Indice

- 22.1 Che cos'è il Framework delle Collezioni
- 22.2 Le Interfacce Principali
  - 22.2.1 Principali interfacce di Collection
  - 22.2.2 Gerarchia di Map
- 22.3 Collezioni Sequenced Java-21
- 22.4 Perché esiste il Framework delle Collezioni
- 22.5 I due lati del Framework Collections-vs-Maps
- 22.6 Tipi generici nel Framework delle Collezioni
- 22.7 Mutabilità vs Immutabilità
- 22.8 Aspettative di Prestazioni Big-O
- 22.9 Riepilogo

Il `Java Collections Framework (JCF)` è un insieme di **interfacce, classi e algoritmi** progettato per memorizzare, manipolare ed elaborare gruppi di dati in modo efficiente.

Fornisce un'architettura unificata per gestire collezioni, consentendo agli sviluppatori di scrivere codice riutilizzabile e interoperabile con comportamenti prevedibili e caratteristiche di prestazioni.

Questo capitolo introduce i concetti fondamentali necessari prima di studiare List, Set, Queue, Map e Sequenced Collections, esplorati in dettaglio nei capitoli successivi.

22.1 Che cos'è il Framework delle Collezioni?

Il Framework delle Collezioni fornisce:

- Un **insieme di interfacce** (Collection, List, Set, Queue, Deque, Map…)
- Un **insieme di implementazioni** (ArrayList, HashSet, TreeSet, LinkedList…)
- Un **insieme di algoritmi di utilità** (ordinamento, ricerca, copia, inversione…) in java.util.Collections e java.util.Arrays.
- Un linguaggio comune per le aspettative di prestazioni (complessità Big-O).

Tutte le principali strutture di collezione condividono un design coerente così che il codice che funziona con un'implementazione può spesso essere riutilizzato con un'altra.

22.2 Le Interfacce Principali

Al cuore del Java Collections Framework c'è un piccolo insieme di **interfacce radice** che definiscono comportamenti generici di gestione dei dati.

- **List**: una collezione `ordinata` di elementi che consente `duplicati`;
- **Set**: una collezione che non consente `duplicati`;
- **Queue**: una collezione progettata per contenere elementi in corso di elaborazione, tipicamente FIFO (first-in-first-out), con varianti come priority queue e deque.
- **Map**: una struttura che mappa chiavi a valori, dove non sono consentite chiavi duplicate; ogni chiave può mappare al massimo un valore.

22.2.1 Principali interfacce di Collection

Sotto è riportata la gerarchia concettuale.

```
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'interfaccia **Map** non estende Collection perché una map memorizza coppie chiave/valore piuttosto che singoli valori.

22.2.2 Gerarchia di 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 Collezioni Sequenced (Java 21+)

Java 21 introduce la nuova interfaccia `SequencedCollection`, che formalizza l'idea che una collezione mantenga un **ordine di encounter definito**. Questo era già vero per List, LinkedHashSet, LinkedHashMap, Deque, ecc., ma ora il comportamento è standardizzato.

- `SequencedCollection` definisce metodi come `getFirst()`, `getLast()`, `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()`, e `reversed()`.
- SequencedSet, SequencedMap estendono l'idea per set e map.

Questo semplifica drasticamente la specifica dei comportamenti di ordinamento e sarà usato in tutti i capitoli seguenti.

22.4 Perché esiste il Framework delle Collezioni

- Evitare di reinventare le strutture dati
- Fornire algoritmi ben testati e ad alte prestazioni

- Migliorare l'interoperabilità tramite interfacce condivise
- Supportare tipi generici per collezioni type-safe

Prima di Java 1.2, le strutture dati erano ad-hoc, incoerenti e non tipizzate.

Il Collections Framework ha unificato tutto questo in una API coerente.

22.5 I due lati del Framework: Collections vs. Maps

"Map estende Collection?" **No.** Una Map memorizza **coppie**, mentre una Collection memorizza **singoli elementi**.

- Collection = List, Set, Queue, Deque, SequencedCollection
- Map = archivio chiave/valore in stile dizionario

22.6 Tipi generici nel Framework delle Collezioni

Le collezioni sono quasi sempre usate con i generics. L'uso di raw types è sconsigliato.

```
List<String> names = new ArrayList<>();
Map<Integer, String> map = new HashMap<>();
```

> **Note**
>
> I generics nelle collezioni funzionano tramite `type erasure`: fai riferimento al Paragrafo "**18.4 Type Erasure**" nel Capitolo: Generics in Java.

22.7 Mutabilità vs. Immutabilità

Molti metodi nella Collections API restituiscono collezioni **unmodifiable**:

```
List<String> immutable = List.of("a", "b");
immutable.add("c"); // ❌ UnsupportedOperationException
```

Java fornisce diversi modi per creare collezioni immutabili:

- `List.of()`, `Set.of()`, `Map.of()`
- `List.copyOf(collection)`
- wrapper `Collections.unmodifiableList(...)`
- `Records` usati come contenitori di valori immutabili

> **Note**
>
> Il metodo `Arrays.asList(varargs)`, che è costruito su un array, si comporta diversamente: vedi esempi sotto.

```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


// Non possiamo fare  ADD o REMOVE di elementi da "fromAsList" ma possiamo sostituirli;
// o modificando l'array sottostante "vargs" o mutando la lista stessa:


fromAsList.set(0, "k");  // l'aggiornamento sarà riflesso anche sull'array sottostante.
```

> **Note**
>
> `Arrays.asList(...)` restituisce una vista List a dimensione fissa, ma **mutabile**, supportata dall'array originale. Non puoi aggiungere/rimuovere elementi, ma puoi sostituire quelli esistenti.

22.8 Aspettative di Prestazioni Big-O

Capire la complessità dei tipi "Collectio" è essenziale. Ecco alcuni esempi comuni:

Type	Methods	Complexity
ArrayList	`get()`, `add()`, `remove()`	**`O(1)`, `O(1) ammortizzato`, `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) in media`**
TreeMap	`get()/put()`	**`O(log n)`**
Deque	`add/remove first/last`	**`O(1)`**

> **Note**
>
> Questi valori sono medie; il caso peggiore può essere diverso (specialmente per strutture basate su hash).

22.9 Riepilogo

- Il Collection Framework è costruito su un piccolo insieme di interfacce principali.
- Java 21 aggiunge Sequenced Collections per unificare il comportamento di ordinamento.
- Le Map non sono Collection — formano una gerarchia parallela.
- Le collezioni fanno massiccio uso dei generics.
- La mutabilità conta — i metodi factory spesso restituiscono collezioni immutabili.
- Le caratteristiche prestazionali sono prevedibili.

◀ 21. Java Optional e Streams | ▲ Index | 23. Operazioni Condivise delle Collezioni & Uguaglianza ▶

23. Operazioni Condivise delle Collezioni & Uguaglianza

Indice

- 23.1 Metodi Fondamentali delle Collezioni Disponibili per la Maggior Parte delle Collezioni
  - 23.1.1 Operazioni di Mutazione
  - 23.1.2 Operazioni di Query
- 23.2 Uguaglianza
- 23.3 Comportamento Fail-Fast
- 23.4 Operazioni Bulk
- 23.5 Tipi di Ritorno ed Eccezioni Comuni
- 23.6 Tabella di Riepilogo — Operazioni Condivise

Questo capitolo copre le operazioni fondamentali condivise in tutta la Java Collections API, incluso il modo in cui viene determinata l'uguaglianza all'interno delle collezioni stesse.

Questi concetti si applicano a tutte le principali famiglie di collezioni basate su Collection (List, Set, Queue, Deque e le loro varianti Sequenced).

Map condivide diversi comportamenti concettuali (iterazione, uguaglianza) ma non eredita da Collection.

Padroneggiare queste operazioni è essenziale, poiché spiegano come le collezioni si comportano quando si aggiungono, cercano, rimuovono, confrontano, iterano e ordinano elementi.

23.1 Metodi Fondamentali delle Collezioni (Disponibili per la Maggior Parte delle Collezioni)

I seguenti metodi provengono dall'interfaccia `Collection<E>` e sono ereditati da **tutte** le principali collezioni eccetto `Map` (che ha una propria famiglia di operazioni).

> **Note**
>
> `Map` non implementa `Collection`, ma le sue viste `keySet()`, `values()` ed `entrySet()` **lo fanno**, esponendo, quindi, queste operazioni condivise.

23.1.1 Operazioni di Mutazione

- `boolean add(E e)` — Aggiunge un elemento (le liste consentono duplicati).
- `boolean remove(Object o)` — Rimuove il primo elemento corrispondente.
- `void clear()` — Rimuove tutti gli elementi.
- `boolean addAll(Collection<? extends E> c)` — Inserimento bulk.
- `boolean removeAll(Collection<?> c)` — Rimuove tutti gli elementi contenuti nella collezione fornita.
- `boolean retainAll(Collection<?> c)` — Mantiene solo gli elementi corrispondenti.

23.1.2 Operazioni di Query

- `int size()` — Numero di elementi.
- `boolean isEmpty()` — Indica se la collezione contiene zero elementi.
- `boolean contains(Object o)` — Si basa sulle regole di uguaglianza degli elementi.
- `Iterator<E> iterator()` — Restituisce un iteratore (fail-fast).
- `Object[] toArray()` e `<T> T[] toArray(T[] a)` — Copia in un array.

23.2 Uguaglianza

Un'implementazione personalizzata del metodo `equals()` consente di confrontare il tipo e il contenuto di due collezioni.

L'implementazione differirà a seconda che si tratti di `List` o di `Set`.

- Esempio

```
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));
```

Output

```
firstList.equals(secondList): true
secondList.equals(thirdList): false
firstSet.equals(secondSet): true
secondSet.equals(thirdSet): true
```

Note

- Le List confrontano dimensione, ordine ed uguaglianza degli elementi uno per uno.
- I Set confrontano solo dimensione e appartenenza — l'ordine di encounter è irrilevante.
- Due set con gli stessi elementi logici sono uguali anche se mantengono internamente ordini di iterazione diversi.

23.3 Comportamento Fail-Fast

La maggior parte degli iteratori delle collezioni (eccetto le collezioni concorrenti) sono `fail-fast`: modificare strutturalmente una collezione durante l'iterazione provoca una `ConcurrentModificationException`.

```
List<Integer> list = new ArrayList<>(List.of(1,2,3));
for (Integer i : list) {
    list.add(99); // ❌ ConcurrentModificationException
}
```

Note

Usa `Iterator.remove()` quando devi rimuovere elementi durante l'iterazione. Il comportamento fail-fast **non è garantito** — l'eccezione viene lanciata secondo il principio del best-effort. Non devi fare affidamento sulla sua intercettazione per la correttezza del programma.

23.4 Operazioni Bulk

- `removeIf(Predicate<? super E> filter)` — Rimuove tutti gli elementi corrispondenti.
- `replaceAll(UnaryOperator<E> op)` — Sostituisce ogni elemento.
- `forEach(Consumer<? super E> action)` — Applica un'azione a ciascun elemento.

- `stream()` — Restituisce uno stream per operazioni di pipeline.

23.5 Tipi di Ritorno ed Eccezioni Comuni

- `add(E)` restituisce **boolean** — sempre `true` per `ArrayList`, può essere `false` per i `Set` se non avviene alcuna modifica.
- `remove(Object)` restituisce boolean (non l'elemento rimosso).
- `get(int)` lancia `IndexOutOfBoundsException`.
- `iterator().remove()` lancia `IllegalStateException` se chiamato due volte senza `next()`.
- `toArray()` restituisce sempre un `Object[]` — non un `T[]`.

23.6 Tabella di Riepilogo — Operazioni Condivise

Operazione	Si applica a	Note
`add(e)`	Tutte le collezioni eccetto Map	Le List consentono duplicati
`remove(o)`	Tutte le collezioni eccetto Map	Rimuove la prima occorrenza
`contains(o)`	Tutte le collezioni eccetto Map	Usa equals()
`size(), isEmpty()`	Tutte le collezioni	Tempo costante per la maggior parte
`iterator()`	Tutte le collezioni	Fail-fast
`clear()`	Tutte le collezioni	Rimuove tutti gli elementi
`stream()`	Tutte le collezioni	Restituisce stream sequenziale
`removeIf(), replaceAll()`	Solo List (la maggior parte dei Set non supporta replaceAll)	Operazioni bulk
`toArray()`	Tutte le collezioni	Restituisce Object[]

◀ 22. Introduzione al Framework delle Collezioni | ▲ Index | 24. Comparable, Comparator & Ordinamento in Java

24. Comparable, Comparator & Ordinamento in Java

Indice

- 24.1 Comparable — Ordinamento Naturale
  - 24.1.1 Contratto del Metodo di Comparable
  - 24.1.2 Classe di Esempio che Implementa Comparable
  - 24.1.3 Errori Comuni di Comparable
- 24.2 Comparator — Ordinamento Personalizzato
  - 24.2.1 Metodi Principali di Comparator
    - 24.2.1.1 Metodi di Supporto **Statici** di Comparator
    - 24.2.1.2 Metodi di **Istanza** su Comparator
  - 24.2.2 Esempio di Comparator
- 24.3 Comparable vs Comparator
- 24.4 Ordinamento di Array e Collezioni
  - 24.4.1 Arrays sort
  - 24.4.2 Collections sort
- 24.5 Ordinamento Multi-Livello thenComparing
- 24.6 Confrontare Primitivi in Modo Efficiente
- 24.7 Trappole Comuni
- 24.8 Esempio Completo
- 24.9 Riepilogo

Java fornisce due strategie principali per l'ordinamento e il confronto: `Comparable` (ordinamento naturale) e `Comparator` (ordinamento personalizzato).

Comprendere le loro regole, i vincoli e le interazioni con i `generics` è essenziale.

- Per i **tipi numerici**, l'ordinamento segue l'ordine numerico naturale, il che significa che i valori più piccoli vengono prima di quelli più grandi.
- L'ordinamento delle **stringhe** segue l'ordine lessicografico (`code point` Unicode): confronto carattere per carattere; le cifre vengono prima delle maiuscole, le maiuscole prima delle minuscole.

Questo ordinamento si basa sul `code point Unicode` di ogni carattere, non sull'intuizione alfabetica.

Un **Unicode code point** è un valore numerico unico assegnato ai caratteri nell'Unicode standard.

Più precisamente: un `Unicode code point` è un integer (scritto in esadecimale come U+XXXX) che rappresenta uno specifico carattere, simbolo, o carattere speciale indipendentemente da font, lingua, o piattaforma.

- Esempi:
  - U+0041 → A
  - U+0061 → a
  - U+0030 → 0
  - U+1F600 → 😀

Un code point non è una sequenza di byte; è un numero astratto.

Come il code point sia poi stoccato nella memoria fisica dipende dall'encoding (UTF-8, UTF-16, UTF-32).

Unicode definisce code point da U+0000 a U+10FFFF.

In breve: Unicode code points definisce quale sia il carattere; l'encodings definisce come questo sia rappresentato in bytes.

- Esempi di natural ordering

```
List<String> items = List.of("10", "2", "A", "Z", "a", "b");

List<String> sorted = new ArrayList<>(items);
Collections.sort(sorted);

System.out.println(sorted);
```

Output:

```
[10, 2, A, Z, a, b]
```

> **Note**
>
> L'ordinamento naturale è definito solo per i tipi che implementano `Comparable`.

24.1 Comparable — Ordinamento Naturale

L'interfaccia `Comparable<T>` definisce l'ordine naturale di un tipo.

Una classe la implementa quando vuole definire la propria regola di ordinamento predefinita.

24.1.1 Contratto del Metodo di Comparable

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

Regole e restituzione:

- Restituisce **negativo** → `this` < `other`
- Restituisce **zero** → `this` == `other`
- Restituisce **positivo** → `this` > `other`

> **Important**
>
> - L'ordinamento naturale deve essere consistente con `equals()`, a meno che non sia esplicitamente documentato diversamente:
> - `compareTo()` è consistente con `equals()` se, e solo se, `a.compareTo(b) == 0` e `a.equals(b) è true`.

> **Warning**
>
> compareTo può lanciare ClassCastException se riceve un tipo non confrontabile — ma questo di solito succede solo con tipi raw.

24.1.2 Esempio: Classe che Implementa 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 lista viene ordinata per età, perché quello è l'ordine numerico naturale.

24.1.3 Errori Comuni di Comparable

- Confrontare tutti i campi rilevanti → risultati inconsistenti se non lo si fa
- Violare la transitività → porta a comportamento indefinito
- Lanciare eccezioni dentro compareTo() rompe l'ordinamento
- Non implementare la stessa logica di equals() → trappola comune

---

24.2 Comparator — Ordinamento Personalizzato

L'interfaccia `Comparator<T>` consente di definire più strategie di ordinamento senza modificare la classe stessa.

24.2.1 Metodi Principali di Comparator

```
int compare(T a, T b);
```

Metodi di supporto aggiuntivi:

24.2.1.1 Metodi di Supporto Statici di Comparator

Metodo	Statico / Istanza	Tipo di Ritorno	Parametri	Descrizione
`Comparator.comparing(keyExtractor)`	statico	Comparator	Function<? super T, ? extends U>	Costruisce un comparator che confronta le chiavi estratte usando l'ordinamento naturale.
`Comparator.comparing(keyExtractor, keyComparator)`	statico	Comparator	Function<T,U>, Comparator	Costruisce un comparator che confronta le chiavi estratte usando un comparator personalizzato.
`Comparator.comparingInt(keyExtractor)`	statico	Comparator	ToIntFunction	Comparator ottimizzato per chiavi int (evita il boxing).
`Comparator.comparingLong(keyExtractor)`	statico	Comparator	ToLongFunction	Comparator ottimizzato per chiavi long.
`Comparator.comparingDouble(keyExtractor)`	statico	Comparator	ToDoubleFunction	Comparator ottimizzato per chiavi double.
`Comparator.naturalOrder()`	statico	Comparator	none	Comparator che usa l'ordinamento naturale (Comparable).
`Comparator.reverseOrder()`	statico	Comparator	none	Ordinamento naturale inverso.
`Comparator.nullsFirst(comparator)`	statico	Comparator	Comparator	Incapsula un comparator in modo che i null vengano confrontati prima dei non-null.
`Comparator.nullsLast(comparator)`	statico	Comparator	Comparator	Incapsula un comparator in modo che i null vengano confrontati dopo i non-null.

24.2.1.2 Metodi di Istanza su Comparator

Metodo	Statico / Istanza	Tipo di Ritorno	Parametri	Descrizione
`thenComparing(otherComparator)`	istanza	Comparator	Comparator	Aggiunge un comparator secondario quando il primario confronta come uguali.
`thenComparing(keyExtractor)`	istanza	Comparator	Function<T,U>	Confronto secondario usando l'ordinamento naturale della chiave estratta.
`thenComparing(keyExtractor, keyComparator)`	istanza	Comparator	Function<T,U>, Comparator	Confronto secondario con comparator personalizzato.
`thenComparingInt(keyExtractor)`	istanza	Comparator	ToIntFunction	Confronto numerico secondario (ottimizzato).
`thenComparingLong(keyExtractor)`	istanza	Comparator	ToLongFunction	Confronto numerico secondario.
`thenComparingDouble(keyExtractor)`	istanza	Comparator	ToDoubleFunction	Confronto numerico secondario.
`reversed()`	istanza	Comparator	none	Restituisce un comparator invertito per la stessa logica di confronto.

24.2.2 Esempio di 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

Caratteristica	Comparable	Comparator
Package	java.lang	java.util
Metodo	compareTo(T)	compare(T,T)
Tipo di Ordinamento	Naturale (predefinito)	Personalizzato (strategie multiple)
Modifica la Classe Sorgente	SI	NO
Utile Per	Ordinamento predefinito	Ordinamento esterno o alternativo
Consente Ordini Multipli	NO	SI
Usato da Collections.sort	SI	SI
Usato da Arrays.sort	SI	SI

24.4 Ordinamento di Array e Collezioni

24.4.1 Arrays sort()

```
int[] nums = {3,1,2};
Arrays.sort(nums); // ordine naturale

Person[] arr = {...};
Arrays.sort(arr); // Person deve implementare Comparable
Arrays.sort(arr, byName); // usando Comparator
```

24.4.2 Collections sort()

```
Collections.sort(list); // ordine naturale
Collections.sort(list, byName); // comparator
```

> **Note**
>
> Collections.sort(list) delega a list.sort(comparator) da Java 8.

24.5 Ordinamento Multi-Livello (thenComparing)

```
var cmp = Comparator
    .comparing(Person::getLastName)
        .thenComparing(Person::getFirstName)
            .thenComparingInt(Person::getAge);
```

24.6 Confrontare Primitivi in Modo Efficiente

```
Comparator.comparingInt(Person::getAge)
Comparator.comparingLong(...)
Comparator.comparingDouble(...)
```

> **Note**
>
> Questi evitano il boxing e sono preferiti nel codice sensibile alle prestazioni.

24.7 Trappole Comuni

- Ordinare una lista di Object senza Comparable → ClassCastException a runtime
- compareTo inconsistente con equals → comportamento imprevedibile
- Comparator che rompe la transitività → l'ordinamento diventa indefinito
- Elementi null → a meno che il Comparator li gestisca, l'ordinamento lancia NPE
- Comparator che confronta campi di tipi misti → ClassCastException
- Usare la sottrazione per confrontare int può causare overflow → usare sempre `Integer.compare()`
- Ordinare una lista con elementi null e ordine naturale → NPE
- compareTo non deve mai restituire negativo/zero/positivo inconsistenti sugli stessi due oggetti (niente casualità)

24.8 Esempio Completo

```
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()` si applica all'intero comparator composto, non solo alla prima chiave di confronto.

24.9 Riepilogo

- Usare `Comparable` per l'ordinamento naturale (1 ordine predefinito).
- Usare `Comparator` per strategie di ordinamento flessibili o multiple.
- I comparator possono essere composti (reversed, thenComparing).
- L'ordinamento richiede una logica di confronto consistente.
- Arrays.sort e Collections.sort usano sia Comparable che Comparator.

◀ 23. Operazioni Condivise delle Collezioni & Uguaglianza | ▲ Index | 25. L'API List ▶

25. L'API List

Indice

- 25.1 Caratteristiche delle List
- 25.2 Creare List (Costruttori)
  - 25.2.1 Costruttori di ArrayList
  - 25.2.2 Costruttori di LinkedList
- 25.3 Metodi Factory
  - 25.3.1 List of immutable
  - 25.3.2 List copyOf immutable-copy
  - 25.3.3 Arrays asList fixed-size-list
- 25.4 Operazioni Fondamentali di List
  - 25.4.1 Aggiungere Elementi
  - 25.4.2 Accedere agli Elementi
  - 25.4.3 Rimuovere Elementi
  - 25.4.4 Comportamenti e Caratteristiche Importanti
- 25.5 contains, equals e hashCode
  - 25.5.1 contains
  - 25.5.2 Uguaglianza delle List
  - 25.5.3 hashCode
- 25.6 Iterare una List
  - 25.6.1 Ciclo For Classico
  - 25.6.2 Ciclo For Migliorato
  - 25.6.3 Iterator–ListIterator
- 25.7 Il Metodo subList
  - 25.7.1 Sintassi
  - 25.7.2 Regole
  - 25.7.3 Esempi
  - 25.7.4 Modificare la lista padre invalida la vista
  - 25.7.5 Modificare la subList modifica il padre
  - 25.7.6 Svuotare la subList svuota parte della lista padre
  - 25.7.7 Trappole Comuni
- 25.8 Tabella Riassuntiva delle Operazioni Importanti

Nel `Collections Framework`, una **List** rappresenta una collezione ordinata, basata su indice, che consente duplicati.

L'interfaccia List estende `Collection` ed è implementata da:

```
List
├── ArrayList (Array ridimensionabile — accesso casuale veloce, inserimenti/rimozioni più lenti nel mezzo)
├── LinkedList (Lista doppiamente collegata — inserimenti/rimozioni veloci, accesso casuale più lento)
└── Vector (Lista sincronizzata legacy — raramente usata oggi)
```

> **Note**
>
> Vector è legacy e sincronizzato — evitarlo a meno che non sia esplicitamente richiesto.

25.1 Caratteristiche delle List

- Ordinate — gli elementi preservano l'ordine di inserimento.
- Indicizzate — accessibili tramite `get(int)` e `set(int,E)`.
- Consentono duplicati — `List` non impone unicità.
- Possono contenere `null` — a meno di usare implementazioni speciali.

25.2 Creare List (Costruttori)

25.2.1 Costruttori di ArrayList

```
List<String> a1 = new ArrayList<>();
List<String> a2 = new ArrayList<>(50); // capacità iniziale
List<String> a3 = new ArrayList<>(List.of("A", "B"));
```

> **Note**
>
> La capacità iniziale non è una dimensione. Decide solo quanti elementi l'array interno può contenere prima di ridimensionarsi.

25.2.2 Costruttori di LinkedList

```
List<String> l1 = new LinkedList<>();
List<String> l2 = new LinkedList<>(List.of("A", "B"));
```

> **Note**
>
> `LinkedList` implementa anche `Deque`.

25.3 Metodi Factory

25.3.1 `List.of()` (immutabile)

```
List<String> list1 = List.of("A", "B", "C");
list1.add("X"); // ❌ UnsupportedOperationException
list1.set(0, "Z"); // ❌ UnsupportedOperationException
```

> **Note**
>
> Tutte le liste `List.of()`: - non accettano i `null` - sono immutabili - lanciano `UOE` su modifiche strutturali

25.3.2 `List.copyOf()` (copia immutabile)

```
List<String> src = new ArrayList<>();
src.add("Hello");

List<String> copy = List.copyOf(src); // snapshot immutabile
```

25.3.3 Arrays.asList() (lista a dimensione fissa)

```
String[] arr = {"A", "B"};
List<String> list = Arrays.asList(arr);

list.set(0, "Z"); // OK
list.add("X"); // ❌ UOE — la dimensione è fissa
```

> **Note**
>
> La lista è supportata dall'array: modificare uno modifica anche l'altro.

25.4 Operazioni Fondamentali di List

25.4.1 Aggiungere Elementi

```
list.add("A");
list.add(1, "B"); // inserisce all'indice
list.addAll(otherList);
list.addAll(2, otherList);
```

25.4.2 Accedere agli Elementi

```
String x = list.get(0);
list.set(1, "NewValue");
```

> **Note**
>
> `get()` lancia `IndexOutOfBoundsException` per indici non validi.

Se si tenta di `aggiornare` un elemento in una List vuota, anche all'indice 0, si ottiene una `IndexOutOfBoundsException`

```
List<Integer> list = new ArrayList<Integer>();
list.add(3);
list.add(5);
System.out.println(list.toString());
list.clear();
list.set(0, 2);
```

Output

```
[3, 5]
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of bounds for leng
```

> **Warning**
>
> Chiamare get/set con un indice non valido lancia IndexOutOfBoundsException

25.4.3 Rimuovere Elementi

```
list.remove(0); // remove(int index) — rimuove per indice; remove(Object) — rimuove il primo e
list.remove("A"); // rimuove la prima occorrenza
list.removeIf(s -> s.startsWith("X"));
list.clear();
```

25.4.4 Comportamenti e Caratteristiche Importanti

Operazione	Comportamento	Eccezione(i)
`add(E)`	aggiunge sempre in coda	—
`add(int,E)`	sposta gli elementi a destra	IndexOutOfBoundsException
`get(int)`	tempo costante per ArrayList, lineare per LinkedList	IndexOutOfBoundsException
`set(int,E)`	sostituisce l'elemento	IndexOutOfBoundsException
`remove(int)`	sposta gli elementi a sinistra	IndexOutOfBoundsException
`remove(Object)`	rimuove il primo elemento uguale	—

25.5 `contains()`, `equals()` e `hashCode()`

25.5.1 `contains()`

Il metodo `contains()` usa `.equals()` sugli elementi.

25.5.2 Uguaglianza delle List

`List.equals()` esegue un confronto elemento per elemento, in ordine.

```
List<String> a = List.of("A", "B");
List<String> b = List.of("A", "B");

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

> **Note**
>
> - L'ordine conta.
> - Il tipo di lista NON conta.

25.5.3 `hashCode()`

Calcolato in base al contenuto.

25.6 Iterare una List

25.6.1 Ciclo For Classico

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

25.6.2 Ciclo For Migliorato

```
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

Tutti gli iteratori standard di List sono fail-fast: una modifica strutturale fuori dall'iteratore causa ConcurrentModificationException.

Note

Solo `ListIterator` supporta l'iterazione bidirezionale e la modifica.

25.7 Il Metodo `subList()`

`subList()` crea una vista di una porzione della lista, non una copia. Modificare una delle due può modificare l'altra.

25.7.1 Sintassi

```
List<E> subList(int fromIndex, int toIndex);
```

25.7.2 Regole

Regola	Spiegazione
fromIndex inclusivo	l'elemento a fromIndex è incluso
toIndex esclusivo	l'elemento a toIndex NON è incluso
La vista è supportata dalla lista originale	modificare una modifica l'altra
Modifica strutturale del padre invalida la subList	→ ConcurrentModificationException

25.7.3 Esempi

```
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 Modificare la lista padre invalida la vista

```
List<String> list = new ArrayList<>(List.of("A","B","C","D"));
List<String> view = list.subList(1, 3);

list.add("E"); // modifica strutturale della lista padre

view.get(0); // ❌ ConcurrentModificationException
```

25.7.5 Modificare la subList modifica il padre

```
view.remove(1);
// rimuove "C" sia dalla view che dalla lista padre
```

25.7.6 Svuotare la subList svuota parte della lista padre

```
view.clear();
// rimuove gli indici 1 e 2 dalla lista padre
```

25.7.7 Trappole Comuni

- Supporre che subList sia indipendente: è una vista, non una copia
- Supporre che subList consenta il ridimensionamento: funziona solo su liste padre modificabili
- Dimenticare che le modifiche al padre invalidano la vista causando ConcurrentModificationException
- Aspettative errate sugli indici: l'indice finale è esclusivo

---

25.8 Tabella Riassuntiva delle Operazioni Importanti

Operazione	ArrayList	LinkedList	List Immutabili
`add(E)`	veloce	veloce	❌ non supportato
`add(index,E)`	lento (shift)	veloce	❌
`get(index)`	veloce	lento	veloce
`remove(index)`	lento	lento (a meno che si rimuova primo/ultimo)	❌
`remove(Object)`	più lento	più lento	❌
`set(index,E)`	veloce	lento	❌
`iterator()`	veloce	veloce	veloce
`listIterator()`	veloce	veloce	veloce
`contains(Object)`	O(n)	O(n)	O(n)

````
---

::: {.chapter-nav}
[◀ 24. Comparable, Comparator & Ordinamento in Java](#chap-comparing) | [▲ Index](#book-index) | [26. Set API ▶](#chap-set-api)
:::


# 26. Set API {#chap-set-api}

### Indice {#indice}


- [26.1 Gerarchia dei Set Java-Collections-Framework](#261-gerarchia-dei-set-java-collections-framework)
- [26.2 Caratteristiche di Ogni Implementazione di Set](#262-caratteristiche-di-ogni-implementazione-di-set)
    - [26.2.1 HashSet](#2621-hashset)
    - [26.2.2 LinkedHashSet](#2622-linkedhashset)
    - [26.2.3 TreeSet](#2623-treeset)
- [26.3 Regole di Uguaglianza nei Set](#263-regole-di-uguaglianza-nei-set)
    - [26.3.1 HashSet--LinkedHashSet](#2631-hashset--linkedhashset)
    - [26.3.2 TreeSet](#2632-treeset)
- [26.4 Creare Istanze di Set](#264-creare-istanze-di-set)
    - [26.4.1 Usando i Costruttori](#2641-usando-i-costruttori)
    - [26.4.2 Costruttori di Copia](#2642-costruttori-di-copia)
    - [26.4.3 Metodi Factory](#2643-metodi-factory)
- [26.5 Operazioni Principali sui Set](#265-operazioni-principali-sui-set)
    - [26.5.1 Aggiungere Elementi](#2651-aggiungere-elementi)
    - [26.5.2 Verificare l’Appartenenza](#2652-verificare-lappartenenza)
    - [26.5.3 Rimuovere Elementi](#2653-rimuovere-elementi)
    - [26.5.4 Operazioni Bulk](#2654-operazioni-bulk)
- [26.6 Trappole Comuni](#266-trappole-comuni)
- [26.7 Tabella Riassuntiva](#267-tabella-riassuntiva)


---

Un **Set** in Java rappresenta una collezione che **non contiene elementi duplicati**.

Modella il concetto matematico di `insieme`: non ordinato (a meno di usare
un’implementazione ordinata) e composto da valori unici.

Tutte le implementazioni di Set si basano su **semantiche di uguaglianza** (tramite
`equals()` oppure logica di `Comparator`).

## 26.1 Gerarchia dei Set (Java Collections Framework) {#261-gerarchia-dei-set-java-collections-framework}

```text
Set<E>
 ├── SequencedSet<E> (Java 21+)
 │     └── LinkedHashSet<E>    (ordinato)
 ├── HashSet<E>                (non ordinato)
 └── SortedSet<E>
       └── NavigableSet<E>
             └── TreeSet<E>    (ordinato)
````

Tutte le implementazioni di `Set` richiedono:

- unicità degli elementi
- uguaglianza e hashing prevedibili (a seconda dell’implementazione)

> **Note**
>
> `LinkedHashSet` è ora formalmente un `SequencedSet` a partire da Java 21.

````

26.2 Caratteristiche di Ogni Implementazione di Set

26.2.1 HashSet

- Set generico più veloce
- Non ordinato (nessuna garanzia sull'ordine di iterazione)
- Usa `hashCode()` ed `equals()`
- Consente un solo elemento `null`

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("B");
set.add("A");   // duplicato ignorato
System.out.println(set); // ordine non garantito
```

26.2.2 LinkedHashSet

- Mantiene l'**ordine di inserimento**
- Leggermente più lento di HashSet
- Utile quando è richiesto un ordine di iterazione prevedibile

```
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 **ordinato** il cui ordine è determinato da:

- Ordinamento naturale (`Comparable`)
- Un `Comparator` fornito

TreeSet:

- Non consente elementi `null` (NullPointerException a runtime)
- Garantisce iterazione ordinata
- Supporta viste di intervallo: `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` richiede che tutti gli elementi siano mutuamente confrontabili — mescolare tipi non confrontabili produce `ClassCastException` . Le operazioni (add, remove, contains) sono O(log n).

26.3 Regole di Uguaglianza nei Set

Le regole differiscono in base all'implementazione.

26.3.1 HashSet & LinkedHashSet

L' `unicità` è determinata da due metodi:

- `hashCode()`
- `equals()`

Due oggetti sono considerati lo stesso elemento se:

- I loro hash code coincidono
- Il loro metodo `equals()` restituisce `true`

> **Warning**
>
> Se si muta un oggetto dopo averlo aggiunto a un HashSet o LinkedHashSet, il suo hashCode può cambiare e il set può perdere il riferimento a quell'elemento.

26.3.2 TreeSet

L'unicità è basata su `compareTo()` o sul `Comparator` fornito.

Se `compare(a, b) == 0` allora gli oggetti sono considerati duplicati, anche se `equals()` restituisce false.

```
Comparator<String> comp = (a, b) -> a.length() - b.length();
Set<String> set = new TreeSet<>(comp);

set.add("Hi");
set.add("Yo"); // stessa lunghezza → trattato come duplicato

System.out.println(set);  // ["Hi"]
```

26.4 Creare Istanze di Set

26.4.1 Usando i Costruttori

```
Set<String> s1 = new HashSet<>();
Set<String> s2 = new LinkedHashSet<>();
Set<String> s3 = new TreeSet<>();
```

26.4.2 Costruttori di Copia

```
List<String> list = List.of("A", "B", "C");

Set<String> copy = new HashSet<>(list); // ordine perso
System.out.println(copy);

Set<String> ordered = new LinkedHashSet<>(list); // mantiene l'ordine della lista
System.out.println(ordered);
```

26.4.3 Metodi Factory

```
Set<String> s1 = Set.of("A", "B", "C");    // immutabile
Set<String> empty = Set.of();              // set immutabile vuoto
```

> **Note**
>
> I set creati tramite factory sono **immutabili**: aggiungere o rimuovere elementi lancia `UnsupportedOperationException`. `Set.of(...)` rifiuta duplicati in fase di creazione → IllegalArgumentException e rifiuta null → NullPointerException

---

26.5 Operazioni Principali sui Set

26.5.1 Aggiungere Elementi

```
set.add("A");          // restituisce true se aggiunto
set.add("A");          // restituisce false se duplicato
```

26.5.2 Verificare l'Appartenenza

```
set.contains("A");
```

26.5.3 Rimuovere Elementi

```
set.remove("A");
set.clear();
```

26.5.4 Operazioni Bulk

```
set.addAll(otherSet);
set.removeAll(otherSet);
set.retainAll(otherSet); // intersezione
```

26.6 Trappole Comuni

- Usare TreeSet con oggetti non confrontabili → `ClassCastException`
- TreeSet non usa affatto `equals()` : solo comparator/compareTo determina l'unicità
- Usare oggetti mutabili come chiavi di Set → rompe le regole di hashing
- I Set creati con Set.of() sono immutabili — la modifica fallisce
- HashSet non garantisce l'ordine di iterazione
- TreeSet tratta oggetti con compare()==0 come duplicati anche se non uguali

---

26.7 Tabella Riassuntiva

Implementazione	Mantiene l'Ordine?	Consente Null?	Ordinato?	Logica Sottostante
HashSet	No	Sì (1 null)	No	hashCode + equals
LinkedHashSet	Sì (ordine di inserimento)	Sì (1 null)	No	tabella hash + lista collegata
TreeSet	Sì (ordinato)	No	Sì (naturale/comparator)	compareTo / Comparator

---

◀ 25. L'API List | ▲ Index | 27. API Queue & Deque ▶

27. API Queue & Deque

Indice

- 27.1 Queue — Panoramica
  - 27.1.1 Metodi Principali di Queue
  - 27.1.2 Implementazioni di Queue
- 27.2 Deque — Panoramica
  - 27.2.1 Metodi Principali di Deque
  - 27.2.2 Implementazioni di Deque
- 27.3 Usare una Queue
- 27.4 Usare una Deque come Queue e come Stack
  - 27.4.1 Esempio FIFO (Comportamento Queue)
  - 27.4.2 Esempio LIFO (Comportamento Stack)
- 27.5 PriorityQueue — Queue Speciale
- 27.6 Blocking Queue (Basi)
- 27.7 Trappole Comuni
- 27.8 Tabella Riassuntiva

Le interfacce `Queue` e `Deque` di Java modellano collezioni ordinate progettate per elaborare elementi in una sequenza specifica.

- Una **Queue** modella tipicamente una struttura **FIFO** (First-In, First-Out).
- Una **Deque** (`double-ended queue`) consente inserimento e rimozione da entrambe le estremità, permettendo comportamenti **FIFO** e **LIFO** in una singola API.

27.1 Queue — Panoramica

L'interfaccia `Queue` estende `Collection` ed è comunemente utilizzata nella programmazione asincrona, nella distribuzione del carico, negli algoritmi e nel buffering.

Esistono due famiglie di metodi: quelli che **lanciano eccezioni** e quelli che **restituiscono valori speciali** (di solito `null`).

27.1.1 Metodi Principali di Queue

Operazione	Lancia Eccezione	Restituisce Valore Speciale	Descrizione
Inserimento	`add(e)`	`offer(e)`	Aggiunge un elemento; `offer` è preferibile per queue con capacità limitata
Rimozione	`E remove()`	`E poll()`	Rimuove e restituisce la testa. `remove()` lancia NoSuchElementException se la queue è vuota, `poll()` restituisce null
Lettura	`E element()`	`E peek()`	Restituisce la testa senza rimuoverla. `element()` lancia NoSuchElementException se la queue è vuota, `peek()` restituisce null

27.1.2 Implementazioni di Queue

Classi comuni che implementano `Queue`:

- `LinkedList` — non limitata, implementa anche `Deque` e `List`.
- `ArrayDeque` — queue veloce basata su array ridimensionabile; non può contenere `null`.
- `PriorityQueue` — ordina gli elementi per ordine naturale o comparator; non è FIFO.
- `ConcurrentLinkedQueue` — thread-safe, lock-free.

> **Note**
>
> `PriorityQueue` non garantisce che l'ordine di iterazione corrisponda all'ordinamento per priorità.

> **Warning**
>
> La maggior parte delle implementazioni di Queue rifiuta `null` perché `null` è usato come valore di ritorno per "vuoto".

27.2 Deque — Panoramica

`Deque` (double-ended queue) supporta inserimento, rimozione e ispezione sia dalla testa sia dalla coda.

È più versatile di una Queue:

- FIFO (simile a una queue)
- LIFO (simile a uno stack)
- Algoritmi bidirezionali

27.2.1 Metodi Principali di Deque

Operazione	Fronte	Fondo
Inserimento	addFirst(e), offerFirst(e)	addLast(e), offerLast(e)
Rimozione	removeFirst(), pollFirst()	removeLast(), pollLast()
Ispezione	getFirst(), peekFirst()	getLast(), peekLast()

27.2.2 Implementazioni di Deque

- `ArrayDeque` — implementazione consigliata per uso generale (veloce, senza limite di capacità).
- `LinkedList` — completa ma più lenta a causa della struttura a nodi.
- `ConcurrentLinkedDeque` — deque concorrente non bloccante.

> **Note**
>
> `Stack` è legacy; usare `Deque` per il comportamento di stack (push/pop). Le operazioni di queue di ArrayDeque e LinkedList (add/remove/peek) sono O(1) ammortizzato.

27.3 Usare una Queue

```java
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 vuota)
```

27.4 Usare una Deque (come Queue e come Stack)

27.4.1 Esempio FIFO (Comportamento Queue)

```java
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 Esempio LIFO (Comportamento Stack)

```java
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 Speciale

`PriorityQueue` ordina gli elementi per **ordine naturale** o tramite un `Comparator` fornito.

Caratteristiche importanti:

- Non FIFO — la testa è l’elemento “più piccolo”.
- L’ordine è garantito solo durante la rimozione, non durante l’iterazione.
- Gli elementi `null` non sono consentiti.

```java
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 (Basi)

Negli ambienti concorrenti, il package `java.util.concurrent` fornisce tipi di queue bloccanti.

- `ArrayBlockingQueue` — array sottostante a dimensione fissa.
- `LinkedBlockingQueue` — opzionalmente limitata.
- `PriorityBlockingQueue` — priority queue thread-safe.
- `DelayQueue` — elementi rilasciati dopo un ritardo.

> **Note**
>
> - BlockingQueue non consente mai `null`.
> - put(e) — blocca finché c'è spazio disponibile
> - take() — blocca finché un elemento è disponibile
> - BlockingQueue supporta anche operazioni temporizzate: offer(e, timeout), poll(timeout)

27.7 Trappole Comuni

- I metodi di `Queue` e `Deque` esistono in varianti "con eccezione" e "con valore speciale" — memorizzare quali sono quali.
- `ArrayDeque` non può contenere `null` — `null` è usato internamente.
- L'ordine di iterazione di `PriorityQueue` NON è ordinato.
- L'uso di `Stack` è sconsigliato; usare invece `Deque`.
- Deque consente sia FIFO sia LIFO e ha l'API **più completa**.

27.8 Tabella Riassuntiva

Interfaccia	Comportamento Tipico	Null Consentito?	Implementazioni Comuni	Note
Queue	FIFO	Dipende	LinkedList, ArrayDeque, PriorityQueue	PriorityQueue non FIFO
Deque	FIFO + LIFO	No (ArrayDeque)	ArrayDeque, LinkedList	Operazioni complete a doppia estremità
PriorityQueue	Ordinata per priorità	No	PriorityQueue	Rimuove prima l'elemento più piccolo
BlockingQueue	FIFO thread-safe	No	ArrayBlockingQueue, LinkedBlockingQueue	differenze tra add/offer e put
ConcurrentLinkedQueue	FIFO lock-free	No	ConcurrentLinkedQueue	Molto veloce per il multi-threading

---

◀ 26. Set API | ▲ Index | 28. Map API ▶

28. Map API

Indice

- 28.1 Caratteristiche Fondamentali di Map
- 28.2 Principali Implementazioni di Map
- 28.3 Creare Map
- 28.4 Operazioni di Base sulle Map
- 28.5 Iterare su una Map
- 28.6 Determinare l'Uguaglianza nelle Map
- 28.7 Comportamento Speciale di TreeMap
- 28.8 Gestione dei Null
- 28.9 Trappole Comuni
- 28.10 Riepilogo

L'interfaccia `Map` rappresenta una collezione di **coppie chiave–valore**, dove ogni chiave è associata ad un unico valore.

A differenza degli altri tipi di collezione, `Map` **non** estende `Collection` e quindi possiede una propria gerarchia e regole specifiche.

28.1 Caratteristiche Fondamentali di Map

- Ogni chiave è unica; **chiavi duplicate sovrascrivono il valore precedente**
- I valori possono essere duplicati
- Le Map non supportano accesso posizionale (basato su indice)
- L'iterazione avviene tramite `keySet()`, `values()` o `entrySet()`

> **Note**
>
> Una `Map` non è una `Collection`, ma le sue viste (keySet, values, entrySet) sono collezioni.

28.2 Principali Implementazioni di Map

Implementazione	Ordinamento	Chiavi Null	Valori Null	Thread-Safe	Note
`HashMap`	Nessun ordine	1	Molti	No	Veloce, la più comune
`LinkedHashMap`	Ordine di inserimento	1	Molti	No	Iterazione prevedibile
`TreeMap`	Ordinata per chiave	No	Molti	No	Le chiavi devono essere confrontabili
`Hashtable`	Nessun ordine	No	No	Sì	Legacy
`ConcurrentHashMap`	Nessun ordine	No	No	Sì	Adatta alla concorrenza

> **Note**
>
> L’ordinamento di `TreeMap` è determinato da `Comparable` o da un `Comparator` fornito al momento della creazione.

28.3 Creare Map

Le `Map` possono essere create usando costruttori o metodi 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**
>
> Le Map create con `Map.of(...)` e `Map.ofEntries(...)` sono **immutabili**. Qualsiasi tentativo di modifica lancia `UnsupportedOperationException`.

28.4 Operazioni di Base sulle Map

Metodo	Descrizione	Valore di Ritorno
`put(k, v)`	Aggiunge o sostituisce una associazione	Valore precedente o null
`putIfAbsent(k,v)`	Aggiunge solo se la chiave non è presente	Valore esistente o null
`get(k)`	Restituisce il valore o null	Valore specifico o null
`getOrDefault(k, default)`	Restituisce valore o default	Valore specifico o default
`remove(k)`	Rimuove l’associazione	Valore rimosso o null
`containsKey(k)`	Verifica presenza chiave	boolean
`containsValue(v)`	Verifica presenza valore	boolean
`size()`	Numero di entry	int
`isEmpty()`	Verifica se vuota	boolean
`clear()`	Rimuove tutte le entry	void
`V merge(k, v, BiFunction(V, V, V))`	merge(k, v, remappingFunction)	se la chiave è assente → imposta il valore; se presente → function(oldValue, newValue); se la funzione restituisce null → mapping rimosso

```java
Map<String, String> map = new HashMap<>();
map.put("A", "Apple");
map.put("B", "Banana");

map.put("A", "Avocado"); // sovrascrive il valore

String v = map.get("B"); // Banana
```

28.5 Iterare su una Map

Le Map vengono iterate tramite le viste:

- `keySet()` → Set di chiavi
- `values()` → Collection di valori
- `entrySet()` → Set di 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**
>
> Modificare la map durante l'iterazione su queste viste può lanciare `ConcurrentModificationException` (eccetto per le map concorrenti).

28.6 Determinare l'Uguaglianza nelle Map

L'uguaglianza tra map è definita come segue:

- Due map sono uguali se contengono le stesse associazioni chiave–valore
- Il confronto delle chiavi usa `equals()`
- Il confronto dei valori usa `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'ordine di iterazione non influisce sull'uguaglianza delle map.

28.7 Comportamento Speciale di TreeMap

TreeMap mantiene le entry in ordinate in base alle chiavi.

```
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**
>
> Tutte le chiavi in una `TreeMap` devono essere mutuamente confrontabili. Mescolare tipi incompatibili causa `ClassCastException` a runtime.

28.8 Gestione dei Null

Implementazione	Chiave Null	Valore Null
HashMap	Sì (1)	Sì
LinkedHashMap	Sì (1)	Sì
TreeMap	No	Sì
Hashtable	No	No
ConcurrentHashMap	No	No

> **Note**
>
> `TreeMap` accetta valori `null` solo quando non partecipano al confronto delle chiavi. In pratica questo è raro, perché le chiavi null sono vietate e i comparator possono rifiutare i null.
>
> `HashMap` e `LinkedHashMap` consentono `una sola chiave null` — inserirne un'altra sostituisce quella esistente.

28.9 Trappole Comuni

- Supporre che Map sia una Collection
- Dimenticare che chiavi duplicate sovrascrivono i valori
- Usare chiavi null in TreeMap o ConcurrentHashMap
- Confondere l'ordine di iterazione con l'uguaglianza
- Tentare di modificare map immutabili create con Map.of

28.10 Riepilogo

- Le Map memorizzano chiavi uniche associate a valori
- L'ordinamento dipende dall'implementazione
- L'uguaglianza è basata sulle coppie chiave–valore
- TreeMap richiede chiavi confrontabili
- Le map immutabili lanciano eccezioni in caso di modifica

◀ 27. API Queue & Deque | ▲ Index | 29. Collezioni Sequenziate & Map Sequenziate ▶

29. Collezioni Sequenziate & Map Sequenziate

Indice

- 29.1 Motivazione e Contesto
- 29.2 Interfaccia SequencedCollection
  - 29.2.1 Metodi Principali di SequencedCollection
  - 29.2.2 Implementazioni di SequencedCollection
  - 29.2.3 Viste Invertite
- 29.3 Interfaccia SequencedMap
  - 29.3.1 Metodi Principali di SequencedMap
  - 29.3.2 Implementazioni di SequencedMap
  - 29.3.3 Map Invertite
- 29.4 Relazione con le API Esistenti
  - 29.4.1 Quali Tipi Built-in Sono Sequenziati
- 29.5 Trappole Comuni
- 29.6 Riepilogo

Java 21 introduce le `Collezioni Sequenziate` e le `Map Sequenziate` per unificare e formalizzare l'accesso agli elementi in base al loro ordine di incontro.

Questa aggiunta risolve incoerenze di lunga data tra liste, set, queue, deque e map, fornendo un'API comune per lavorare con il primo e l'ultimo elemento, oltre che con viste invertite.

29.1 Motivazione e Contesto

Prima di Java 21, le collezioni ordinate (come List, LinkedHashSet, Deque o LinkedHashMap) esponevano operazioni basate sull'ordine tramite metodi diversi o, in alcuni casi, non le esponevano affatto.

Gli sviluppatori dovevano fare affidamento su API specifiche dell'implementazione o su soluzioni indirette.

Le interfacce sequenziate introducono un contratto coerente per tutte le collezioni e map ordinate, rendendo le operazioni basate sull'ordine esplicite, sicure e uniformi.

29.2 Interfaccia SequencedCollection

`SequencedCollection<E>` è una nuova interfaccia che estende `Collection<E>` e rappresenta collezioni con un ordine di incontro ben definito.

È implementata da `List`, `Deque` e `LinkedHashSet` (`TreeSet` è ordinato ma non implementa direttamente SequencedCollection).

29.2.1 Metodi Principali di SequencedCollection

L'interfaccia definisce metodi per accedere e manipolare gli elementi a entrambe le estremità della collezione.

Metodo	Descrizione
`E getFirst()`	Restituisce il primo elemento
`E getLast()`	Restituisce l'ultimo elemento
`void addFirst(E e)`	Inserisce un elemento all'inizio
`void addLast(E e)`	Inserisce un elemento alla fine
`E removeFirst()`	Rimuove e restituisce il primo elemento
`E removeLast()`	Rimuove e restituisce l'ultimo elemento
`SequencedCollection<E> reversed()`	Restituisce una vista invertita

29.2.2 Implementazioni di SequencedCollection

I seguenti tipi standard implementano SequencedCollection:

Tipo	Note
List	Ordinata per indice
Deque	Coda a doppia estremità
LinkedHashSet	Mantiene l'ordine di inserimento

29.2.3 Viste Invertite

La chiamata a `reversed()` non crea una copia.

Restituisce una vista live della stessa collezione con ordine invertito.

```
List<Integer> list = new ArrayList<>(List.of(1, 2, 3));
SequencedCollection<Integer> rev = list.reversed();

rev.removeFirst(); // rimuove 3
System.out.println(list); // [1, 2]
```

> **Note**
>
> Le viste invertite condividono la stessa collezione sottostante. Le modifiche strutturali in una delle due viste influenzano anche l'altra: modificare sia la collezione originale sia la vista invertita ha effetto su entrambe.

29.3 Interfaccia SequencedMap

`SequencedMap<K,V>` estende `Map<K,V>` e rappresenta map con un ordine di incontro delle entry ben definito.

Standardizza operazioni che in precedenza esistevano solo in implementazioni specifiche come `LinkedHashMap`.

29.3.1 Metodi Principali di SequencedMap

Metodo	Descrizione
`Entry<K,V> firstEntry()`	Prima entry della map
`Entry<K,V> lastEntry()`	Ultima entry della map
`Entry<K,V> pollFirstEntry()`	Rimuove e restituisce la prima entry, oppure null se vuota
`Entry<K,V> pollLastEntry()`	Rimuove e restituisce l'ultima entry, oppure null se vuota
`SequencedMap<K,V> reversed()`	Vista invertita della map

29.3.2 Implementazioni di SequencedMap

Attualmente, la principale implementazione standard è:

Tipo	Ordinamento
`LinkedHashMap`	Ordine di inserimento (o ordine di accesso se configurato)

> **Note**
>
> LinkedHashMap può riordinare le entry in lettura se costruita con `accessOrder=true`.
>
> In tal caso, "prima" e "ultima" riflettono l'ordine di accesso più recente.

29.3.3 Map Invertite

Come per le collezioni, `reversed()` su una map sequenziata restituisce una vista, non una copia.

```
SequencedMap<String, Integer> map =
new LinkedHashMap<>(Map.of("A", 1, "B", 2, "C", 3));

SequencedMap<String, Integer> rev = map.reversed();

rev.pollFirstEntry(); // rimuove C=3
System.out.println(map); // {A=1, B=2}
```

> **Note**
>
> Come per SequencedCollection, `reversed()` restituisce una vista live — le mutazioni si applicano a entrambe le map.

29.4 Relazione con le API Esistenti

Le interfacce sequenziate non sostituiscono i tipi di collezione esistenti.

Si collocano sopra di essi nella gerarchia e unificano i comportamenti comuni.

Tutte le collezioni ordinate esistenti beneficiano automaticamente di queste API senza rompere la retrocompatibilità.

29.4.1 Quali Tipi Built-in Sono Sequenziati?

La tabella seguente riassume se i tipi standard di collezione sono ordinati e se implementano le nuove interfacce Sequenced.

Tipo	Ordinato?	SequencedCollection?	SequencedMap?
`List`	✓ Sì	✓ Sì	✗ No
`Deque`	✓ Sì	✓ Sì	✗ No
`LinkedHashSet`	✓ Sì	✓ Sì	✗ No
`TreeSet`	✓ Sì (ordinato)	✗ No*	✗ No
`HashSet`	✗ No	✗ No	✗ No
`LinkedHashMap`	✓ Sì	✗ No	✓ Sì
`HashMap`	✗ No	✗ No	✗ No
`TreeMap`	✓ Sì (ordinato)	✗ No	✗ No

> **Note**
>
> `TreeSet` è ordinato, ma implementa `SortedSet` / `NavigableSet` , non `SequencedCollection` .

29.5 Trappole Comuni

- Le interfacce sequenziate definiscono viste, non copie
- `reversed()` riflette le modifiche in modo bidirezionale
- Non tutte le implementazioni di Set o Map sono sequenziate
- HashSet e HashMap non implementano interfacce sequenziate
- L'ordine è garantito solo quando esplicitamente definito
- La rimozione di elementi tramite iterator sulla vista invertita impatta immediatamente l'ordine originale

---

29.6 Riepilogo

- Le interfacce sequenziate formalizzano l'ordine di incontro
- Forniscono accesso first/last e inversione
- Funzionano tramite viste live, non copie
- Unificano le API tra liste, deque, set e map

◀ 28. Map API | ▲ Index | 30. Thread Java – Fondamenti e Modello di Esecuzione ▶

Module 07

Concurrency and Threads

30. Thread Java – Fondamenti e Modello di Esecuzione

Indice

- 30.1 Thread, Processi e il Sistema Operativo
- 30.2 Modello di Memoria Stack e Heap
- 30.3 Contesto e Context Switching
- 30.4 Concorrenza vs Parallelismo
- 30.5 Thread in Java Modello Concettuale
- 30.6 Categorie di Thread in Java 21
- 30.7 Creare Thread in Java
- 30.8 Ciclo di Vita ed Esecuzione di un Thread
- 30.9 Avviare vs Eseguire un Thread Sincrono-o-Asincrono
- 30.10 Priorità dei Thread e Scheduling
- 30.11 Differimento e Yield dei Thread
- 30.12 Interruzione dei Thread e Cancellazione Cooperativa
  - 30.12.1 Cosa Significa Interrompere un Thread
  - 30.12.2 Interrompere Operazioni Bloccanti
  - 30.12.3 Controllare lo Stato di Interruzione
  - 30.12.4 Esempio Interrompere un Thread in Sleep
  - 30.12.5 Osservazioni Chiave
- 30.13 Thread e il Thread Principale
- 30.14 Concorrenza dei Thread e Stato Condiviso
- 30.15 Sommario

Questo capitolo introduce i **thread** a partire dai principi di base e spiega come sono modellati e utilizzati in Java 21.

Questo testo stabilisce inoltre le fondamenta concettuali necessarie per comprendere `concurrency`, `synchronization` e la `Java Concurrency API` trattata nel prossimo capitolo.

30.1 Thread, Processi e il Sistema Operativo

Per comprendere i thread, dobbiamo partire dal modello di esecuzione del sistema operativo.

I moderni sistemi operativi eseguono programmi utilizzando **processi** e **thread**.

- **Processo**: Un'istanza di programma in esecuzione gestita dal sistema operativo. Un processo possiede il proprio spazio di memoria virtuale, risorse di sistema (file, socket) e almeno un thread.
- **Thread**: Un'unità di esecuzione leggera all'interno di un processo. I thread condividono memoria e risorse del processo ma eseguono in modo indipendente.
- **Task**: Un'unità logica di lavoro da eseguire. Un task può essere eseguito da un thread ma non è esso stesso un thread.
- **Core CPU**: Un'unità di esecuzione fisica o logica capace di eseguire un thread alla volta. Più core permettono vera esecuzione parallela.

Un singolo processo può contenere molti thread, tutti operanti nello stesso ambiente condiviso. Questo ambiente condiviso è sia fonte delle potenzialità della Concurrency sia dei suoi rischi.

30.2 Modello di Memoria: Stack e Heap

I thread interagiscono con la memoria in due modi fondamentalmente diversi.

- **Stack del Thread**: Area di memoria privata per ogni thread. Memorizza frame delle chiamate di metodo, variabili locali e stato di esecuzione. Ogni thread ha esattamente uno stack.
- **Heap**: Area di memoria condivisa usata per oggetti e istanze di classe. Tutti i thread nello stesso processo possono accedere all'heap.

Poiché gli `stack sono isolati` e l' `heap è condiviso`, i problemi di concorrenza sorgono quando più thread accedono agli stessi oggetti nell'heap senza adeguata coordinazione.

30.3 Contesto e Context Switching

Il sistema operativo pianifica l'esecuzione dei thread sui core della CPU.

Poiché il numero di thread eseguibili spesso supera il numero di core disponibili, il sistema operativo esegue il **context switching**.

- **Contesto**: Lo stato completo di esecuzione di un thread, inclusi registri, contatore di programma e puntatore allo stack.
- **Context Switch**: L'atto di sospendere un thread e riprenderne un altro salvando e ripristinando i rispettivi contesti.

Il `context switching` abilita la concorrenza ma ha un costo: cicli CPU vengono consumati senza eseguire logica applicativa.

I programmatori Java devono progettare sistemi che bilancino concorrenza ed efficienza.

30.4 Concorrenza vs Parallelismo

Questi due termini sono spesso confusi ma descrivono concetti differenti.

- **Concorrenza**: Più thread sono in esecuzione nello stesso intervallo di tempo, possibilmente interlacciati su un singolo core CPU.
- **Parallelismo**: Più thread vengono eseguiti simultaneamente su core CPU differenti.

Java supporta la concorrenza indipendentemente dal parallelismo hardware.

Anche su un sistema single-core, i thread Java possono essere concorrenti tramite time slicing.

30.5 Thread in Java: Modello Concettuale

In Java, un **thread** rappresenta un percorso indipendente di esecuzione all'interno di un singolo processo JVM. Tutti i thread Java operano nello stesso heap e nello stesso contesto di `class loading`, a meno che non siano esplicitamente isolati tramite meccanismi avanzati.

- **Thread Java**: Un oggetto di tipo `java.lang.Thread` che mappa a un'unità di esecuzione sottostante.
- **Runnable**: Un'interfaccia funzionale che rappresenta un `task` il cui metodo `run()` contiene la logica eseguibile.

Un thread esegue codice invocando il proprio metodo `run()`, direttamente o indirettamente tramite lo scheduler dei thread della JVM: vedere Avviare vs Eseguire un Thread

---

30.6 Categorie di Thread in Java 21

Java 21 definisce diversi tipi di thread, che differiscono per ciclo di vita, scheduling e uso previsto.

- **Platform Thread**: Un thread Java tradizionale mappato uno-a-uno con un thread del sistema operativo.
- **Virtual Thread**: Un thread leggero gestito dalla JVM e schedulato su thread carrier. Introdotto per abilitare massiva concorrenza con overhead minimo.
- **Carrier Thread**: Un Platform Thread usato internamente dalla JVM per eseguire thread virtuali.
- **Daemon Thread**: Un thread in background che non impedisce la terminazione della JVM. Quando restano in esecuzione solo thread daemon, la JVM termina.
- **Thread Utente**: Qualsiasi thread non-daemon. La JVM attende che tutti i thread utente completino prima di terminare.
- **Thread di Sistema**: Thread creati internamente dalla JVM per garbage collection, compilazione JIT e altri servizi runtime.

> **Note**
>
> - I `virtual threads` sono thread utente leggeri ; non sono **daemon** per default ;
> - Un VirtualThread (creato direttamente tramite `Thread.startVirtualThread()` oppure `Thread.ofVirtual().start(...)`) accetta un Runnable come task. Non accetta direttamente un Callable : Se è necessario eseguire un Callable con virtual threads e ottenere un risultato, è necessario utilizzare un ExecutorService ;
> - I virtual threads sono implementati dalla classe `java.lang.VirtualThread`. Questa classe estende `BaseVirtualThread`, che a sua volta estende Thread. Pertanto, un virtual thread è tecnicamente una sottoclasse di Thread. Tuttavia, non è corretto descrivere un virtual thread come un istanza diretta della classe Thread, poiché è in realtà un istanza di una sottoclasse specializzata progettata specificamente per il comportamento dei virtual threads.

30.7 Creare Thread in Java

I thread possono essere creati in diversi modi, tutti concettualmente centrati nel fornire logica eseguibile.

- Estendendo `Thread` e sovrascrivendo `run()`.
- Passando un `Runnable` al costruttore di `Thread`.
- Usando factory di thread ed executor (trattati nella sezione Concurrency API).

```
Runnable runnable = ...

  // Crea un thread di piattaforma tramite costruttore
  Thread thread = new Thread(runnable);
  thread.start();

  // Avvia un thread daemon per eseguire un task
  Thread thread = Thread.ofPlatform().daemon().start(runnable);

  // Crea un thread non avviato con nome "duke", il suo metodo start()
  // deve essere invocato per pianificarne l'esecuzione.
  Thread thread = Thread.ofPlatform().name("duke").unstarted(runnable);

  // Una ThreadFactory che crea thread daemon chiamati "worker-0", "worker-1", ...
  ThreadFactory factory = Thread.ofPlatform().daemon().name("worker-", 0).factory();

  // Avvia un thread virtuale per eseguire un task
  Thread thread = Thread.ofVirtual().start(runnable);

  // Una ThreadFactory che crea thread virtuali
  ThreadFactory factory = Thread.ofVirtual().factory();
```

Warning

- La sola creazione di un thread non ne avvia l'esecuzione.
- L'esecuzione inizia solo quando lo scheduler della JVM è coinvolto.

30.8 Ciclo di Vita ed Esecuzione di un Thread

Un thread Java attraversa stati ben definiti durante il suo ciclo di vita.

- **New**: Oggetto thread creato ma non ancora avviato.
- **Runnable**: Idoneo all'esecuzione da parte dello scheduler.
- **Running**: In esecuzione attiva su un core CPU.
- **Blocked / Waiting**: Temporaneamente incapace di proseguire a causa di sincronizzazione o coordinazione.
- **Terminated**: Esecuzione completata o interrotta.

La JVM e il sistema operativo cooperano per muovere i thread tra questi stati.

I thread in stato `BLOCKED`, `WAITING` o `TIMED_WAITING` **non stanno utilizzando risorse CPU**

30.9 Avviare vs Eseguire un Thread: Sincrono o Asincrono

Esiste una distinzione concettuale critica tra invocare `run()` e invocare `start()`.

- Chiamare direttamente `run()` esegue il metodo in modo sincrono nel thread corrente, come una normale chiamata di metodo.
- Chiamare `start()` richiede alla JVM di creare un nuovo stack di chiamata ed eseguire `run()` in modo asincrono in un thread separato.

Pertanto, codice come `new Thread(r).run();` NON crea concorrenza. L'esecuzione rimane sincrona e blocca il thread chiamante fino al completamento.

Note

`Esecuzione asincrona` significa che il chiamante continua immediatamente mentre il nuovo thread prosegue in modo indipendente, soggetto allo scheduling.

`Esecuzione sincrona` significa che il chiamante attende che l'operazione sia completata.

Important

La concorrenza inizia **solo** quando viene invocato `start()`.

30.10 Priorità dei Thread e Scheduling

I thread Java hanno una priorità associata che influenza lo scheduling.

- `Priorità del Thread`: Un valore intero che ne indica l'importanza relativa, che va da minimo a massimo.
- `Scheduling`: La JVM delega le decisioni di scheduling al sistema operativo, che può o meno rispettare rigorosamente le priorità.

La priorità del thread influenza la probabilità di scheduling ma non garantisce mai l'ordine di esecuzione. Il codice Java portabile non deve mai fare affidamento sulle priorità per la correttezza.

È possibile impostare la **priorità** sui `thread di piattaforma`; per i `thread virtuali` la **priorità** è sempre impostata a **5** (`Thread.NORM_PRIORITY`) e tentare di modificarla non ha effetto.

30.11 Differimento e Yield dei Thread

I thread possono influenzare volontariamente il comportamento di scheduling.

Chiamare `Thread.yield()` segnala la disponibilità a sospendere l'esecuzione.

- `Yielding`: Un thread suggerisce di essere disposto a sospendere l'esecuzione per permettere ad altri thread eseguibili di proseguire.
- `Sleeping`: Un thread sospende l'esecuzione per una durata fissa, entrando in uno stato di attesa temporizzata.

Questi meccanismi non garantiscono l'esecuzione immediata di altri thread; forniscono solo suggerimenti di scheduling.

30.12 Interruzione dei Thread e Cancellazione Cooperativa

I thread Java non possono essere fermati forzatamente dall'esterno.

Invece, Java fornisce un meccanismo cooperativo chiamato **interruzione del thread**, che permette a un thread di richiedere che un altro thread interrompa ciò che sta facendo.

Il thread di destinazione decide come e quando rispondere.

30.12.1 Cosa Significa Interrompere un Thread

Interrompere un thread **non** lo termina. Chiamare `interrupt()` imposta un **flag di interruzione** interno sul thread di destinazione. È responsabilità del thread in esecuzione osservare questo flag e reagire in modo appropriato.

- `Richiesta di Interruzione`: Un segnale inviato a un thread che indica che dovrebbe fermarsi o cambiare la propria attività corrente.
- `Flag di Interruzione`: Uno stato booleano associato a ciascun thread, impostato quando viene invocato `interrupt()`.
- `Cancellazione Cooperativa`: Un design pattern in cui i thread controllano periodicamente eventuali interruzioni e terminano in modo pulito.

30.12.2 Interrompere Operazioni Bloccanti

Alcuni metodi bloccanti in Java rispondono immediatamente all'interruzione lanciando `InterruptedException` e azzerando il flag di interruzione. Questi metodi includono `sleep()`, `wait()` e `join()`.

Quando un thread è bloccato in uno di questi metodi e un altro thread lo interrompe, il thread bloccato viene risvegliato e viene lanciata un'eccezione. Questo fornisce un punto di uscita sicuro dalle operazioni bloccanti.

30.12.3 Controllare lo Stato di Interruzione

I thread che non sono bloccati devono controllare esplicitamente se sono stati interrotti. Java fornisce due modi per farlo.

- `Thread.currentThread().isInterrupted()`: Restituisce lo stato di interruzione senza azzerarlo.
- `Thread.interrupted()`: Restituisce lo stato di interruzione e lo azzera. Questo è sottile: la chiamata successiva restituirà false.

Non controllare lo stato di interruzione può far sì che i thread ignorino richieste di cancellazione e continuino a eseguire indefinitamente.

30.12.4 Esempio: Interrompere un Thread in Sleep

Il seguente esempio dimostra la cancellazione cooperativa tramite interruzione.

Un thread worker dorme mentre esegue del lavoro. Il thread main lo interrompe, causando uno shutdown pulito.

```
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'ordine dell'output può variare leggermente a causa dello scheduling.

30.12.5 Osservazioni Chiave

- Chiamare `interrupt()` non ferma direttamente il thread.
- L'interruzione viene rilevata e `sleep()` lancia una `InterruptedException`.
- Il thread worker termina da solo in modo controllato.
- Una corretta gestione dell'interruzione permette ai thread di rilasciare risorse e mantenere la coerenza del programma.

> **Note**
>
> Ignorare `InterruptedException` senza terminare o ripristinare lo stato di interruzione è considerata cattiva pratica e può portare a thread non reattivi.

30.13 Thread e il Thread Principale

Ogni applicazione Java inizia con un **thread principale**. Questo thread esegue il metodo `main(String[])`.

- Il thread principale è un thread utente.
- La JVM rimane attiva finché almeno un thread utente è in esecuzione.
- Se il thread principale termina ma esistono altri thread utente, la JVM continua l'esecuzione aspettando che i thread utente terminino.
- I thread daemon non mantengono viva la JVM.

Comprendere il ruolo del thread principale è essenziale per ragionare sulla terminazione del programma e sull'elaborazione in background.

30.14 Concorrenza dei Thread e Stato Condiviso

La `Concorrenza` nasce quando più thread accedono a stato mutabile condiviso.

- `Stato Condiviso`: Qualsiasi dato locato nello heap accessibile da più di un thread.
- `Race Condition`: Un errore di correttezza causato da accesso non sincronizzato a stato condiviso.
- `Problema di Visibilità`: Un thread opera su dati obsoleti a causa della mancanza di corretta sincronizzazione della memoria.

Java risolve questi problemi con sincronizzazione, volatile, lock, atomiche e framework di alto livello (Executors, futures).

La sincronizzazione, le variabili volatile e le utility di concorrenza di alto livello saranno studiate nelle sezioni successive.

30.15 Sommario

- I `Thread` sono il mattone fondamentale dell'esecuzione concorrente in Java.
- Esistono all'interno dei processi, condividono memoria e sono schedulati dalla JVM in cooperazione con il sistema operativo.
- Una corretta gestione dei thread evita perdite, deadlock e spreco di CPU.

◀ 29. Collezioni Sequenziate & Map Sequenziate | ▲ Index | 31. Java Concurrency APIs ▶

31. Java Concurrency APIs

Indice

- 31.1 Obiettivi e Ambito della Concurrency API
- 31.2 Problemi Fondamentali del Threading
  - 31.2.1 Race Conditions
  - 31.2.2 Deadlock
  - 31.2.3 Starvation
  - 31.2.4 Livelock
- 31.3 Dai Thread ai Task
- 31.4 Executor Framework
  - 31.4.1 Submitting Task e Futures
  - 31.4.2 Callable vs Runnable
- 31.5 Thread Pools e Scheduling
- 31.6 Lifecycle e Terminazione dell'Executor
- 31.7 Strategie di Thread Safety
  - 31.7.1 Sincronizzazione
  - 31.7.2 Variabili Atomiche
    - 31.7.2.1 Atomic classes
    - 31.7.2.2 Metodi Atomici
  - 31.7.3 Lock Framework
    - 31.7.3.1 Lock implementations
    - 31.7.3.2 Common Lock methods
  - 31.7.4 Coordination Utilities
- 31.8 Concurrent Collections
- 31.9 Parallel Streams
- 31.10 Relazione con Virtual Threads
- 31.11 Sommario

Questo capitolo introduce la **Java Concurrency API**, che fornisce astrazioni di alto livello per gestire la concorrenza in modo sicuro, efficiente e scalabile.

A differenza della manipolazione di basso livello dei thread presentata nel capitolo precedente, la Concurrency API si concentra su **task**, **executor** e **meccanismi di coordination**, permettendo ai programmatori di ragionare su cosa debba essere fatto piuttosto che su come i thread vengano schedulati.

31.1 Obiettivi e Ambito della Concurrency API

La `Java Concurrency API`, principalmente collocata nel package `java.util.concurrent`, è stata introdotta per affrontare problemi fondamentali inerenti alla gestione manuale dei thread.

- Separare la sottomissione dei task dalla gestione dei thread.
- Ridurre la `synchronization` di basso livello soggetta a errori.
- Migliorare scalabilità e performance su sistemi multi-core.
- Fornire meccanismi strutturati per `coordination`, `cancellation` e `shutdown`.

L'API non elimina i problemi di concorrenza ma fornisce strumenti per gestirli in modo sicuro e prevedibile.

Invece di creare e controllare esplicitamente i thread, i programmatori eseguono task e lasciano che il framework gestisca **thread allocation, riuso, e synchronization**.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Task executed"));
executor.shutdown();
```

31.2 Problemi Fondamentali del Threading

Prima di comprendere la `Concurrency API`, è essenziale comprendere le problematiche di concorrenza che essa vuole mitigare.

Questi problemi sorgono da `shared mutable state`, `scheduling unpredictability` e `improper coordination`.

31.2.1 Race Conditions

Una **race condition** si verifica quando più thread accedono a `shared mutable state` (uno stato mutabile e condiviso) e la correttezza del programma dipende dal timing o dall'intercalare della loro esecuzione.

- Causata da accesso non sincronizzato a dati condivisi.
- Porta a stato del programma inconsistente o incorretto.

```
class Counter {
    int count = 0;
    void increment() {
        count++;
    }
}
```

Se più thread invocano `increment()` in modo concorrente, alcuni incrementi possono andare persi perché l'operazione non è atomica.

31.2.2 Deadlock

Un **deadlock** si verifica quando due o più thread sono bloccati in modo permanente, ciascuno in attesa di una risorsa detenuta da un altro thread.

- Tipicamente causato da dipendenze circolari tra lock.
- Nessun thread coinvolto può fare progressi.

```
synchronized (lockA) {
    synchronized (lockB) {
    }
}
```

Se un altro thread acquisisce prima `lockB` e poi attende `lockA`, può verificarsi un deadlock.

> **Note**
>
> I deadlock nel mondo reale coinvolgono tipicamente lock multipli e inversioni d'ordine.

31.2.3 Starvation

La **starvation** si verifica quando a un thread viene negato indefinitamente l'accesso alle risorse, anche se tali risorse sono disponibili.

- Spesso causata da `unfair locking` o policy di scheduling.
- Il thread rimane `runnable` ma non viene mai eseguito.

```
ReentrantLock lock = new ReentrantLock(false); // unfair lock
```

Alcuni thread possono acquisire ripetutamente il lock mentre altri attendono indefinitamente.

31.2.4 Livelock

In un **livelock**, i thread non sono bloccati ma reagiscono continuamente l'uno all'altro in un modo che ne impedisce il progresso.

- I thread rimangono attivi ma inefficaci.

- Spesso causato da logica di retry o avoidance aggressiva.

```
while (!tryLock()) {
    Thread.sleep(10);
}
```

Entrambi i thread possono ripetere continuamente il retry, impedendo il forward progress.

31.3 Dai Thread ai Task

La Concurrency API sposta il modello di programmazione dalla gestione diretta dei **thread** alla sottomissione di **task**.

Un **task** rappresenta un'unità logica di lavoro indipendente dal thread che lo esegue.

- **Runnable**: Rappresenta un task che non restituisce un risultato.
- **Callable**: Rappresenta un task che restituisce un risultato e può lanciare checked exceptions.

```
Runnable task = () -> System.out.println("Runnable task");
Callable<Integer> callable = () -> 42;
```

Questa astrazione permette ai task di essere riusati, schedulati in modo flessibile ed eseguiti tramite strategie di esecuzione differenti.

31.4 Executor Framework

L'**Executor Framework** è il cuore della Concurrency API.

Gestisce la creazione dei thread, il riuso ed l'esecuzione dei task attraverso una interfaccia semplice.

- **Executor**: Interfaccia di base per eseguire task.
- **ExecutorService**: Estende Executor con controllo del lifecycle e gestione dei risultati.
- **ScheduledExecutorService**: Supporta esecuzione di task delayed e periodici.

```
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 e Futures

I task sottomessi tramite `execute()` restituiscono `void`: è un metodo "fire-and-forget" che non restituisce alcuna informazione sul risultato del task.

I task sottomessi usando `submit()` restituiscono un **Future**, che rappresenta il risultato di una computazione asincrona.

Entrambi i metodi sono usati per sottomettere lavoro per esecuzione asincrona.

```
Future<Integer> future = executor.submit(() -> 10 + 20);
Integer result = future.get();
```

Method	Description
void **execute(Runnable task)**	Esegue un task in modo asincrono senza valore di ritorno e senza `Future`.
Future<?> **submit(Runnable task)**	Esegue un task in modo asincrono; non viene prodotto alcun risultato (`Future.get()` restituisce `null`).
Future **submit(Callable task)**	Esegue un task in modo asincrono e restituisce un risultato di tipo `T`.
List<Future> **invokeAll(Collection<? extends Callable> tasks)**	Esegue tutti i task e restituisce un `Future` per ciascuno, dopo che tutti completano.
T **invokeAny(Collection<? extends Callable> tasks)**	Esegue i task e restituisce il risultato di uno che completa con successo; gli altri vengono cancellati.

Method	Description
boolean **isDone()**	Restituisce `true` se il task è completato (normalmente, eccezionalmente, o via cancellazione).
boolean **isCancelled()**	Restituisce `true` se il task è stato cancellato prima del completamento normale.
boolean **cancel(boolean mayInterruptIfRunning)**	Tenta di cancellare l'esecuzione. Se `true`, interrompe il thread in esecuzione se possibile.
T **get()**	Blocca fino al completamento e restituisce il risultato, o lancia un'eccezione se fallito o cancellato.
T **get(long timeout, TimeUnit unit)**	Blocca fino al timeout dato e restituisce il risultato, o lancia `TimeoutException` se non completato.

> **Warning**
>
> `execute()` scarterà le eccezioni silenziosamente a meno che non vengano gestite all'interno del task.

31.4.2 Callable vs Runnable

Entrambe le interfacce rappresentano task, ma con capacità differenti.

- `Runnable`: Nessun valore di ritorno, non può lanciare checked exceptions.
- `Callable`: Restituisce un valore e supporta checked exceptions.

```
Callable<String> c = () -> "done";
Runnable r = () -> System.out.println("done");
```

Per computazione asincrona orientata al risultato, `Callable` è generalmente preferito.

31.5 Thread Pools e Scheduling

Gli executor gestiscono thread pools che riutilizzano un numero fisso o dinamico di thread per eseguire i task in modo efficiente.

- **Fixed thread pool**: Limita la concorrenza a un numero fisso di thread.
- **Cached thread pool**: Cresce e si riduce dinamicamente in base alla domanda: crea nuovi thread quando necessario ma riusa thread disponibili.
- **Single-thread executor**: Garantisce esecuzione sequenziale dei task.
- **Scheduled executor**: Supporta task delayed e periodici.

Metodo Factory	Tipo di Ritorno	Descrizione
`Executors.newFixedThreadPool(int nThreads)`	ExecutorService	Crea un thread pool con un numero fisso di thread.
`Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFactory)`	ExecutorService	Come newFixedThreadPool ma con un ThreadFactory personalizzato.
`Executors.newSingleThreadExecutor()`	ExecutorService	Crea un thread pool a singolo thread che esegue i task in modo sequenziale.
`Executors.newSingleThreadExecutor(ThreadFactory threadFactory)`	ExecutorService	Executor a singolo thread con un ThreadFactory personalizzato.
`Executors.newCachedThreadPool()`	ExecutorService	Crea un thread pool che crea nuovi thread quando necessario e riutilizza quelli inattivi.
`Executors.newCachedThreadPool(ThreadFactory threadFactory)`	ExecutorService	Thread pool cached con un ThreadFactory personalizzato.
`Executors.newSingleThreadScheduledExecutor()`	ScheduledExecutorService	Crea un scheduled executor a singolo thread.
`Executors.newSingleThreadScheduledExecutor(ThreadFactory threadFactory)`	ScheduledExecutorService	Scheduled executor a singolo thread con ThreadFactory personalizzato.
`Executors.newScheduledThreadPool(int corePoolSize)`	ScheduledExecutorService	Crea un scheduled thread pool con la dimensione core specificata.
`Executors.newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)`	ScheduledExecutorService	Scheduled thread pool con ThreadFactory personalizzato.
`Executors.newWorkStealingPool()`	ExecutorService	Crea un work-stealing pool usando il numero di processori disponibili come livello di parallelismo.
`Executors.newWorkStealingPool(int parallelism)`	ExecutorService	Crea un work-stealing pool con il livello di parallelismo specificato.
`Executors.newThreadPerTaskExecutor(ThreadFactory threadFactory)`	ExecutorService	Crea un executor che avvia un nuovo thread per ogni task.

Metodo Factory	Tipo di Ritorno	Descrizione
`Executors.newVirtualThreadPerTaskExecutor()`	ExecutorService	Crea un executor che avvia un nuovo virtual thread per ogni task.

`Task scheduling`: i task sottomessi a un executor vengono messi in coda e prelevati dai thread del pool; l'ordine di esecuzione dipende dall'implementazione dell'executor, dalla politica della coda e dalla disponibilità dei thread. Nei scheduled executor, i task sono ordinati in base al delay di attivazione; i task periodici vengono reinseriti in coda dopo ogni esecuzione.

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

scheduler.schedule(
    () -> System.out.println("Delayed"),
    2, TimeUnit.SECONDS);
```

Metodo	Descrizione
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)	Pianifica un'azione one-shot che diventa eseguibile dopo il delay specificato.
ScheduledFuture schedule(Callable callable, long delay, TimeUnit unit)	Pianifica un task one-shot che restituisce un valore dopo il delay specificato.
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Pianifica un'esecuzione periodica a fixed rate: ogni esecuzione è pianificata rispetto al tempo iniziale; se un'esecuzione è in ritardo, le successive possono tentare di "recuperare".
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Pianifica un'esecuzione periodica con fixed delay: ogni esecuzione è pianificata rispetto al completamento della precedente; non esiste comportamento di recupero.

Important

Non creare mai thread manualmente in un loop: usa invece pools o virtual threads.

31.6 Lifecycle e Terminazione dell'Executor

Gli executor devono essere chiusi esplicitamente per rilasciare risorse e permettere la terminazione della JVM.

- **shutdown()**: Inizia shutdown ordinato: completa i task in attesa ma non accetta ulteriori task.
- **close()**: (Java 19+) chiama shutdown() e attende che i task finiscano, comportandosi come supporto try-with-resources per ExecutorService.
- **shutdownNow()**: Tenta shutdown immediato e interrompe i task in esecuzione.
- **awaitTermination()**: Attende completamento o timeout.

```
executor.shutdown();
executor.awaitTermination(5, TimeUnit.SECONDS);
```

31.7 Strategie di Thread Safety

La Concurrency API fornisce molteplici strategie complementari per ottenere thread safety.

31.7.1 Sincronizzazione

La sincronizzazione impone `mutual exclusion` e `memory visibility` usando un lock intrinseco (monitor) associato a un oggetto o a una classe.

```
synchronized void increment() {
    count++;
}
```

Quando un thread entra in un metodo synchronized:

- Acquisisce l'intrinsic lock dell'oggetto target (`this` per i metodi di istanza).
- Solo un thread alla volta può detenere lo stesso lock, prevenendo esecuzione concorrente.
- Quando il metodo termina, il lock viene rilasciato automaticamente.

La synchronization stabilisce una **happens-before relationship** nel Java Memory Model:

- Tutte le scritture fatte dentro la regione synchronized vengono flushate nella memoria principale quando il lock viene rilasciato.
- Un thread che acquisisce lo stesso lock in seguito è garantito vedere quegli update.

La keyword synchronized può essere applicata a:

- **Metodi di istanza** (lock su `this`)
- **Metodi statici** (lock sull'oggetto `Class`)
- **Blocchi** (lock su un oggetto specifico, permettendo controllo più fine)

> **Important**
>
> La sincronizzazione è semplice ma può ridurre la scalabilità sotto contention.

31.7.2 Variabili Atomiche

Le `atomic classes` forniscono operazioni lock-free, thread-safe implementate usando primitive CPU di basso livello come Compare-And-Swap (CAS).

```
AtomicInteger count = new AtomicInteger();
count.incrementAndGet();
```

31.7.2.1 Atomic classes

Atomic Class	Description
AtomicBoolean	Aggiorna e legge atomicamente un valore `boolean` .
AtomicInteger	Aggiorna e legge atomicamente un valore `int` .
AtomicLong	Aggiorna e legge atomicamente un valore `long` .
AtomicReference	Aggiorna e legge atomicamente un reference a oggetto.
AtomicIntegerArray	Fornisce operazioni atomiche sugli elementi di un array `int` .
AtomicLongArray	Fornisce operazioni atomiche sugli elementi di un array `long` .
AtomicReferenceArray	Fornisce operazioni atomiche sugli elementi di un array di reference.
AtomicStampedReference	Aggiorna atomicamente un reference con un integer stamp per evitare problemi ABA.
AtomicMarkableReference	Aggiorna atomicamente un reference con un boolean mark.

31.7.2.2 Metodi Atomici

Method	Description
get()	Restituisce il valore corrente con semantica volatile-read.
set(value)	Imposta il valore con semantica volatile-write.
lazySet(value)	Imposta eventualmente il valore con garanzie di ordering più deboli.
compareAndSet(expect, update)	Imposta atomicamente il valore se il valore corrente è uguale al valore atteso.
getAndSet(value)	Imposta atomicamente il valore e restituisce il valore precedente.
incrementAndGet()	Incrementa atomicamente il valore e restituisce il risultato aggiornato.
getAndIncrement()	Incrementa atomicamente il valore e restituisce il risultato precedente.
decrementAndGet()	Decrementa atomicamente il valore e restituisce il risultato aggiornato.
getAndDecrement()	Decrementa atomicamente il valore e restituisce il risultato precedente.
addAndGet(delta)	Aggiunge atomicamente il delta dato e restituisce il risultato aggiornato.
getAndAdd(delta)	Aggiunge atomicamente il delta dato e restituisce il risultato precedente.

Variabili Atomiche

:

- Eseguono singole operazioni **atomicamente**
- Forniscono **memory visibility guarantees** simili a `volatile`
- Evitano thread blocking, rendendole altamente scalabili sotto contention

Tuttavia, le atomic variables garantiscono atomicità solo per **operazioni individuali**.

Comporre più operazioni richiede comunque synchronization esterna.

Le variabili atomiche sono tipicamente usate per

:

- Counter e sequence generator
- Flag e state indicator
- Update ad alto throughput e bassa latenza

31.7.3 Lock Framework

Il package `java.util.concurrent.locks` fornisce meccanismi di locking espliciti che offrono maggiore flessibilità e controllo rispetto a synchronized.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

Caratteristiche chiave del Lock framework:

- I lock devono essere acquisiti e rilasciati esplicitamente

- L'acquisizione del lock può essere interruptible o time-bounded
- I lock possono essere configurati con fairness policy (parametro) quando l'ordering è richiesto (quando devi controllare l'ordine in cui i thread girano)
- Più oggetti Condition possono essere associati a un singolo lock

31.7.3.1 Lock implementations

Lock Implementation	Description
Lock	Interfaccia core che definisce operazioni di lock esplicite.
ReentrantLock	Lock reentrant di mutual exclusion con fairness policy opzionale.
ReadWriteLock	Interfaccia che definisce lock separati di read e write.
ReentrantReadWriteLock	Fornisce lock separati reentrant di read e write per migliorare la scalabilità in lettura.
StampedLock	Lock che supporta modalità optimistic, read e write locking (non-reentrant).

> **Warning**
>
> A differenza di altri lock, StampedLock **non è reentrant** — riacquisirlo dallo stesso thread causa deadlock.

31.7.3.2 Common Lock methods

Method	Description
lock()	Acquisisce il lock, bloccando indefinitamente finché disponibile.
unlock()	Rilascia il lock; deve essere chiamato dal thread proprietario.
tryLock()	Tenta di acquisire il lock immediatamente senza bloccare: restituisce boolean che indica se il lock è stato acquisito con successo
tryLock(long, TimeUnit)	Tenta di acquisire il lock entro il timeout dato.
lockInterruptibly()	Acquisisce il lock a meno che il thread sia interrotto.
newCondition()	Crea un'istanza `Condition` per coordination fine-grained tra thread.

A differenza di synchronized, i lock non vengono rilasciati automaticamente, rendendo essenziale l'uso corretto di try/finally per evitare deadlock.

31.7.4 Coordination Utilities

Le coordination utilities permettono ai thread di coordinare fasi di esecuzione senza proteggere dati condivisi tramite mutual exclusion.

Altre coordination primitives includono: - `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`:

- Blocca i thread finché un numero predefinito di thread raggiunge la barrier
- Rilascia simultaneamente tutti i thread in attesa una volta che la barrier viene tripped
- Può essere riusato per più cicli di coordination

Queste utilities si concentrano su execution ordering e synchronization, non su data protection.

31.8 Concurrent Collections

Le concurrent collections sono **thread-safe data structures** progettate per supportare **alti livelli di concorrenza** senza richiedere sincronizzazione esterna.

A differenza dei synchronized wrappers (es. `Collections.synchronizedMap`), le concurrent collections: - Usano **fine-grained locking** o **lock-free techniques** - Permettono a più thread di accedere e modificare la collection simultaneamente - Scalano meglio sotto contention

Esempi comuni includono:

- **ConcurrentHashMap**
- Una concurrent map ad alte performance che permette letture e update concorrenti partizionando lo stato interno e minimizzando lock contention.

- **CopyOnWriteArrayList**
- Una thread-safe list ottimizzata per scenari con **molte letture e poche scritture**. Le operazioni di write creano un nuovo array interno, permettendo alle letture di procedere senza locking.
- **BlockingQueue**
- Una queue progettata per **producer-consumer patterns**, dove i thread possono bloccare mentre attendono elementi o capacità disponibile.

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();
queue.put("item");    // blocks if the queue is full
queue.take();         // blocks if the queue is empty
```

Le blocking queue gestiscono la synchronization internamente, semplificando la coordination tra thread producer e consumer.

> **Caution**
>
> Le CopyOnWrite collections sono memory-expensive; evitarle in workload write-heavy.

31.9 Parallel Streams

I `parallel streams` forniscono **declarative data parallelism**, permettendo che le operazioni dello stream vengano eseguite in modo concorrente su più thread con cambiamenti minimi di codice.

Caratteristiche chiave: - Attivati tramite `parallelStream()` o `stream().parallel()` - Eseguiti internamente usando il **common ForkJoinPool** - Dividono automaticamente i dati in chunk processati in parallelo

I parallel streams funzionano meglio quando: - Le operazioni sono **CPU-bound** - Le funzioni sono **stateless e non-blocking** - La sorgente dati è abbastanza grande da ammortizzare l'overhead della parallelizzazione

```
list.parallelStream()
    .map(x -> x * x)
    .forEach(System.out::println);
```

Poiché l'ordine di esecuzione non è garantito, i parallel streams dovrebbero evitare: - Shared mutable state - Blocking I/O - Order-dependent side effects

> **Note**
>
> Usa `forEachOrdered()` se è richiesto output deterministico.

31.10 Relazione con Virtual Threads

In Java 21, l'`Executor framework` integra in modo seamless con **virtual threads**, abilitando massive concurrency con uso minimo di risorse.

```
ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor();

executor.submit(() -> blockingIO());
executor.close();
```

Questo permette al codice blocking di scalare efficientemente senza ridisegnare le API.

31.11 Summary

- La `Java Concurrency API` fornisce un'alternativa robusta, scalabile e più sicura alla gestione manuale dei thread.

- Astrarre l'esecuzione, coordinare i task e offrire utilities thread-safe permette agli sviluppatori di costruire sistemi concorrenti sia performanti sia manutenibili.
- Scegli lo strumento giusto: synchronized → locks → atomics → executors → virtual threads.

◀ 30. Thread Java – Fondamenti e Modello di Esecuzione | ▲ Index | 32. Fondamenti di File e Path ▶

Module 08

Java I/O and NIO

32. Fondamenti di File e Path

Indice

- 32.1 Modello Concettuale: Filesystem, File, Directory, Link e Target-di-I/O
- 32.2 Filesystem – L'Astrazione Globale
- 32.3 Path – Localizzare una Entry in un Filesystem
- 32.4 File – Contenitori Persistenti di Dati
- 32.5 Directory – Contenitori Strutturali
- 32.6 Link – Meccanismi di Indirezione
  - 32.6.1 Hard Link
  - 32.6.2 Link Simbolici Soft
- 32.7 Altri Tipi di Entry del Filesystem
- 32.8 Come Java IO Interagisce con Questi Concetti
- 32.9 Trappole Concettuali Fondamentali
- 32.10 Perché Esistono Path e Files (Contesto-IO)
- 32.11 File È (API legacy) sia un path sia una api-di-operazioni-su-file
  - 32.11.1 Cosa È Davvero File
  - 32.11.2 Responsabilità tipo-Path
  - 32.11.3 Responsabilità di Operazioni sul Filesystem
  - 32.11.4 Cosa NON È File
  - 32.11.5 Il vecchio doppio ruolo
  - 32.11.6 Come NIO Ha Risolto Questo
  - 32.11.7 Riepilogo
- 32.12 Path È una Descrizione, Non una Risorsa
- 32.13 Path Assoluti vs Relativi
  - 32.13.1 Path Assoluti
  - 32.13.2 Path Relativi
- 32.14 Consapevolezza del Filesystem e Separatori
  - 32.14.1 FileSystem
  - 32.14.2 Separatori di Path
- 32.15 Cosa Fa Davvero Files e Cosa Non Fa
  - 32.15.1 Files FA
  - 32.15.2 Files NON FA
- 32.16 Filosofia di Gestione degli Errori: Old-vs-NIO
- 32.17 Falsi Miti Comuni

Questa sezione si concentra su `Path`, `File`, `Files` e classi correlate, spiegando perché esistono, quali problemi risolvono e quali sono le differenze tra le API legacy `java.io` e `NIO v.2` (nuove API di I/O), con particolare attenzione alla semantica del filesystem, alla risoluzione dei path e ai falsi miti comuni.

32.1 Modello Concettuale: Filesystem, File, Directory, Link e Target-di-I/O

Prima di comprendere le API di Java I/O, è essenziale capire con cosa esse interagiscono.

Java I/O non opera nel vuoto: interagisce con astrazioni di filesystem fornite dal sistema operativo.

Questa sezione definisce questi concetti indipendentemente da Java, poi spiega come Java I/O li mappa e quali problemi vengono risolti.

32.2 Filesystem – L'Astrazione Globale

Un `filesystem` è un meccanismo strutturato fornito da un sistema operativo per organizzare, memorizzare, recuperare e gestire dati su dispositivi di storage persistente.

A livello concettuale, un filesystem risolve diversi problemi fondamentali:

- Storage persistente oltre l'esecuzione del programma
- Organizzazione gerarchica dei dati
- Nominare e localizzare i dati
- Controllo degli accessi e permessi
- Garanzie di concorrenza e consistenza

In Java NIO, un filesystem è rappresentato dall'astrazione `FileSystem`, tipicamente ottenuta tramite `FileSystems.getDefault()` per il filesystem del sistema operativo.

Aspetto	Significato
Persistenza	I dati sopravvivono alla terminazione della JVM
Ambito	Gestito dal SO, non gestito dalla JVM
Molteplicità	Possono esistere più filesystem
Esempi	Disk FS, ZIP FS, in-memory FS

> **Note**
>
> Java non implementa filesystem; si adatta a implementazioni di filesystem fornite dal SO o da providers custom.

32.3 Path – Localizzare una Entry in un Filesystem

Un `path` è un localizzatore logico, non una risorsa.

Descrive dove qualcosa sarebbe in un filesystem, non cos'è o se esiste.

Un `path` risolve il problema dell' `addressing`:

- Identifica una posizione
- È interpretato all'interno di un filesystem specifico
- Può o non può corrispondere a una entry esistente

Proprietà	Path
Consapevole dell'esistenza	No
Consapevole del tipo	No
Immutabile	Sì
Risorsa del SO	No

> **Note**
>
> In Java, `Path` rappresenta entry potenziali del filesystem, non entry reali.

32.4 File – Contenitori Persistenti di Dati

Un `file` è una entry del filesystem il cui ruolo primario è memorizzare dati.

Il filesystem tratta i file come sequenze di byte opache.

Problemi risolti dai file:

- Storage duraturo di informazioni
- Accesso sequenziale e random ai dati
- Condivisione dei dati tra processi

Dal punto di vista del filesystem, un file ha:

- Contenuto (byte)
- Metadati (dimensione, timestamp, permessi)
- Una posizione (path)

Aspetto	Descrizione
Contenuto	Orientato ai byte
Interpretazione	Definita dall'applicazione
Durata	Indipendente dai processi
Accesso Java	Stream, channel, metodi di Files

> **Note**
>
> `Testo` vs `binario` non è un concetto del filesystem; è un'interpretazione a livello applicazione.

32.5 Directory – Contenitori Strutturali

Una `directory (o folder)` è una entry del filesystem il cui scopo è organizzare altre entry.

Le `directory` risolvono il problema della scalabilità e dell'organizzazione:

- Raggruppare entry correlate
- Abilitare naming gerarchico
- Supportare lookup efficiente

Aspetto	Directory
Memorizza dati	No (memorizza riferimenti)
Contiene	File, directory, link
Lettura/scrittura	Strutturale, non basata sul contenuto
Accesso Java	Files.list, Files.walk

> **Note**
>
> Una directory non è un file con contenuto, anche se entrambi condividono metadati comuni.

32.6 Link – Meccanismi di Indirezione

Un `link` è una entry del filesystem che riferisce un'altra entry.

I link risolvono il problema dell'indirezione e del riuso.

32.6.1 Hard Link

Un `hard link` è un nome aggiuntivo per gli stessi dati sottostanti.

- Più path puntano agli stessi dati del file
- La cancellazione avviene solo quando tutti i link vengono rimossi

32.6.2 Link Simbolici (Soft)

Un `link simbolico` è un file speciale che contiene un path verso un'altra entry:

- Può puntare a target non esistenti
- Risolto al momento dell'accesso

Tipo di Link	Riferisce	Può essere dangling	Gestione Java
Hard	Dati	No	Trasparente
Simbolico	Path	Sì	Controllo esplicito

> **Note**
>
> Java NIO espone il comportamento dei link in modo esplicito tramite `LinkOption`.
>
> In molti filesystem comuni, il codice Java non può creare hard link in modo pienamente portabile, mentre i link simbolici sono supportati direttamente tramite `Files.createSymbolicLink(...)` (dove permesso dal SO / permessi).

32.7 Altri Tipi di Entry del Filesystem

Alcune entry del filesystem non sono contenitori di dati ma endpoint di interazione.

Tipo	Scopo
Device file	Interfaccia verso hardware
FIFO / Pipe	Comunicazione tra processi
Socket file	Comunicazione di rete

> **Note**
>
> Java I/O può interagire con queste entry, ma il comportamento dipende dalla piattaforma.

32.8 Come Java IO Interagisce con Questi Concetti

Le API Java I/O operano a diversi livelli di astrazione:

- `Path` (NIO) / `File` (API legacy) → descrive una entry del filesystem
- `File` (API legacy) / `Files` → interroga o modifica lo stato del filesystem
- `Streams` / `Channels` → muovono byte o caratteri

API Java	Ruolo
`Path`	Addressing
`File` (API legacy)	Addressing / operazioni su filesystem
`Files`	Operazioni su filesystem
`InputStream` / `Reader`	Lettura dati
`OutputStream` / `Writer`	Scrittura dati
`Channel` / `SeekableByteChannel`	Avanzato / accesso random

> **Note**
>
> Nessuna API Java "è" un file; le API mediano l'accesso a risorse gestite dal filesystem.

32.9 Trappole Concettuali Fondamentali

- Confondere i path con i file
- Assumere che i path implichino esistenza
- Assumere che le directory memorizzino i dati dei file
- Assumere che i link siano sempre risolti automaticamente

> **Note**
>
> Separare sempre posizione, struttura e flusso dati quando si ragiona su I/O.

32.10 Perché Esistono Path e Files (Contesto-IO)

Il classico `java.io` mescolava tre compiti diversi in un API poco specifica:

- Rappresentazione del path (dove si trova la risorsa?)
- Interazione con il filesystem (esiste? che tipo?)
- Accesso ai dati (lettura/scrittura di byte o caratteri)

Il design di NIO.2 (Java 7+) separa deliberatamente queste responsabilità:

- `Path` → descrive una posizione
- `Files` → esegue operazioni sul filesystem
- `Streams / Channels` → spostano dati

> **Note**
>
> Un `Path` non apre mai un file e non tocca mai il disco da solo.

32.11 File È (API legacy) sia un path sia una api-di-operazioni-su-file?

Sì — nella vecchia API di I/O, `java.io.File` gioca in modo confuso due ruoli allo stesso tempo, e questo design è esattamente una delle ragioni per cui è stato introdotto `java.nio.file`.

Risposta Breve

- `File` rappresenta un path del filesystem
- `File` espone anche operazioni sul filesystem

- Non rappresenta **né** un file aperto, **né** i contenuti del file

> **Note**
>
> Questo mix di responsabilità è considerato un difetto di design.

32.11.1 Cosa È Davvero File

Concettualmente, `File` è più vicino a ciò che oggi chiamiamo `Path`, ma con metodi operativi aggiunti.

Aspetto	java.io.File
Rappresenta una posizione	Sì
Apre un file	No
Legge / scrive dati	No
Interroga il filesystem	Sì
Modifica il filesystem	Sì
Contiene handle del SO	No

> **Note**
>
> Un oggetto `File` può esistere anche se il file non esiste.

32.11.2 Responsabilità tipo-Path

`File` si comporta come un'astrazione di path perché:

- Incapsula un pathname del filesystem (assoluto o relativo)
- Può essere risolto rispetto alla working directory
- Può essere convertito in forma assoluta o canonica

Esempi:

```
File f = new File("data.txt"); // path relativo
File abs = f.getAbsoluteFile(); // path assoluto
File canon = f.getCanonicalFile(); // normalizzato + risolto
```

32.11.3 Responsabilità di Operazioni sul Filesystem

Allo stesso tempo, `File` espone metodi che toccano il filesystem:

- exists()
- isFile(), isDirectory()
- length()
- delete()
- mkdir(), mkdirs()
- list(), listFiles()

> **Note**
>
> La maggior parte di questi metodi restituisce `boolean` invece di lanciare `IOException`, il che nasconde le cause degli eventuali problemi.

32.11.4 Cosa NON È File

- Non è un file descriptor aperto
- Non è uno stream
- Non è un channel

- Non è un contenitore di dati del file

Si devono comunque usare stream o reader/writer per accedere ai contenuti.

32.11.5 Il vecchio doppio ruolo

Il doppio ruolo di `File` ha causato diversi problemi:

- Ruolo misto (path + operazioni)
- Gestione errori insufficiente (boolean invece di eccezioni)
- Supporto debole per link e filesystem multipli
- Comportamento dipendente dalla piattaforma

32.11.6 Come NIO ha Risolto Questo

NIO.2 separa esplicitamente le responsabilità:

Responsabilità	Vecchia API	API NIO
`Rappresentazione Path`	`File`	`Path`
`Operazioni su filesystem`	`File`	`Files`
`Accesso ai dati`	Stream	Stream / Channel

> **Note**
>
> Questa separazione è uno dei miglioramenti concettuali più importanti in Java I/O.

32.11.7 Riepilogo

- `File` rappresenta un path E svolge operazioni sul filesystem
- Non legge né scrive mai i contenuti del file
- Non apre mai un file
- `Path` + `Files` è il sostituto moderno

32.12 Path È una Descrizione, Non una Risorsa

Un `Path` è un'astrazione pura che rappresenta una sequenza di elementi nominati in un filesystem.

- Non implica esistenza
- Non implica accessibilità
- Non contiene un file descriptor

Questo è fondamentalmente diverso da stream o channel.

Concetto	Path	Stream / Channel
`Apre risorsa`	No	Sì
`Tocca disco`	No	Sì
`Contiene handle SO`	No	Sì
`Immutabile`	Sì	No

> **Note**
>
> Creare un Path non può lanciare `IOException` perché non avviene alcun I/O.

32.13 Path Assoluti vs Relativi

Comprendere la risoluzione dei path è essenziale.

32.13.1 Path Assoluti

Un path assoluto identifica interamente una posizione dalla root del filesystem.

- Root dipendente dalla piattaforma
- Indipendente dalla JVM working directory

Piattaforma	Esempio di Path Assoluto
Unix	`/home/user/file.txt`
Windows	`C:\Users\User\file.txt`

Important

- Un path che inizia con una slash `(/)` (tipo Unix) o con una lettera di drive come `C:` (Windows) è **tipicamente** considerato un path assoluto.
- Il simbolo `.` è un riferimento alla directory corrente mentre `..` è un riferimento alla directory padre. Su Windows, un path come `\dir\file.txt` (senza lettera di drive) è *rooted* sul drive corrente, non pienamente qualificato con drive + path.

Esempio:

```
/dirA/dirB/../dirC/./content.txt

is equivalent to:

/dirA/dirC/content.txt

// in this example the symbols were redundant and unnecessary
```

32.13.2 Path Relativi

Un path relativo viene risolto rispetto alla directory di lavoro corrente della JVM.

- Dipende da dove è stata avviata la JVM
- Fonte comune di bug

Note

La working directory è tipicamente disponibile tramite `System.getProperty("user.dir")`.

Esempio:

```
dirB/dirC/content.txt
```

32.14 Consapevolezza del Filesystem e Separatori

NIO introduce l'astrazione del filesystem, che era in gran parte assente in java.io.

32.14.1 FileSystem

Un `FileSystem` rappresenta una specifica implementazione concreta di filesystem.

- Il filesystem di default corrisponde al filesystem del SO
- Possibili altri filesystem (ZIP, memoria, rete)

> **Note**
>
> I path sono sempre associati a esattamente UN FileSystem.

32.14.2 Separatori di Path

I separatori differiscono tra piattaforme, ma `Path` li astrae.

Aspetto	java.io.File	java.nio.file.Path
Separatore	Basato su stringhe	Consapevole del filesystem
Portabilità	Gestione manuale	Automatica
Comparazione	Soggetta a errori	Più sicura

> **Note**
>
> Hardcodare `"/"` o `"\\"` è sconsigliato; `Path` lo gestisce automaticamente.

32.15 Cosa Fa Davvero Files e Cosa Non Fa

La classe `Files` esegue vere operazioni di I/O.

32.15.1 Files FA

- Apre file indirettamente (tramite stream / channel restituiti dai suoi metodi)
- Crea e cancella entry del filesystem
- Lancia eccezioni checked in caso di fallimento
- Rispetta i permessi del filesystem

32.15.2 Files NON FA

- Mantenere risorse aperte dopo il ritorno del metodo (eccetto gli stream)
- Memorizzare contenuti del file internamente
- Garantire atomicità se non specificato
- Mantenere un handle persistente a file aperti (sono stream/channel a possedere l'handle)

> **Note**
>
> I metodi che restituiscono stream (es. `Files.lines()`) tengono il file aperto finché lo stream non viene chiuso.

32.16 Filosofia di Gestione degli Errori: Old vs NIO

Una grande differenza concettuale risiede nel reporting degli errori.

Aspetto	`java.io.File`	`java.nio.file.Files`
Segnalazione errori	boolean / `null`	`IOException`
Diagnostica	Scarsa	Ricca
Consapevolezza race	Debole	Migliorata
Preferenza	Sconsigliato	Preferito

32.17 Falsi Miti Comuni

- “Path rappresenta un file” → falso
- “normalize controlla l’esistenza” → falso
- “Files.readAllLines streamma i dati” → falso
- “I path relativi sono portabili” → falso
- “Creare un Path può fallire per permessi” → falso

> **Note**
>
> Molti metodi NIO che suonano “sicuri” sono puramente sintattici (come `normalize` o `resolve`): non toccano il filesystem e non possono rilevare file mancanti.

◀ 31. Java Concurrency APIs | ▲ Index | 33. API di Files e Path ▶

33. API di Files e Path

Indice

- 33.1 Creazione-e-Conversione di File legacy e Path NIO
  - 33.1.1 Creare un File Legacy
  - 33.1.2 Creare un Path NIO-v2
  - 33.1.3 Assoluto vs Relativo Cosa Significa Relativo
  - 33.1.4 Unire–Costruire-Path
    - 33.1.4.1 resolve
    - 33.1.4.2 relativize
  - 33.1.5 Convertire tra File e Path
  - 33.1.6 Conversione URI Quando-Serve
  - 33.1.7 Canonico vs Assoluto vs Normalizzato Differenze-Fondamentali
    - 33.1.7.1 normalize
  - 33.1.8 Tabella di Confronto Rapida Creazione–Conversione
- 33.2 Gestire File e Directory: Creare-Copiare-Spostare-Sostituire-Confrontare-Cancellare
  - 33.2.1 Modello Mentale Path-Locator-vs-Operazioni
  - 33.2.2 Creare File e Directory
    - 33.2.2.1 Creare un File
    - 33.2.2.2 Creare Directory
  - 33.2.3 Copiare File e Directory
    - 33.2.3.1 Copiare un File NIO
    - 33.2.3.2 Copia Manuale Legacy-Basata-su-Stream
  - 33.2.4 Spostare–Rinominare-e-Sostituire
    - 33.2.4.1 Rinomina Legacy Trappola-Comune
    - 33.2.4.2 NIO Move Preferito
  - 33.2.5 Confrontare Path e File
    - 33.2.5.1 Uguaglianza-vs-Stesso-File
  - 33.2.6 Cancellare File e Directory
    - 33.2.6.1 Delete Legacy
    - 33.2.6.2 NIO Delete e Delete-If-Exists
  - 33.2.7 Copiare–Cancellare-Ricorsivamente-Alberi-di-Directory Pattern-NIO
  - 33.2.8 Checklist di Riepilogo

Questa sezione si concentra su come creare localizzatori su filesystem usando la API legacy `java.io.File` e la moderna API `java.nio.file.Path`: come convertire tra di loro e comprendere overload, default e trappole comuni.

33.1 `File` legacy e `Path` NIO: Creazione e Conversione

33.1.1 Creare un `File` (Legacy)

Una istanza `File` rappresenta un pathname del filesystem (assoluto o relativo).

Crearne una non accede al filesystem e non lancia `IOException`.

Costruttori core (più comuni):

- `new File(String pathname)`
- `new File(String parent, String child)`
- `new File(File parent, String child)`
- `new File(URI uri)` (tipicamente `file:...`)

```
import java.io.File;
import java.net.URI;

File f1 = new File("data.txt"); // relativo
File f2 = new File("/tmp", "data.txt"); // parent + child
File f3 = new File(new File("/tmp"), "data.txt");

File f4 = new File(URI.create("file:///tmp/data.txt"));
```

> **Note**
>
> - `new File(...)` non apre mai il file.
> - Esistenza/permessi vengono controllati solo quando invocati metodi come `exists()`, `length()`, o quando si apre uno stream/channel.

33.1.2 Creare un `Path` (NIO v.2)

Un `Path` è solo un locator.

Come `File`, creare un `Path` non accede al filesystem.

Factory core:

- `Path.of(String first, String... more)` (Java 11+)
- `Paths.get(String first, String... more)` (stile più vecchio; ancora valido)
- `Path.of(URI uri)` (es. `file:///...`)

```
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.Paths;

Path p1 = Path.of("data.txt"); // relativo
Path p2 = Path.of("/tmp", "data.txt"); // parent + child

Path p3 = Paths.get("data.txt"); // stile factory legacy
Path p4 = Path.of(URI.create("file:///tmp/data.txt"));
```

> **Note**
>
> - `Path.of(...)` e `Paths.get(...)` sono equivalenti per il filesystem di default.
> - Preferisci `Path.of` nel codice moderno.

33.1.3 Assoluto vs Relativo: Cosa Significa “Relativo”

Sia `File` sia `Path` possono essere creati come path relativi.

I path relativi vengono risolti rispetto alla working directory del processo (tipicamente `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**
>
> I path relativi sono una fonte comune di bug "funziona sulla mia macchina" perché `user.dir` dipende da come/dove la JVM è stata lanciata.

33.1.4 Unire / Costruire Path

- Il `File` legacy usa i costruttori (parent + child).
- NIO usa `resolve` e metodi correlati.

Task	Legacy (File)	NIO (Path)
Unire parent + child	`new File(parent, child)`	`parent.resolve(child)`
Unire molti segmenti	Costruttori ripetuti	`Path.of(a, b, c)` o `resolve()` concatenati

```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()`

Combina path in modo filesystem-aware.

- I path relativi vengono appesi
- Un argomento assoluto sostituisce il base path

> **Note**
>
> `Path.resolve(...)` ha una regola: se l'argomento è assoluto, restituisce l'argomento e scarta la base (non puoi combinare due path assoluti usando `resolve`).

33.1.4.2 `relativize()`

`Path.relativize` calcola un **path relativo** da un path a un altro. Il path risultante, quando `resolved` rispetto al path sorgente, produce il path target.

In altre parole:

- Risponde alla domanda: "Come vado dal path A al path B?"
- Il risultato è sempre un path **relativo**
- Non avviene alcun accesso al filesystem

Regole Fondamentali

`relativize` ha precondizioni strette. Violandole si lancia una eccezione.

Regola	Spiegazione
Entrambi i path devono essere assoluti	o entrambi relativi
Entrambi i path devono appartenere allo stesso filesystem	stesso provider
I componenti di root devono combaciare	stessa root (su Windows, stesso drive)
Il risultato non è mai assoluto	sempre relativo

> **Note**
>
> Se un path è assoluto e l'altro relativo, viene lanciata `IllegalArgumentException`.

Esempio Relativo Semplice:

Entrambi i path sono relativi, quindi la relativizzazione è consentita.

```
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
```

Interpretazione: da `docs/manual`, sali di un livello, poi entra in `images/logo.png`.

Esempio di Path Assoluti:

I path assoluti funzionano esattamente allo stesso modo.

```
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
```

Usare `resolve` per Verificare il Risultato

Una proprietà chiave di `relativize` è questa identità:

```
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
```

Esempio: Mescolare Path Assoluti e Relativi (CASO ERRORE)

Questo è uno degli errori più comuni.

```
Path abs = Path.of("/a/b");
Path rel = Path.of("c/d");

abs.relativize(rel); // lancia eccezione
```

```
Exception in thread "main" java.lang.IllegalArgumentException
```

> **Note**
>
> `relativize` NON tenta di convertire automaticamente i path in assoluti.

Esempio: Root Diverse (Trappola Specifica Windows)

Su Windows, path con lettere di drive diverse non possono essere relativizzati.

```
Path p1 = Path.of("C:\\data\\a");
Path p2 = Path.of("D:\\data\\b");

p1.relativize(p2); // IllegalArgumentException
```

> **Note**
>
> Su sistemi Unix-like, la root è sempre `/`, quindi questo problema non si verifica.

33.1.5 Convertire tra `File` e `Path`

La conversione è diretta e lossless per normali path del filesystem locale.

Conversione	Come
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 conversione non valida l'esistenza. Converte solo rappresentazioni.

33.1.6 Conversione URI (Quando Serve)

Gli `URI` sono utili quando i path devono essere rappresentati in forma standard e assoluta (es. interoperare con risorse in rete o configurazione).

Entrambe le API supportano la conversione URI.

Direzione	Legacy (File)	NIO (Path)
Da URI	`new File(uri)`	`Path.of(uri)`
A 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)` richiede un URI `file:` e lancia `IllegalArgumentException` se l'URI non è gerarchico o non è un file URI.

33.1.7 Canonico vs Assoluto vs Normalizzato (Differenze Fondamentali)

Questi termini vengono spesso confusi. Non sono la stessa cosa.

Concetto	Legacy (File)	NIO (Path)	Tocca filesystem
Assoluto	`getAbsoluteFile()`	`toAbsolutePath()`	No
Normalizzato	(nessun normalize puro, usa canonical)*	`normalize()`	`normalize()`: No
Canonico / Reale	`getCanonicalFile()`	`toRealPath()`	Sì

> **Note**
>
> `File.getCanonicalFile()` e `Path.toRealPath()` possono risolvere symlink e richiedere che il path esista, quindi possono lanciare `IOException`.
>
> File non fornisce un metodo per una normalizzazione puramente sintattica: storicamente molti sviluppatori usavano getCanonicalFile(), ma questo accede al filesystem e può fallire.

```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()); // assoluto, può ancora contenere ".."

try {
    System.out.println(f.getCanonicalPath()); // risolve "..", può toccare filesystem
} catch (IOException e) {
    System.out.println("Canonical failed: " + e.getMessage());
}

Path p = Path.of("a/../data.txt");
System.out.println(p.toAbsolutePath()); // assoluto, può ancora contenere ".."
System.out.println(p.normalize()); // puramente sintattico

try {
    System.out.println(p.toRealPath()); // risolve symlink, richiede esistenza
} catch (IOException e) {
    System.out.println("RealPath failed: " + e.getMessage());
}
```

33.1.7.1 `normalize()`

Rimuove elementi di nome **ridondanti** come `.` e `..`.

- Puramente sintattico

- Non controlla se il path esiste

> **Note**
>
> `normalize()` è puramente sintattico, non controlla l'esistenza, e può produrre path invalidi se usato male.

33.1.8 Tabella di Confronto Rapida (Creazione + Conversione)

Esigenza	Legacy (File)	NIO (Path)	Preferito oggi
Creare da stringa	`new File("x")`	`Path.of("x")`	Path
Parent + child	`new File(p, c)`	`Path.of(p, c)` o `resolve()`	Path
Convertire tra API	`toPath()`	`toFile()`	Path-centric
Normalizzare	`getCanonicalFile()` (basato su filesystem)	`normalize()` (solo sintattico)	Path
Risolvere symlink	Canonical	`toRealPath()`	Path

33.2 Gestire File e Directory: Creare, Copiare, Spostare, Sostituire, Confrontare, Cancellare (Legacy vs NIO)

Questa sezione copre le operazioni che esegui sulle entry del filesystem (file/directory): creazione, copia, spostamento/rinominazione, sostituzione, confronto e cancellazione.

Confronta il legacy `java.io.File` (e helper legacy correlati) con il moderno `java.nio.file` (NIO.2).

33.2.1 Modello Mentale: "Path/Locator" vs "Operazioni"

Entrambe le API usano oggetti che rappresentano un path, ma le operazioni differiscono:

- Legacy: `File` è sia un wrapper di path sia una API di operazioni (responsabilità mescolata)
- NIO: `Path` è il path; `Files` esegue le operazioni (separazione delle responsabilità)

Responsabilità	Legacy	NIO
Rappresentazione Path	`File`	`Path`
Operazioni su filesystem	`File`	`Files`
Reporting degli errori	Debole (boolean)	Forte (eccezioni)

> **Note**
>
> I metodi legacy spesso restituiscono `boolean` (fallimento silenzioso), mentre NIO lancia `IOException` con causa.

33.2.2 Creare File e Directory

La creazione è dove la vecchia API è più scomoda e la API NIO è più espressiva.

Task	Approccio legacy	Approccio NIO	Note
Creare file vuoto	apri+chiudi stream	`Files.createFile`	NIO fallisce se esiste
Creare una directory	`mkdir`	`Files.createDirectory`	Il parent deve esistere
Creare directory ricorsivamente	`mkdirs`	`Files.createDirectories`	Crea i parent

33.2.2.1 Creare un File

Il legacy non ha un metodo "crea file vuoto", quindi tipicamente crei un file aprendo uno stream di output (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)) {
    // il file è creato (o troncato) come side effect
}
```

NIO fornisce un metodo esplicito di creazione.

```
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` lancia `FileAlreadyExistsException` se la entry esiste.

33.2.2.2 Creare Directory

```
import java.io.File;

File dir1 = new File("a/b");
boolean ok1 = dir1.mkdir(); // fallisce se il parent "a" non esiste
boolean ok2 = dir1.mkdirs(); // crea i parent
```

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;

Path d = Path.of("a/b");
Files.createDirectory(d); // il parent deve esistere
Files.createDirectories(d); // crea i parent, ok se già esiste
```

> **Note**
>
> I legacy `mkdir()/mkdirs()` restituiscono `false` in caso di fallimento senza dire perché. NIO lancia `IOException`.

33.2.3 Copiare File e Directory

La copia legacy è di solito una copia manuale via stream (o librerie esterne). NIO ha una singola operazione esplicita.

Capacità	Legacy	NIO
Copiare contenuti file	Stream manuali	`Files.copy`
Copiare in target esistente	Manuale	Opzione `REPLACE_EXISTING`
Copiare albero directory	Ricorsione manuale	Ricorsione manuale (ma strumenti migliori: `Files.walk` + `Files.copy`)

33.2.3.1 Copiare un File (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); // fallisce se dst esiste
Files.copy(src, dst, StandardCopyOption.REPLACE_EXISTING);
```

> **Note**
>
> `Files.copy` lancia `FileAlreadyExistsException` se il target esiste e non hai usato `REPLACE_EXISTING`.

33.2.3.2 Copia Manuale (Legacy, Basata su 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**
>
> Ricorda `read(byte[])` restituisce il numero di byte letti; devi scrivere solo quel conteggio, non l'intero buffer.

33.2.4 Spostare / Rinominare e Sostituire

In entrambe le API, rinomina/sposta è "a livello metadata" quando possibile, ma può comportarsi come copy+delete tra filesystem. NIO lo rende esplicito tramite opzioni.

Operazione	Legacy	NIO
Rinominare/spostare	`File.renameTo`	`Files.move`
Sostituire esistente	Inaffidabile	`REPLACE_EXISTING`
Spostamento atomico	Non supportato	`ATOMIC_MOVE` (se supportato)

33.2.4.1 Rinomina Legacy (Trappola Comune)

```java
import java.io.File;

File from = new File("old.txt");
File to = new File("new.txt");

boolean ok = from.renameTo(to); // può fallire silenziosamente
System.out.println(ok);
```

> **Note**
>
> - `renameTo` è notoriamente platform-dependent e restituisce solo `boolean`.
> - Può fallire perché il target esiste, il file è aperto, permessi, o spostamento cross-filesystem.

33.2.4.2 NIO Move (Preferito)

```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); // fallisce se il target esiste
Files.move(from, to, StandardCopyOption.REPLACE_EXISTING);
```

> **Note**
>
> `Files.move` lancia `FileAlreadyExistsException` quando il target esiste e `REPLACE_EXISTING` non è specificato.

33.2.5 Confrontare Path e File

Confrontare locator può significare: uguaglianza string/path, uguaglianza normalizzata/canonica, o "stesso file su disco".

Le API differiscono significativamente qui.

Obiettivo confronto	Legacy	NIO
Stesso testo di path	`File.equals`	`Path.equals`
Normalizzare path	`getCanonicalFile`	`normalize`
Stesso file/risorsa su disco	debole (euristica canonica)	`Files.isSameFile`

33.2.5.1 Uguaglianza vs Stesso File

Due stringhe di path diverse possono riferirsi allo stesso file.

```
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 (testo path diverso)
System.out.println(p1.normalize().equals(p2.normalize())); // può ancora essere false se relat

try {
    System.out.println(Files.isSameFile(p1, p2)); // può essere true, può lanciare se non acce
} catch (IOException e) {
    System.out.println("isSameFile failed: " + e.getMessage());
}
```

> **Note**
>
> `Files.isSameFile` può accedere al filesystem e può lanciare `IOException` (problemi di permessi, file mancanti, ecc.).

33.2.6 Cancellare File e Directory

La cancellazione è semplice in concetto ma ha casi limite importanti: directory non vuote, target mancanti e differenze nel reporting degli errori.

Task	Legacy	NIO	Comportamento se mancante
Cancellare file/dir	`File.delete`	`Files.delete`	Legacy false, NIO eccezione
Cancellare se esiste	Nessun diretto (check+delete)	`Files.deleteIfExists`	restituisce boolean
Cancellare dir non vuota	Ricorsione manuale	Ricorsione manuale (walk)	Entrambe richiedono ricorsione

33.2.6.1 Delete Legacy

```
import java.io.File;

File f = new File("x.txt");
boolean ok = f.delete(); // false se non cancellato
System.out.println(ok);
```

> **Note**
>
> Legacy `delete()` fallisce (restituisce false) per una directory non vuota e spesso non fornisce motivo.

33.2.6.2 NIO Delete e 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` lancia `NoSuchFileException` se mancante, mentre `deleteIfExists` restituisce `false`.

33.2.7 Copiare / Cancellare Ricorsivamente Alberi di Directory (Pattern NIO)

NIO non fornisce un singolo metodo “copyTree/deleteTree”, ma l’approccio standard usa `Files.walk` o `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**
>
> Cancellare un albero di directory richiede cancellare prima i file, poi le directory (post-order). Questa è una domanda di ragionamento comune.

33.2.8 Checklist di Riepilogo

- Preferisci `Files.createFile/createDirectory/createDirectories` rispetto a workaround legacy
- `File.renameTo` è inaffidabile; preferisci `Files.move` con opzioni
- `Files.copy/move` lanciano `FileAlreadyExistsException` a meno che non venga usato `REPLACE_EXISTING`
- `Files.delete` lancia; `Files.deleteIfExists` restituisce boolean

- `Files.isSameFile` può lanciare `IOException` e può toccare il filesystem
- La cancellazione di directory non vuote richiede ricorsione (entrambe le API)

◀ 32. Fondamenti di File e Path | ▲ Index | 34. Stream I/O in Java ▶

34. Stream I/O in Java

Indice

- 34.1 Che cos'è uno Stream I/O in Java
- 34.2 Stream di Byte vs Stream di Caratteri
  - 34.2.1 Stream di Byte
  - 34.2.2 Stream di Caratteri
  - 34.2.3 Tabella di riepilogo
- 34.3 Stream di Basso Livello vs Stream di Alto Livello
  - 34.3.1 Stream di Basso Livello Node-Streams
  - 34.3.2 Stream comuni di Basso Livello
  - 34.3.3 Stream di Alto Livello Filter–Processing-Streams
  - 34.3.4 Stream comuni di Alto Livello
  - 34.3.5 Regole di chaining degli stream e errori comuni
    - 34.3.5.1 Regola fondamentale di chaining
    - 34.3.5.2 Incompatibilità tra stream di byte e stream di caratteri
    - 34.3.5.3 Chaining non valido errore-di-compilazione
    - 34.3.5.4 Bridging da stream di byte a stream di caratteri
    - 34.3.5.5 Pattern corretto di conversione
    - 34.3.5.6 Regole di ordinamento nelle catene di stream
    - 34.3.5.7 Ordine logico corretto
    - 34.3.5.8 Regola di gestione delle risorse
    - 34.3.5.9 Trappole comuni
- 34.4 Classi base principali di javaio e metodi chiave
  - 34.4.1 InputStream
    - 34.4.1.1 Metodi chiave
    - 34.4.1.2 Esempio tipico di utilizzo
  - 34.4.2 OutputStream
    - 34.4.2.1 Metodi chiave
    - 34.4.2.2 Esempio tipico di utilizzo
  - 34.4.3 Reader e Writer
    - 34.4.3.1 Gestione del charset
- 34.5 Stream bufferizzati e prestazioni
  - 34.5.1 Perché il buffering conta
  - 34.5.2 Come funziona la lettura non bufferizzata
  - 34.5.3 Come funziona BufferedInputStream
  - 34.5.4 Esempio di output bufferizzato
  - 34.5.5 BufferedReader vs Reader
  - 34.5.6 Esempio di BufferedWriter
- 34.6 java io vs java nio e java nio file
  - 34.6.1 Differenze concettuali
  - 34.6.2 java-nio I/O file moderno
- 34.7 Quando usare quale API
- 34.8 Trappole comuni e suggerimenti

Questo capitolo fornisce una spiegazione dettagliata degli `stream I/O in Java`.

Copre gli stream classici **java.io**, li mette a confronto con **java.nio / java.nio.file**, e spiega principi di progettazione, API, casi limite e distinzioni rilevanti.

34.1 Che cos'è uno Stream I/O in Java?

Uno `stream I/O` rappresenta un flusso di dati tra un programma Java e una sorgente o destinazione esterna.

I dati scorrono in modo sequenziale, come acqua in un tubo.

- Uno stream non è una struttura dati; non memorizza dati in modo permanente
- Gli stream sono unidirezionali (input O output)
- Gli stream astraggono la sorgente sottostante (file, rete, memoria, dispositivo)
- Gli stream operano in modo bloccante, sincrono (I/O classico)

In Java, gli stream sono organizzati attorno a due dimensioni principali:

- `Direzione`: Input vs Output
- `Tipo di dato`: Byte vs Carattere

34.2 Stream di Byte vs Stream di Caratteri

Java distingue gli stream in base all'unità di dato che elaborano.

34.2.1 Stream di Byte

- Lavorano con byte grezzi a 8 bit
- Usati per dati binari (immagini, audio, PDF, ZIP)
- Classi base: `InputStream` e `OutputStream`

34.2.2 Stream di Caratteri

- Lavorano con caratteri Unicode a 16 bit
- Gestiscono automaticamente l'encoding dei caratteri
- Classi base: `Reader` e `Writer`

34.2.3 Tabella di riepilogo

Aspetto	Stream di Byte	Stream di Caratteri
`Unità di dato`	byte (8 bit)	char (16 bit)
`Gestione encoding`	Nessuna	Sì (consapevole del charset)
`Classi base`	InputStream / OutputStream	Reader / Writer
`Uso tipico`	File binari	File di testo
`Focus`	I/O a basso livello	Elaborazione testo

34.3 Stream di Basso Livello vs Stream di Alto Livello

Gli stream in `java.io` seguono un pattern decorator. Gli stream vengono impilati per aggiungere funzionalità.

34.3.1 Stream di Basso Livello (Node Streams)

Gli stream di basso livello si collegano direttamente a una sorgente o a una destinazione di dati.

- Sanno come leggere/scrivere byte o caratteri
- NON forniscono buffering, formattazione o gestione di oggetti

34.3.2 Stream comuni di Basso Livello

Classe Stream	Scopo
`FileInputStream`	Legge byte da file
`FileOutputStream`	Scrive byte su file
`FileReader`	Legge caratteri da file
`FileWriter`	Scrive caratteri su file

- Esempio: stream di byte a basso livello

```
try (InputStream in = new FileInputStream("data.bin")) {
    int b;
    while ((b = in.read()) != -1) {
        System.out.println(b);
    }
}
```

> **Note**
>
> Gli stream di basso livello sono raramente usati da soli nelle applicazioni reali a causa di prestazioni scarse e funzionalità limitate.

34.3.3 Stream di Alto Livello (Filter / Processing Streams)

Gli stream di alto livello avvolgono altri stream per aggiungere funzionalità.

- Buffering
- Conversione del tipo di dato
- Serializzazione di oggetti
- Lettura/scrittura di primitivi

34.3.4 Stream comuni di Alto Livello

Classe Stream	Aggiunge funzionalità
`BufferedInputStream`	Buffering
`BufferedReader`	Lettura basata su linee
`DataInputStream`	Tipi primitivi
`ObjectInputStream`	Serializzazione oggetti
`PrintWriter`	Output testo formattato

- Esempio: chaining degli stream

```
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 Regole di chaining degli stream e errori comuni

L'esempio precedente illustra lo stream chaining, un concetto centrale in `java.io` basato sul pattern decorator.

Ogni stream avvolge un altro stream, aggiungendo funzionalità preservando una gerarchia di tipi rigorosa.

34.3.5.1 Regola fondamentale di chaining

Uno stream può avvolgere solo un altro stream di un livello di astrazione compatibile.

- Gli stream di byte possono avvolgere solo stream di byte
- Gli stream di caratteri possono avvolgere solo stream di caratteri
- Gli stream di alto livello richiedono uno stream di basso livello sottostante

> **Note**
>
> Non puoi mescolare arbitrariamente `InputStream` con `Reader` o `OutputStream` con `Writer`.

34.3.5.2 Incompatibilità tra stream di byte e stream di caratteri

Un errore molto comune è tentare di avvolgere uno stream di byte direttamente con una classe basata su caratteri (o viceversa).

34.3.5.3 Chaining non valido (errore di compilazione)

```
BufferedReader reader =
    new BufferedReader(new FileInputStream("text.txt"));
```

> **Note**
>
> Questo fallisce perché `BufferedReader` si aspetta un `Reader`, non un `InputStream`.

34.3.5.4 Bridging da stream di byte a stream di caratteri

Per convertire tra stream basati su byte e stream basati su caratteri, Java fornisce classi ponte che eseguono decodifica/codifica esplicita del charset.

- `InputStreamReader` converte byte → caratteri
- `OutputStreamWriter` converte caratteri → byte

34.3.5.5 Pattern corretto di conversione

```
BufferedReader reader =
    new BufferedReader(
        new InputStreamReader(new FileInputStream("text.txt")));
```

> **Note**
>
> Il ponte gestisce la decodifica dei caratteri usando un charset (predefinito o esplicito).

34.3.5.6 Regole di ordinamento nelle catene di stream

L'ordine di wrapping non è arbitrario.

- Lo stream di basso livello deve essere il più interno
- I bridge (se necessari) vengono dopo
- Gli stream bufferizzati o di elaborazione vengono per ultimi

34.3.5.7 Ordine logico corretto

```
FileInputStream → InputStreamReader → BufferedReader
```

34.3.5.8 Regola di gestione delle risorse

Chiudere lo stream più esterno chiude automaticamente tutti gli stream avvolti.

> **Note**
>
> Per questo try-with-resources dovrebbe riferirsi solo allo stream di livello più alto.

34.3.5.9 Trappole comuni

- Tentare di bufferizzare uno stream del tipo sbagliato
- Dimenticare il bridge tra stream di byte e stream di char
- Assumere che `Reader` funzioni con dati binari
- Usare il charset predefinito involontariamente
- Chiudere manualmente gli stream interni (rischiando double-close): `close()` sul wrapper esterno è sufficiente ed è raccomandato

---

34.4 Classi base principali di `java.io` e metodi chiave

Il package `java.io` è costruito attorno a un piccolo insieme di **classi base astratte**. Comprendere queste classi e i loro contratti è essenziale, perché tutte le classi I/O concrete si basano su di esse.

34.4.1 InputStream

Classe base astratta per input orientato ai byte. Tutti gli input stream leggono byte grezzi (valori a 8 bit) da una sorgente come un file, un socket di rete o un buffer di memoria.

34.4.1.1 Metodi chiave

Metodo	Descrizione
int `read()`	Legge un byte (0–255); ritorna -1 a fine stream
int `read(byte[])`	Legge byte in un buffer; ritorna numero di byte letti o -1
int `read(byte[], int, int)`	Legge fino a length byte in una slice del buffer
int `available()`	Byte leggibili senza bloccare (hint, non garanzia)
void `close()`	Rilascia la risorsa sottostante

> **Note**
>
> I metodi `read()` sono bloccanti per default.
>
> Sospendono il thread chiamante finché i dati non sono disponibili, finché non si raggiunge end-of-stream, o finché non si verifica un errore I/O.

Il metodo `read()` a singolo byte è principalmente un primitivo di basso livello.

In pratica, leggere un byte alla volta è inefficiente e dovrebbe quasi sempre essere evitato a favore di letture bufferizzate.

34.4.1.2 Esempio tipico di utilizzo

```
try (InputStream in = new FileInputStream("data.bin")) {
    byte[] buffer = new byte[1024];
    int count;
    while ((count = in.read(buffer)) != -1) {
        // process buffer[0..count-1]
    }
}
```

34.4.2 OutputStream

Classe base astratta per output orientato ai byte.

Rappresenta una destinazione dove possono essere scritti byte grezzi.

34.4.2.1 Metodi chiave

Metodo	Descrizione
void `write(int b)`	Scrive gli 8 bit meno significativi dell'intero
void `write(byte[])`	Scrive un intero array di byte
void `write(byte[], int, int)`	Scrive una slice di un array di byte
void `flush()`	Forza la scrittura dei dati bufferizzati
void `close()`	Esegue flush e rilascia la risorsa

> **Note**
>
> Chiamare `close()` richiama implicitamente `flush()`.
>
> Non eseguire flush o close su un OutputStream può causare perdita di dati.

34.4.2.2 Esempio tipico di utilizzo

```
try (OutputStream out = new FileOutputStream("out.bin")) {
    out.write(new byte[] {1, 2, 3, 4});
    out.flush();
}
```

34.4.3 Reader e Writer

`Reader` e `Writer` sono le controparti `orientate ai caratteri` di InputStream e OutputStream.

Operano su caratteri Unicode a 16 bit invece di byte grezzi.

Classe	Direzione	Basata su caratteri	Consapevole dell'encoding
`Reader`	Input	Sì	Sì
`Writer`	Output	Sì	Sì

Reader e Writer implicano sempre un `charset`, esplicitamente o implicitamente.

Questo li rende l'astrazione corretta per l'elaborazione di testo.

34.4.3.1 Gestione del charset

```
Reader reader = new InputStreamReader(
    new FileInputStream("file.txt"),
    StandardCharsets.UTF_8
);
```

> **Note**
>
> `InputStreamReader` e `OutputStreamWriter` sono classi ponte.
>
> Convertono tra `stream di byte` e `stream di caratteri` usando un `charset`.

34.5 Stream bufferizzati e prestazioni

Gli `stream bufferizzati` avvolgono un altro stream e aggiungono un buffer in memoria.

Invece di interagire con il sistema operativo a ogni read o write, i dati vengono accumulati in memoria e trasferiti in blocchi più grandi.

- `BufferedInputStream` / `BufferedOutputStream` per stream di byte
- `BufferedReader` / `BufferedWriter` per stream di caratteri

> **Note**
>
> Gli `stream bufferizzati` sono `decorator`: non sostituiscono lo stream sottostante, lo migliorano aggiungendo comportamento di buffering.

34.5.1 Perché il buffering conta

Aspetto	Non bufferizzato	Bufferizzato
`System calls`	Frequenti	Ridotte
`Prestazioni`	Scarse	Alte
`Uso memoria`	Minimo	Leggermente più alto

Le system call sono operazioni costose.

Il buffering le minimizza raggruppando più letture o scritture logiche in meno operazioni I/O fisiche.

34.5.2 Come funziona la lettura non bufferizzata

In uno stream non bufferizzato, ogni chiamata a read() può risultare in una system call nativa.

Questo è particolarmente inefficiente quando si leggono grandi quantità di dati.

```
try (InputStream in = new FileInputStream("data.bin")) {
    int b;
    while ((b = in.read()) != -1) {
        // ogni read() può innescare una system call
    }
}
```

> **Note**
>
> Leggere byte-per-byte senza buffering è quasi sempre un anti-pattern di prestazioni.

34.5.3 Come funziona BufferedInputStream

`BufferedInputStream` internamente legge un grande blocco di byte in un buffer.

Le successive chiamate `read()` sono servite direttamente dalla memoria finché il buffer non è vuoto.

```
try (InputStream in =
    new BufferedInputStream(new FileInputStream("data.bin"))) {
        int b;
        while ((b = in.read()) != -1) {
            // la maggior parte delle letture è servita dalla memoria, non dall'OS
        }
}
```

> **Note**
>
> Il programma chiama ancora `read()` ripetutamente, ma il sistema operativo viene invocato solo quando il buffer interno deve essere riempito di nuovo.

34.5.4 Esempio di output bufferizzato

L'output bufferizzato accumula dati in memoria e li scrive in blocchi più grandi.

L'operazione `flush()` forza la scrittura immediata del buffer.

```
try (OutputStream out =
    new BufferedOutputStream(new FileOutputStream("out.bin"))) {
        for (int i = 0; i < 1_000; i++) {
            out.write(i);
        }
        out.flush(); // forza i dati bufferizzati su disco
}
```

> **Note**
>
> `close()` chiama automaticamente flush().
>
> Chiamare `flush()` esplicitamente è utile quando i dati devono essere visibili immediatamente.

34.5.5 BufferedReader vs Reader

`BufferedReader` aggiunge una `**lettura basata su linee**` efficiente sopra un Reader.

Senza buffering, ogni carattere letto può coinvolgere una system call.

```
try (BufferedReader reader =
    new BufferedReader(new FileReader("file.txt"))) {

        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
}
```

> **Note**
>
> Il metodo `readLine()` è disponibile solo su `BufferedReader` (non su `Reader`), perché si basa sul buffering per rilevare efficientemente i confini di riga.

34.5.6 Esempio di BufferedWriter

```
try (BufferedWriter writer =
    new BufferedWriter(new FileWriter("file.txt"))) {

        writer.write("Hello");
        writer.newLine();
        writer.write("World");
}
```

`BufferedWriter` minimizza l'accesso al disco e fornisce metodi di convenienza come `newLine()`.

> **Note**
>
> Avvolgi sempre gli stream di file con buffering a meno che non ci sia una forte ragione per non farlo
>
> Preferisci BufferedReader / BufferedWriter per testo
>
> Preferisci BufferedInputStream / BufferedOutputStream per dati binari

---

34.6 java.io vs java.nio (e java.nio.file)

Le applicazioni Java moderne favoriscono sempre più le API NIO e NIO.2, ma java.io rimane fondamentale e ampiamente usato.

34.6.1 Differenze concettuali

Aspetto	java.io	java.nio / nio.2
`Modello di programmazione`	Basato su stream	Basato su buffer / channel
`I/O bloccante`	Bloccante per default	Capace di non-bloccante
`File API`	File	Path + Files
`Scalabilità`	Limitata	Alta
`Introdotto`	Java 1.0	Java 4 / Java 7

> **Note**
>
> `java.nio` non sostituisce `java.io`.
>
> Molte classi NIO internamente si basano su stream o coesistono con essi.

34.6.2 java.nio (I/O file moderno)

Il package `java.nio.file` (NIO.2) fornisce una file API di alto livello, espressiva e più sicura. È l'approccio preferito per operazioni su file in Java 11+.

Esempio: leggere un file (NIO)

```
Path path = Path.of("file.txt");
List<String> lines = Files.readAllLines(path);
```

Codice java.io equivalente

```
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

34.7 Quando usare quale API

Scenario	API raccomandata
`Lettura/scrittura file semplice`	java.nio.file.Files
`Streaming binario`	InputStream / OutputStream
`Elaborazione testo a caratteri`	Reader / Writer
`Server ad alte prestazioni`	java.nio.channels
`API legacy`	java.io

34.8 Trappole comuni e suggerimenti

- End-of-file è indicato da -1, non da un'eccezione
- Chiudere uno stream wrapper chiude automaticamente lo stream avvolto
- `BufferedReader.readLine()` rimuove i separatori di linea
- `InputStreamReader` coinvolge sempre un charset
- I metodi utility Files lanciano IOException checked
- `available()` non deve essere usato per rilevare EOF

Note

La maggior parte dei bug I/O deriva da assunzioni errate su blocking, buffering o character encoding.

◀ 33. API di Files e Path | ▲ Index | 35. API di I/O Java (Legacy e NIO) ▶

35. API di I/O Java (Legacy e NIO)

Indice

- 35.1 Legacy java.io — Design, comportamento e sottigliezze
  - 35.1.1 L'astrazione di stream
  - 35.1.2 Chaining degli stream e pattern Decorator
  - 35.1.3 I/O bloccante: cosa significa
  - 35.1.4 Gestione risorse: close(), flush() e perché esistono
  - 35.1.5 finalize(): perché esiste e perché fallisce
  - 35.1.6 available(): scopo e abuso
  - 35.1.7 mark() e reset(): backtracking controllato
  - 35.1.8 Reader, Writer e codifica dei caratteri
  - 35.1.9 File vs FileDescriptor
- 35.2 java.nio — Buffer, Channel e I/O non bloccante
  - 35.2.1 Dagli stream ai buffer: un cambio concettuale
  - 35.2.2 Buffer: scopo e struttura
  - 35.2.3 Ciclo di vita del buffer: Write → Flip → Read
  - 35.2.4 clear() vs compact()
  - 35.2.5 Heap buffers vs Direct buffers
  - 35.2.6 Channel: cosa sono
  - 35.2.7 Channel bloccanti vs non bloccanti
  - 35.2.8 Scatter/Gather I/O
  - 35.2.9 Selector: multiplexing dell'I/O non bloccante
  - 35.2.10 Quando usare java.nio
- 35.3 java.nio.file (NIO.2) — Operazioni su file e directory (Legacy vs Modern)
  - 35.3.1 Verifiche di esistenza e accessibilità
  - 35.3.2 Creazione di file e directory
  - 35.3.3 Eliminazione di file e directory
  - 35.3.4 Copia di file e directory
  - 35.3.5 Spostamento e rinomina
  - 35.3.6 Lettura e scrittura di testo e byte (miglioramenti di Files)
  - 35.3.7 newInputStream/newOutputStream e newBufferedReader/newBufferedWriter
  - 35.3.8 Listing directory e attraversamento di alberi
  - 35.3.9 Ricerca e filtro
  - 35.3.10 Attributi: lettura, scrittura e view
  - 35.3.11 Link simbolici e follow dei link
  - 35.3.12 Sintesi: perché Files è un miglioramento
- 35.4 Serializzazione — Object stream, compatibilità e trappole
  - 35.4.1 Cosa fa la serializzazione (e cosa non fa)
  - 35.4.2 Le due principali marker interface
  - 35.4.3 Esempio base: scrivere e leggere un oggetto
  - 35.4.4 Grafi di oggetti, riferimenti e identità
  - 35.4.5 serialVersionUID: la chiave di versioning
  - 35.4.6 Campi transient e static
  - 35.4.7 Campi non serializzabili e NotSerializableException
  - 35.4.8 Costruttori e serializzazione
  - 35.4.9 Hook di serializzazione custom: writeObject e readObject
  - 35.4.10 Esempio d'uso: ripristinare un campo derivato transient

- 35.4.11 Externalizable: controllo totale (e responsabilità totale)
- 35.4.12 Considerazioni di sicurezza su readObject()
- 35.4.13 Trappole comuni e consigli pratici
- 35.4.14 Quando usare (o evitare) la serializzazione Java

35.1 Legacy java.io — Design, comportamento e sottigliezze

L'API legacy `java.io` è l'astrazione I/O originale introdotta in Java 1.0.

Essa è orientata agli stream, bloccante, e mappata strettamente sui concetti I/O del sistema operativo.

Anche se esistono API più recenti, `java.io` resta fondamentale: molte API di livello superiore ci si appoggiano, ed è ancora molto usata.

35.1.1 L'astrazione di stream

Uno `stream` rappresenta un flusso continuo di dati tra una sorgente e una destinazione.

In `java.io`, gli stream sono **unidirezionali**: sono o di **input** o di **output**.

Stream	Direzione	Unità di dati	Categoria
`InputStream`	Input	Byte (8-bit)	Stream di byte
`OutputStream`	Output	Byte (8-bit)	Stream di byte
`Reader`	Input	Caratteri	Stream di caratteri
`Writer`	Output	Caratteri	Stream di caratteri

Gli `stream` nascondono l'origine concreta dei dati (file, rete, memoria) ed espongono un'interfaccia uniforme di lettura/scrittura.

35.1.2 Chaining degli stream e pattern Decorator

La maggior parte degli stream java.io è progettata per essere combinata.

Ogni wrapper aggiunge comportamento senza cambiare la sorgente dati sottostante.

```
InputStream in =
    new BufferedInputStream(
        new FileInputStream("data.bin"));
```

In questo esempio:

- `FileInputStream` esegue l'accesso reale al file
- `BufferedInputStream` aggiunge un buffer in memoria

> **Note**
>
> Questo design è noto come **Decorator Pattern**.
>
> Permette di stratificare funzionalità in modo dinamico.

35.1.3 I/O bloccante: cosa significa

Tutti gli stream legacy `java.io` sono **bloccanti**.

Ciò significa che un thread che esegue I/O può essere sospeso dal sistema operativo.

Per esempio, quando chiami `read()`:

- se i dati sono disponibili, vengono restituiti subito
- se non ci sono dati, il thread attende
- se si raggiunge la fine dello stream, viene restituito -1

> **Note**
>
> Il comportamento bloccante semplifica la programmazione, ma limita la scalabilità.

35.1.4 Gestione risorse: `close()`, `flush()` e perché esistono

Gli stream spesso incapsulano risorse native del sistema operativo come `file descriptor` o handle di socket.

Queste risorse sono limitate e devono essere rilasciate esplicitamente.

Metodo	Scopo
`flush()`	Scrive i dati bufferizzati verso la destinazione
`close()`	Esegue flush e rilascia la risorsa

```
try (OutputStream out = new FileOutputStream("file.bin")) {
    out.write(42);
} // close() chiamato automaticamente
```

> **Note**
>
> Non chiudere gli stream può causare perdita di dati o esaurimento delle risorse.

35.1.5 `finalize()`: perché esiste e perché fallisce

Le prime versioni di Java tentarono di automatizzare il cleanup delle risorse usando la finalizzazione.

Il metodo `finalize()` veniva chiamato dal garbage collector prima di recuperare la memoria.

Tuttavia, i tempi del GC sono imprevedibili.

Aspetto	finalize()
Tempo di esecuzione	Non specificato
Affidabilità	Bassa
Stato attuale	Deprecato

> **Note**
>
> `finalize()` non va mai usato per pulizia I/O; è deprecato e non sicuro.

35.1.6 `available()`: scopo e abuso

`available()` stima quanti byte possono essere letti senza bloccare.

Non indica la quantità totale di dati rimanenti.

Casi d'uso tipici:

- evitare blocchi in UI o parsing di protocolli
- dimensionare buffer temporanei

```
if (in.available() > 0) {
    in.read(buffer);
}
```

> **Note**
>
> `available()` non deve essere usato per rilevare EOF. Solo `read()`, che ritorna -1, segnala la fine dello stream.

35.1.7 `mark()` e `reset()`: backtracking controllato

Alcuni input stream consentono di marcare una posizione e tornarci in seguito.

```
BufferedInputStream in = new BufferedInputStream(...);
in.mark(1024);
// read ahead
in.reset();
```

Stream	markSupported()
`FileInputStream`	No
`BufferedInputStream`	Sì
`ByteArrayInputStream`	Sì

35.1.8 Reader, Writer e codifica dei caratteri

`Reader` e `Writer` operano su `caratteri`, non su byte.

Questo richiede una `codifica dei caratteri` (charset).

Se non specifichi un charset, viene usato quello di default della piattaforma.

```
new FileReader("file.txt"); // encoding di default della piattaforma
```

> **Note**
>
> - Affidarsi al charset di default porta a bug di non portabilità.
> - Specifica sempre un charset esplicitamente.

35.1.9 File vs FileDescriptor

`File` rappresenta un `percorso` nel filesystem.

Non rappresenta una risorsa aperta.

`FileDescriptor` rappresenta un handle nativo del SO verso un file o stream aperto.

Classe	Rappresenta	Possiede handle OS?
`File`	Percorso filesystem	No
`FileDescriptor`	Handle file nativo OS	Sì

> **Note**
>
> Più stream possono condividere lo stesso FileDescriptor.
>
> Chiudendone uno, si chiude la risorsa sottostante per tutti.

35.2 `java.nio` — Buffer, Channel e I/O non bloccante

L'API `java.nio` (New I/O) è stata introdotta per risolvere i limiti di `java.io`.

Offre un modello I/O di più basso livello e più esplicito, che mappa bene sui sistemi operativi moderni.

Alla base, `java.nio` ruota attorno a tre concetti:

- `Buffer` — contenitori di memoria espliciti
- `Channel` — connessioni dati bidirezionali
- `Selector` — multiplexing dell'I/O non bloccante

35.2.1 Dagli stream ai buffer: un cambio concettuale

Gli stream legacy nascondono la gestione della memoria al programmatore.

Al contrario, `NIO` rende esplicita la memoria tramite i buffer.

Aspetto	java.io	java.nio
Modello dati	Basato su stream (push)	Basato su buffer (pull dai buffer)
Memoria	Nascosta negli stream	Esplicita via buffer
Controllo	Semplice, poco granulare	Più granulare e configurabile

Con NIO, l'applicazione controlla quando i dati vengono letti in memoria e come vengono consumati.

35.2.2 Buffer: scopo e struttura

Un `buffer` è un contenitore tipizzato a dimensione fissa.

Tutte le operazioni I/O NIO leggono da o scrivono su buffer.

Il buffer più comune è `ByteBuffer`.

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

Proprietà	Significato
`capacity`	Dimensione totale del buffer
`position`	Indice corrente di lettura/scrittura
`limit`	Limite dei dati leggibili o scrivibili

35.2.3 Ciclo di vita del buffer: Write → Flip → Read

I `buffer` hanno un ciclo d'uso rigoroso.

Capirlo male è una fonte comune di bug.

Sequenza tipica:

- scrivi i dati nel buffer
- `flip()` per passare in modalità lettura
- leggi i dati dal buffer
- `clear()` o `compact()` per riutilizzarlo

```
ByteBuffer buffer = ByteBuffer.allocate(16);

buffer.put((byte) 1);
buffer.put((byte) 2);

buffer.flip(); // passa in modalità lettura

while (buffer.hasRemaining()) {
    byte b = buffer.get();
}

buffer.clear(); // pronto a scrivere di nuovo
```

> **Note**
>
> `flip()` non cancella i dati: regola position e limit.

35.2.4 `clear()` vs `compact()`

Dopo la lettura, un buffer può essere riutilizzato in due modi.

Metodo	Comportamento
`clear()`	Scarta i dati non letti
`compact()`	Preserva i dati non letti

`compact()` è utile nei protocolli streaming dove nel buffer possono restare messaggi parziali.

35.2.5 Heap buffers vs Direct buffers

I buffer possono essere allocati in due regioni di memoria diverse.

```
ByteBuffer heap = ByteBuffer.allocate(1024);
ByteBuffer direct = ByteBuffer.allocateDirect(1024);
```

Tipo	Posizione memoria	Caratteristiche
`Heap`	Heap JVM	GC, economico da allocare
`Direct`	Memoria nativa	Miglior throughput I/O, più costoso da allocare

> **Note**
>
> I direct buffer riducono le copie tra JVM e OS, ma vanno usati con attenzione per evitare pressione di memoria.

35.2.6 Channel: cosa sono

Un `channel` rappresenta una connessione verso un'entità I/O come file, socket o device.

A differenza degli stream, **i channel sono bidirezionali**.

Channel	Tipo	Scopo
`FileChannel`	File	I/O su file
`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 bloccanti vs non bloccanti

I channel possono operare in modalità bloccante o non bloccante.

```
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
```

In modalità **non bloccante**:

- `read()` può ritornare subito con 0 byte
- `write()` può scrivere solo una parte dei dati

> **Note**
>
> L'I/O non bloccante sposta complessità dal SO all'applicazione.

35.2.8 Scatter/Gather I/O

NIO supporta lettura/scrittura da/verso più buffer con una singola operazione.

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);

ByteBuffer[] buffers = { header, body };
channel.read(buffers);
```

Utile per protocolli strutturati (header + payload).

35.2.9 Selector: multiplexing dell'I/O non bloccante

I `Selector` permettono a un singolo thread di monitorare più channel.

Sono la base dei server scalabili.

Componente	Ruolo
`Selector`	Monitora più channel
`SelectionKey`	Rappresenta registrazione e stato del channel
`Interest set`	Operazioni osservate (read, write, ecc.)

35.2.10 Quando usare `java.nio`

`NIO` è adatto quando:

- serve alta concorrenza
- ti serve controllo fine sulla memoria
- stai implementando protocolli o server

Per operazioni semplici su file, spesso basta `java.nio.file.Files`.

35.3 `java.nio.file` (NIO.2) — Operazioni su file e directory (Legacy vs Modern)

Questa sezione si concentra sulle operazioni pratiche su file e directory.

Confrontiamo gli approcci legacy (java.io.File + stream java.io) con quelli moderni NIO.2 (Path + Files).

L'obiettivo non è solo conoscere i nomi dei metodi, ma capire:

- cosa fa davvero ogni metodo
- cosa ritorna e come segnala gli errori
- quali trappole esistono (race condition, link, permessi, portabilità)
- quando un metodo di Files è un miglioramento sicuro rispetto al vecchio approccio

35.3.1 Verifiche di esistenza e accessibilità

Un'operazione molto comune è verificare se un file esiste e se è accessibile (lettura, scrittura, esecuzione).

Sia l'API legacy (java.io.File) che NIO.2 (java.nio.file.Files) forniscono metodi per queste verifiche.

È però importante capire che queste verifiche sono volutamente imprecise in entrambe le API.

Sono indizi best-effort, non garanzie affidabili.

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();
```

Questi metodi ritornano boolean e non spiegano perché un'operazione è fallita.

Per esempio, exists() può ritornare false quando:

- il file non esiste davvero
- il file esiste ma l'accesso è negato
- un link simbolico è rotto
- si verifica un errore I/O

L'API non consente di distinguere i casi.

35.3.1.2 API moderna (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);
```

Anche questi metodi ritornano boolean e nascondono la ragione dell'eventuale insuccesso.

NIO.2 aggiunge un metodo esplicito per esprimere incertezza:

```
boolean notExists = Files.notExists(p);
```

> **Note**
>
> `exists()` e `notExists()` possono essere entrambi `false` quando lo stato non è determinabile (per esempio per permessi).

Questo non rende la verifica più accurata: rende solo l'incertezza esplicita.

35.3.1.2.1 Consapevolezza dei link simbolici (miglioramento reale)

Un vero miglioramento di NIO.2 è il controllo su come gestire i link simbolici:

```
Files.exists(p, LinkOption.NOFOLLOW_LINKS);
```

La classe File legacy non distingue in modo affidabile:

- file mancante
- link simbolico rotto
- link verso target inaccessibile

NIO.2 permette check link-aware e ispezione esplicita dei link.

35.3.1.2.2 Pattern d'uso corretto (critico)

Nessuna delle due API dà diagnosi affidabili tramite boolean "di check".

Il codice NIO.2 corretto non "controlla prima".

Invece tenta l'operazione e gestisce l'eccezione:

```
try {
    Files.delete(p);
} catch (NoSuchFileException e) {
    // il file non esiste davvero
} catch (AccessDeniedException e) {
    // problema di permessi
} catch (IOException e) {
    // altro errore I/O
}
```

> **Note**
>
> Il vero vantaggio di NIO.2 è la diagnostica tramite eccezioni durante le azioni, non check di esistenza più "accurati".

35.3.1.2.3 Tabella riassuntiva

Obiettivo	Legacy (File)	Moderno (Files)	Dettaglio chiave
Verificare esistenza	`exists()`	`exists() / notExists()`	notExists() può essere false se lo stato non è determinabile
Verificare read/write	`canRead() / canWrite()`	`isReadable() / isWritable()`	Files può usare LinkOption.NOFOLLOW_LINKS quando supportato
Dettagli errore	Non disponibili	Disponibili via eccezioni sulle azioni	I check boolean non spiegano il motivo del fallimento

35.3.2 Creazione di file e directory

La creazione è una grande debolezza del File legacy.

Nel legacy si usano spesso createNewFile() e mkdir/mkdirs(), che ritornano boolean e danno poche info diagnostiche.

35.3.2.1 API legacy (File)

```
File f = new File("a.txt");
boolean created = f.createNewFile(); // può lanciare IOException

File dir = new File("dir");
boolean ok1 = dir.mkdir();
boolean ok2 = new File("a/b/c").mkdirs();
```

`mkdir()` crea un solo livello; `mkdirs()` crea anche i parent.

Entrambi ritornano false in caso di fallimento ma senza dire il perché.

35.3.2.2 API moderna (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` lancia `FileAlreadyExistsException` se il file esiste.
>
> Spesso è preferibile ai check boolean perché è race-safe.

Obiettivo	Legacy (File)	Moderno (Files)	Dettaglio chiave
Creare file	`createNewFile()`	`createFile()`	NIO lancia FileAlreadyExistsException se esiste
Creare directory	`mkdir()`	`createDirectory()`	NIO lancia eccezioni dettagliate
Creare parent	`mkdirs()`	`createDirectories()`	Atomicità non garantita per directory profonde

35.3.3 Eliminazione di file e directory

La semantica di delete differisce molto tra legacy e NIO.2.

Il legacy `delete()` ritorna boolean; NIO.2 offre metodi che lanciano eccezioni significative.

35.3.3.1 API legacy (File)

```
File f = new File("a.txt");
boolean deleted = f.delete();
```

Se fallisce (permessi, file mancante, directory non vuota), `delete()` spesso ritorna false senza dettagli.

35.3.3.2 API moderna (Files)

```
Files.delete(Path.of("a.txt"));
```

Per “cancella se presente”, usa `deleteIfExists()`.

```
Files.deleteIfExists(Path.of("a.txt"));
```

Obiettivo	Legacy (File)	Moderno (Files)	Dettaglio chiave
Eliminare	`delete()`	`delete()`	`Files.delete()` lancia eccezione con la causa del fallimento
Eliminare se esiste	`exists() + delete()`	`deleteIfExists()`	Evita race TOCTOU (check-then-act)

35.3.4 Copia di file e directory

Nel legacy, copiare richiede tipicamente lettura/scrittura manuale via stream.

NIO.2 fornisce operazioni di copia di alto livello con opzioni.

35.3.4.1 Tecnica legacy (stream manuali)

```
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);
    }
}
```

È verboso ed è facile sbagliare (mancanza di buffering, chiusura, ecc.).

35.3.4.2 API moderna (Files.copy)

```
Files.copy(Path.of("src.bin"), Path.of("dst.bin"));
```

Il comportamento è controllabile con opzioni.

```
Files.copy(
    Path.of("src.bin"),
    Path.of("dst.bin"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES
);
```

> **Note**
>
> `Files.copy` lancia FileAlreadyExistsException per default.
>
> Usa `REPLACE_EXISTING` quando l'overwrite è intenzionale.

Obiettivo	Approccio legacy	Moderno (Files)	Dettaglio chiave
Copiare file	Loop stream manuale	`Files.copy(Path, Path, …)`	Opzioni: `REPLACE_EXISTING`, `COPY_ATTRIBUTES`
Copiare stream	InputStream/OutputStream	`Files.copy(InputStream, Path, …)`	Utile per upload/download e piping
Copiare directory	Ricorsione manuale	`walkFileTree + Files.copy`	Nessun one-liner per copy completa di albero

35.3.5 Spostamento e rinomina

La rinomina legacy usa spesso `File.renameTo()`, notoriamente inaffidabile e dipendente dalla piattaforma.

NIO.2 fornisce `Files.move()` con semantica precisa e opzioni.

35.3.5.1 API legacy

```
boolean ok = new File("a.txt").renameTo(new File("b.txt"));
```

`renameTo()` ritorna false senza spiegare, e può fallire tra filesystem.

35.3.5.2 API moderna

```
Files.move(Path.of("a.txt"), Path.of("b.txt"));
```

Le opzioni rendono il comportamento esplicito.

```
Files.move(
    Path.of("a.txt"),
    Path.of("b.txt"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.ATOMIC_MOVE
);
```

> **Note**
>
> ATOMIC_MOVE è garantito solo se lo spostamento avviene nello stesso filesystem. Altrimenti viene lanciata un'eccezione.

Obiettivo	Legacy (File)	Moderno (Files)	Dettaglio chiave
Rinomina / move	`renameTo()`	`move()`	Exceptions + opzioni esplicite
Move atomico	Non supportato	`move(…, ATOMIC_MOVE)`	Garantito solo stesso filesystem
Replace existing	Non esplicito	`REPLACE_EXISTING`	Intenzione di overwrite esplicita

35.3.6 Lettura e scrittura di testo e byte (miglioramenti di Files)

Un grande miglioramento di NIO.2 è la classe utility `Files`, con metodi di alto livello per lettura/scrittura comuni.

Riduce boilerplate e migliora la correttezza.

35.3.6.1 Lettura/scrittura testo 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");
}
```

Queste classi legacy usano spesso il charset di default se non si utilizza un bridge esplicito.

35.3.6.2 Lettura/scrittura testo moderna

```
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 Lettura/scrittura binaria moderna

```
byte[] data = Files.readAllBytes(Path.of("data.bin"));
Files.write(Path.of("out.bin"), data);
```

> **Important**
>
> `readAllBytes` e `readAllLines` caricano tutto in memoria.
>
> Usa `Files.lines()` (lazy) o, per file grandi, preferisci API streaming come newBufferedReader/newInputStream.

Task	Metodo legacy	Metodo NIO.2 Files	Dettaglio chiave
Leggere tutti i byte	Loop InputStream manuale	`readAllBytes()`	Carica tutto in memoria
Leggere tutte le righe	Loop BufferedReader	`readAllLines()`	Carica tutto in memoria
Leggere righe lazy	Loop BufferedReader	`lines()`	Lazy, stream da chiudere
Scrivere byte	OutputStream	`write(Path, byte[])`	Conciso
Scrivere righe	Loop BufferedWriter	`write(Path, Iterable, …)`	Charset specificabile
Append testo	FileWriter(true)	`write(…, APPEND)`	Opzioni esplicite

35.3.7 newInputStream/newOutputStream e newBufferedReader/newBufferedWriter

Queste `factory method` creano stream/reader a partire da un Path.

Sono il bridge consigliato tra streaming classico e gestione 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 directory e attraversamento di alberi

Nel legacy, il listing directory si basa su `File.list()` e `File.listFiles()`.

Questi metodi ritornano array e offrono poca diagnostica.

35.3.8.1 Listing legacy

```
File dir = new File(".");
File[] children = dir.listFiles();
```

NIO.2 offre più approcci a seconda del bisogno.

35.3.8.2 Listing moderno (DirectoryStream)

```
try (DirectoryStream<Path> ds = Files.newDirectoryStream(Path.of("."))) {
    for (Path p : ds) {
        System.out.println(p);
    }
}
```

35.3.8.3 Walking moderno (Files.walk)

```
Files.walk(Path.of("."))
    .filter(Files::isRegularFile)
    .forEach(System.out::println);
```

> **Note**
>
> `Files.walk` restituisce uno Stream che va chiuso. Usa `try-with-resources`.

```
try (Stream<Path> s = Files.walk(Path.of("."))) {
    s.forEach(System.out::println);
}
```

35.3.8.4 Traversal con FileVisitor

Per controllo completo (skip subtree, gestione errori, follow link), usa `walkFileTree + FileVisitor`.

```
Files.walkFileTree(Path.of("."), new SimpleFileVisitor<>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }
});
```

Obiettivo	Legacy	Moderno	Dettaglio chiave
Listing dir	`list()` / `listFiles()`	`newDirectoryStream()`	Lazy, va chiuso
Walk tree (semplice)	Ricorsione manuale	`walk()` (Stream)	Stream va chiuso
Walk tree (controllo)	Ricorsione manuale	`walkFileTree()`	Controllo fine e gestione errori

35.3.9 Ricerca e filtro

La ricerca è tipicamente `traversal + filtro`.

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 Attributi: lettura, scrittura e view

Il File legacy espone pochi attributi (size, lastModified).

NIO.2 supporta metadata ricchi tramite attribute view.

35.3.10.1 Attributi legacy

```
long size = new File("a.txt").length();
long lm = new File("a.txt").lastModified();
```

35.3.10.2 Attributi moderni

```
BasicFileAttributes a =
    Files.readAttributes(Path.of("a.txt"), BasicFileAttributes.class);

long size = a.size();
FileTime modified = a.lastModifiedTime();
```

Accesso tramite nomi string-based:

```
Object v = Files.getAttribute(Path.of("a.txt"), "basic:size");
Files.setAttribute(Path.of("a.txt"), "basic:lastModifiedTime", FileTime.fromMillis(0));
```

> **Note**
>
> Le attribute view dipendono dal filesystem.
>
> Attributi non supportati generano eccezioni.

35.3.11 Link simbolici e follow dei link

NIO.2 può rilevare e leggere link simbolici in modo esplicito.

```
Path link = Path.of("mylink");
boolean isLink = Files.isSymbolicLink(link);

if (isLink) {
    Path target = Files.readSymbolicLink(link);
}
```

Molti metodi seguono i link di default.

Per evitarlo, passa `LinkOption.NOFOLLOW_LINKS` quando supportato.

35.3.12 Sintesi: perché Files è un miglioramento

La classe utility `Files` migliora la programmazione filesystem perché:

- riduce boilerplate (copy/move/read/write)
- fornisce opzioni esplicite (overwrite, atomic move, follow links)
- offre metadata più ricchi (attributes/views)
- supporta traversal e ricerca scalabili

Le API legacy restano soprattutto per compatibilità o quando richieste da librerie legacy.

35.4 Serializzazione — Object stream, compatibilità e trappole

La serializzazione è il processo di convertire un grafo di oggetti in uno stream di byte per memorizzarlo o trasmetterlo, e ricostruirlo successivamente.

In Java, la serializzazione classica è implementata da `java.io.ObjectOutputStream` e `java.io.ObjectInputStream`.

Questo argomento è importante perché combina:

- stream I/O e grafi di oggetti
- versioning e backward compatibility
- considerazioni di sicurezza e pattern d'uso sicuri
- regole del linguaggio (`transient`, static, `serialVersionUID`)

35.4.1 Cosa fa la serializzazione (e cosa non fa)

Quando un oggetto è serializzato, Java scrive informazioni sufficienti a ricostruirlo:

- nome della classe
- serialVersionUID
- valori dei campi di istanza serializzabili
- riferimenti tra oggetti (identità)

La serializzazione non include automaticamente:

- campi static (stato di classe)
- campi transient (esclusi esplicitamente)
- oggetti referenziati non serializzabili (a meno di gestione speciale)

35.4.2 Le due principali marker interface

La serializzazione Java è abilitata implementando una di queste interfacce.

Interfaccia	Significato	Livello di controllo
`Serializable`	Marker opt-in, meccanismo di default	Medio (hook possibili)
`Externalizable`	Richiede implementazione manuale read/write	Alto (controllo totale sul formato)

> **Note**
>
> `Serializable` non ha metodi: è una marker interface.
>
> `Externalizable` estende Serializable e aggiunge readExternal/writeExternal.

35.4.3 Esempio base: scrivere e leggere un oggetto

Pattern minimo usato in pratica.

```
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()` ritorna Object: serve cast. `readObject()` può lanciare ClassNotFoundException.

35.4.4 Grafi di oggetti, riferimenti e identità

La serializzazione preserva l'identità degli oggetti all'interno dello stesso stream.

Se lo stesso riferimento compare più volte, Java lo scrive una sola volta e poi scrive back-reference.

```
Person p = new Person("Bob", 40);
Object[] arr = { p, p }; // stesso riferimento due volte

out.writeObject(arr);
Object[] restored = (Object[]) in.readObject();

// restored[0] e restored[1] puntano allo stesso oggetto
```

> **Note**
>
> Questo previene ricorsione infinita in grafi ciclici.

35.4.5 `serialVersionUID`: la chiave di versioning

`serialVersionUID` è un identificatore `long` usato per verificare la compatibilità tra stream serializzato e definizione della classe.

Se l'UID differisce, la deserializzazione tipicamente fallisce con InvalidClassException.

Se non dichiari `serialVersionUID`, la JVM ne calcola uno dalla struttura della classe: piccole modifiche possono comprometterlo.

```
class Person implements Serializable {

    private static final long serialVersionUID = 1L;

    private String name;
    private int age;
}
```

Tipo di modifica	Impatto compatibilità (default)
Aggiungere un campo	Spesso compatibile (campo nuovo con default)
Rimuovere un campo	Spesso compatibile (campo mancante ignorato)
Cambiare tipo campo	Spesso incompatibile
Cambiare nome/pacchetto	Incompatibile
Cambiare serialVersionUID	Incompatibile

> **Note**
>
> Dichiarare un serialVersionUID stabile è il modo standard per controllare la compatibilità.

35.4.6 Campi `transient` e `static`

I campi `transient` sono esclusi dalla serializzazione.

Alla deserializzazione, i campi transient assumono valori di default (0, false, null) salvo ripristino manuale.

I campi `static` appartengono alla classe, non all'istanza, quindi non vengono serializzati.

```
class Session implements Serializable {

    private static final long serialVersionUID = 1L;

    static int counter = 0;       // non serializzato
    transient String token;       // non serializzato
    String user;                  // serializzato
}
```

> **Note**
>
> Se un transient serve dopo la deserializzazione, va ricalcolato o ripristinato manualmente.

35.4.7 Campi non serializzabili e NotSerializableException

Se un oggetto contiene un campo il cui tipo non è serializzabile, la serializzazione fallisce con NotSerializableException.

```
class Holder implements Serializable {

    private static final long serialVersionUID = 1L;

    private Thread t; // Thread non è serializzabile
}
```

Soluzioni tipiche:

- marcare il campo transient
- sostituirlo con una rappresentazione serializzabile
- usare hook di serializzazione custom

35.4.8 Costruttori e serializzazione

Il comportamento dei costruttori in deserializzazione è fonte frequente di confusione.

Java ripristina lo stato principalmente dal byte stream, non eseguendo i costruttori.

35.4.8.1 Regola: i costruttori delle classi Serializable NON vengono chiamati

Durante la deserializzazione di una classe Serializable, i suoi costruttori, o gli eventuali blocchi d'inizializzazione statici o d'istanza, NON vengono eseguiti.

L'istanza viene creata senza chiamare quei costruttori (o i blocchi d'inizializzazione statici o d'istanza) e i campi vengono iniettati dallo stream.

> **Note**
>
> Per questo i costruttori delle classi Serializable non devono contenere logica di inizializzazione essenziale: non verrebbe eseguita in deserializzazione.

35.4.8.2 Regola di ereditarietà: viene chiamata la prima superclass non-Serializable

Se una classe Serializable ha una superclasse non Serializable, la deserializzazione deve inizializzare quella parte.

Quindi Java chiama **il costruttore no-arg della prima superclasse non-Serializable**.

Implicazioni:

- la superclasse non Serializable deve avere un no-arg accessibile
- le sottoclassi Serializable saltano i costruttori, le superclassi non Serializable no

35.4.8.3 Tabella: quali costruttori vengono eseguiti

Tipo di classe	Costruttore chiamato in deserializzazione
Classe Serializable	No
Sottoclasse Serializable	No
Prima superclasse non Serializable	Sì (no-arg)
Classe Externalizable	Sì (richiesto public no-arg)

35.4.8.4 Esempio: quali costruttori vengono chiamati

```
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 atteso e spiegazione
Durante la costruzione normale (new C()):

```
A constructor
B constructor
C constructor
```

Durante la deserializzazione (readObject):

```
A constructor
```

Spiegazione:

- C è Serializable → C() non viene chiamato
- B è Serializable → B() non viene chiamato
- A non è Serializable → A() viene chiamato (no-arg)
- I campi di B e C vengono ripristinati dallo stream

> **Note**
>
> Se la prima superclasse non-Serializable non ha un no-arg accessibile, la deserializzazione fallisce.

35.4.9 Hook di serializzazione custom: `writeObject` e `readObject`

Gli hook custom servono quando la serializzazione di default non basta (stato transient, campi derivati, cifratura, validazione, compatibilità).

Sono avanzati ma importanti per una deserializzazione corretta.

35.4.9.1 Perché esiste la serializzazione custom

Di default, Java serializza automaticamente tutti i campi di istanza non static e non transient.

È comodo, ma non copre esigenze frequenti.

Motivi tipici:

- un campo non va salvato direttamente (dati sensibili)
- un campo è derivato/cache e va ricalcolato
- serve validazione in lettura (rifiutare stato invalido)
- serve logica di backward/forward compatibility
- un oggetto referenziato non è Serializable e va gestito

35.4.9.2 Cosa sono davvero `writeObject` e `readObject`

Per personalizzare serializzazione/deserializzazione, una classe può definire due metodi privati speciali chiamati `writeObject` e `readObject`.

Non sono override di metodi di interfacce o superclassi: non fanno parte del normale flusso del programma.

Non li chiami mai tu.

Il framework di serializzazione (ObjectOutputStream/ObjectInputStream) li individua tramite reflection, **solo** se nome e firma sono esatti, e li invoca automaticamente.

Se non esistono (o la firma è sbagliata), viene usata la serializzazione di default.

> **Note**
>
> Se la firma è errata (visibilità, parametri, return type, eccezioni), il framework non la riconosce e torna silenziosamente al default.

35.4.9.3 Firme richieste (esatte)

```java
private void writeObject(ObjectOutputStream out) throws IOException

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException
```

Vincoli:

- devono essere private
- devono ritornare void
- i tipi dei parametri devono combaciare esattamente
- le eccezioni devono essere compatibili

35.4.9.4 Cosa succede in serializzazione: step-by-step

Quando serializzi:

```java
out.writeObject(obj);
```

Meccanismo:

- verifica Serializable
- cerca un private writeObject(ObjectOutputStream)
- se assente → default serialization
- se presente → viene chiamato il tuo writeObject

Punto chiave: dentro writeObject, Java non scrive automaticamente i campi “normali” se non lo chiedi. Per questo esiste:

```java
out.defaultWriteObject();
```

`defaultWriteObject()` significa: “serializza i campi serializzabili normali col meccanismo standard”.

Poi puoi scrivere dati extra come vuoi.

35.4.9.5 Pattern tipico e regola dell'ordine write/read

Pattern tipico: usare default e poi estendere.

L'ordine di lettura deve coincidere con l'ordine di scrittura.

```
private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject(); // scrive i campi normali
    out.writeInt(42);         // scrive dati extra
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();   // legge i campi normali
    int x = in.readInt();     // legge i dati extra nello stesso ordine
}
```

> **Note**
>
> Se scrivi valori extra (int/string/etc.), devi leggerli nella stessa sequenza, altrimenti la deserializzazione fallisce o corrompe lo stato.

35.4.10 Esempio d'uso: ripristinare un campo derivato transient

Caso tipico: ricalcolare un valore cached transient dopo deserializzazione.

```
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();               // ripristina firstName e lastName
        fullName = firstName + " " + lastName; // ricalcola il transient
    }
}
```

35.4.11 Externalizable: controllo totale (e responsabilità totale)

Externalizable richiede di definire manualmente come scrivere e leggere l'oggetto.

Richiede anche un costruttore pubblico no-arg, perché la deserializzazione istanzia prima l'oggetto.

```java
import java.io.*;

class Point implements Externalizable {
    int x;
    int y;

    public Point() { } // richiesto

    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**
>
> Con Externalizable controlli il formato. Se lo cambi, devi gestire tu la backward compatibility.

35.4.12 Considerazioni di sicurezza su `readObject()`

La deserializzazione di dati non fidati è pericolosa perché può eseguire codice indirettamente tramite:

- hook readObject
- logica di inizializzazione
- gadget chain in librerie

Linee guida:

- non deserializzare mai byte non fidati senza un motivo forte
- preferire formati sicuri (JSON, protobuf) per input esterni
- se obbligato, usare object filter e validazione rigorosa

35.4.13 Trappole comuni e consigli pratici

- Serializable è solo marker: non richiede metodi
- `readObject` ritorna Object e può lanciare ClassNotFoundException
- i campi `static` non vengono mai serializzati
- i campi `transient` tornano a default salvo ripristino
- senza `serialVersionUID` la compatibilità può rompersi “a sorpresa”
- Externalizable richiede public no-arg constructor
- NotSerializableException quando un campo referenziato non è serializzabile

35.4.14 Quando usare (o evitare) la serializzazione Java

Usa la serializzazione classica soprattutto per:

- persistenza locale di breve durata con versioni controllate
- caching in memoria quando entrambe le estremità sono fidate
- sistemi legacy che già la usano

Evitala per:

- protocolli di rete pubblici
- storage a lungo termine con schema evolutivo
- input non fidati

36. Interagire con l'Utente (Stream I/O Standard)

Indice

- 36.1 Gli Stream I/O Standard
- 36.2 PrintStream Cosè e Perché Esiste
    - 36.2.1 Caratteristiche Chiave di PrintStream
    - 36.2.2 Uso Base di PrintStream
    - 36.2.3 Formattare l'Output con PrintStream
- 36.3 Leggere Input come Stream I/O
    - 36.3.1 Lettura a Basso Livello da Systemin
    - 36.3.2 Uso di InputStreamReader e BufferedReader
- 36.4 La Classe Scanner Comoda ma Sottile
    - 36.4.1 Problemi Comuni di Scanner
- 36.5 Chiusura degli Stream di Sistema
- 36.6 Acquisire Input con Console
    - 36.6.1 Leggere Input da Console
    - 36.6.2 Leggere Password in Modo Sicuro
- 36.7 Formattare l'Output della Console
- 36.8 Confronto tra Console Scanner e BufferedReader
- 36.9 Redirezione e Stream Standard
- 36.10 Trappole Comuni e Best Practice
- 36.11 Sintesi Finale

I programmi Java spesso devono interagire con l'utente: stampare informazioni, leggere input e formattare l'output.

Questa interazione è implementata usando gli stream I/O standard, che sono normali stream Java connessi al sistema operativo.

Questo capitolo spiega come Java interagisce con la console e l'input/output standard, partendo dai concetti più basilari e passando alle API di livello più alto.

36.1 Gli Stream I/O Standard

Ogni programma Java inizia con tre stream predefiniti forniti dalla JVM.

Sono connessi all'ambiente del processo (di solito un terminale o una console).

Stream	Campo	Tipo	Scopo
Output standard	`System.out`	PrintStream	Output normale
Errore standard	`System.err`	PrintStream	Output di errore
Input standard	`System.in`	InputStream	Input dell'utente

> **Note**
>
> Questi stream sono creati dalla JVM, non dal programma.
>
> Essi esistono per l'intera durata del processo.

36.2 `PrintStream`: Cos'è e Perché Esiste

`PrintStream` è uno stream di output orientato ai byte progettato per output leggibile dall'utente.

Avvolge un altro OutputStream e aggiunge metodi di stampa convenienti.

`System.out` e `System.err` sono entrambi istanze di `PrintStream`.

36.2.1 Caratteristiche Chiave di PrintStream

- Stream orientato ai byte con helper per la stampa di testo
- Fornisce metodi `print()` e `println()`
- Converte automaticamente i valori in testo
- Non lancia `IOException` su errori di scrittura
- Supporta opzionalmente l'auto-flush su newline / `println()`

> **Note**
>
> A differenza della maggior parte degli stream, PrintStream sopprime le IOExceptions.
>
> Gli errori devono essere verificati usando checkError().

36.2.2 Uso Base di PrintStream

```
System.out.println("Hello");
System.out.print("Value: ");
System.out.println(42);
```

`println()` aggiunge automaticamente il separatore di linea specifico della piattaforma.

36.2.3 Formattare l'Output con PrintStream

PrintStream supporta output formattato usando `printf()` e `format()`, che sono basati sulla stessa sintassi di String.format().

```
System.out.printf("Name: %s, Age: %d%n", "Alice", 30);
```

Specificatore	Significato
`%s`	Stringa
`%d`	Intero
`%f`	Virgola mobile
`%n`	Nuova linea indipendente dalla piattaforma

> **Note**
>
> `printf()` non aggiunge automaticamente una nuova linea a meno che non si specifichi `%n`.

36.3 Leggere Input come Stream I/O

L'input standard (System.in) è un InputStream connesso all'input dell'utente.

Fornisce byte grezzi e deve essere adattato per un uso pratico.

36.3.1 Lettura a Basso Livello da System.in

Al livello più basso, puoi leggere byte grezzi da System.in.

Questo è raramente conveniente per programmi interattivi.

```
int b = System.in.read();
```

> **Note**
>
> `System.in.read()` si blocca finché l'input non è disponibile.

36.3.2 Uso di InputStreamReader e BufferedReader

Per leggere input testuale, `System.in` è tipicamente avvolto in un Reader e bufferizzato.

```
BufferedReader reader =
new BufferedReader(new InputStreamReader(System.in));

String line = reader.readLine();
```

Questo converte `byte → caratteri` e permette input basato su linee.

36.4 La Classe Scanner (Comoda ma Sottile)

`Scanner` è un'utilità di alto livello per il parsing di input testuale.

È spesso usata per l'interazione con la console, specialmente in piccoli programmi.

```
Scanner sc = new Scanner(System.in);
int value = sc.nextInt();
String text = sc.nextLine();
```

> **Note**
>
> `Scanner` esegue tokenizzazione e parsing, non semplice lettura.
>
> Questo la rende comoda ma più lenta e talvolta sorprendente.

36.4.1 Problemi Comuni di Scanner

- Mischiare `nextInt()` (e altri `nextXxx()`) con `nextLine()` può sembrare "saltare" input perché il newline finale del token numerico è ancora nel buffer.
- Gli errori di parsing lanciano InputMismatchException
- Scanner è relativamente lenta per input di grandi dimensioni

---

36.5 Chiusura degli Stream di Sistema

Gli `stream di sistema` sono speciali e devono essere gestiti con attenzione.

Stream	Chiudere esplicitamente?
`System.out`	No
`System.err`	No
`System.in`	Di solito no

Chiudere `System.out` o `System.err` chiude lo stream sottostante del sistema operativo e influisce sull'intera JVM: chiudere questi stream influisce sull'intero processo JVM, non solo sulla classe o metodo corrente.

> **Note**
>
> In quasi tutte le applicazioni, NON dovresti chiudere `System.out` o `System.err`.

36.6 Acquisire Input con `Console`

La classe `Console` fornisce un modo di livello più alto e più sicuro per interagire con l'utente.

È progettata specificamente per programmi di console interattivi.

```
Console console = System.console();
if (console == null) {
    throw new IllegalStateException("No console available");
}
```

> **Note**
>
> `System.console()` può restituire `null` quando nessuna console è disponibile (ad es. IDE, input rediretto).

La presenza di una console dipende dalla piattaforma sottostante e da come viene avviata la JVM.

Se la JVM viene avviata da una riga di comando interattiva e i flussi di input/output standard non sono reindirizzati, una console è tipicamente disponibile.
In questo caso, la console è solitamente collegata alla tastiera e al display da cui è stato lanciato il programma.

Se la JVM viene avviata in un contesto non interattivo — ad esempio da un IDE, un pianificatore di processi in background, un gestore di servizi, o con flussi standard reindirizzati — di solito una console non sarà disponibile.

Quando una console esiste, è rappresentata da un'istanza unica della classe Console, che può essere ottenuta invocando il metodo `System.console()`. Se non è disponibile alcun dispositivo console, questo metodo restituirà `null`.

36.6.1 Leggere Input da Console

```
String name = console.readLine("Name: ");
```

`readLine()` stampa un prompt e legge una linea completa di input.

36.6.2 Leggere Password in Modo Sicuro

Console permette di leggere password senza mostrare i caratteri.

```
char[] password = console.readPassword("Password: ");
```

> **Note**
>
> Le password sono restituite come `char[]` così possono essere cancellate dalla memoria.

36.7 Formattare l'Output della Console

Console supporta anche output formattato, simile a PrintStream.

```
console.printf("Welcome %s%n", name);
```

Questo usa gli stessi specificatori di formato di `printf()`.

36.8 Confronto tra Console, Scanner e BufferedReader

API	Caso d'uso	Punti di forza	Limitazioni
`BufferedReader`	Input testuale semplice	Veloce, prevedibile, charset esplicito	Parsing manuale
`Scanner`	Input basato su token / parsing	Comoda, espressiva	Più lenta, comportamento dei token sottile
`Console`	App console interattive	Password, prompt, I/O formattato	Può non essere disponibile (`null`)

36.9 Redirezione e Stream Standard

Gli stream standard possono essere rediretti dal sistema operativo. Il codice Java non deve cambiare.

```
java App < input.txt > output.txt
```

Dal punto di vista del programma, `System.in` e `System.out` si comportano ancora come normali stream.

> **Note**
>
> La redirezione è gestita dal sistema operativo o dalla shell. Il codice Java non deve cambiare per supportarla.

36.10 Trappole Comuni e Best Practice

- PrintStream sopprime le IOExceptions
- `System.console()` può restituire null
- Non chiudere `System.out` o `System.err`
- Scanner mescola parsing e lettura
- Console è preferibile per le password
- Se usi `Scanner` su `System.in`, non chiudere lo Scanner se altre parti del programma devono ancora leggere da `System.in` (chiudere lo Scanner chiude `System.in`).

36.11 Sintesi Finale

- `System.out` e `System.err` sono PrintStream per l'output
- `System.in` è uno stream di byte che deve essere adattato per il testo
- `BufferedReader` e `Scanner` sono strategie comuni di input
- `Console` fornisce input e output interattivo sicuro
- Gli stream standard si integrano naturalmente con la redirezione del sistema operativo

---

Java Platform Module System (JPMS)

37. Java Platform Module System (JPMS)

Indice

- 37.1 Perché i moduli sono stati introdotti
  - 37.1.1 Problemi con il classpath
  - 37.1.2 Esempio di un problema di classpath
- 37.2 Che cos'è un modulo
  - 37.2.1 Proprietà fondamentali dei moduli
  - 37.2.2 Modulo vs package vs JAR
- 37.3 Il descrittore module-info.java
  - 37.3.1 Descrittore di modulo minimo
- 37.4 Struttura delle directory di un modulo
- 37.5 Un primo programma modulare
  - 37.5.1 Classe principale
  - 37.5.2 Descrittore del modulo
- 37.6 Spiegazione dell'incapsulamento forte
- 37.7 Sintesi delle idee chiave

Il `Java Platform Module System` (**JPMS**) è stato introdotto in Java 9.

È un meccanismo a livello di linguaggio e a livello di runtime per strutturare le applicazioni Java in unità fortemente incapsulate chiamate `moduli`.

JPMS influenza come il codice viene:

- organizzato
- compilato
- collegato
- impacchettato
- caricato a runtime

Comprendere JPMS è essenziale per il Java moderno, specialmente per grandi applicazioni, librerie, immagini di runtime e strumenti di deployment.

37.1 Perché i moduli sono stati introdotti

Prima di Java 9, le applicazioni Java erano costruite usando solo:

- `packages`
- file `JAR`
- il `classpath`

Questo modello aveva limitazioni serie man mano che le applicazioni crescevano.

37.1.1 Problemi con il classpath

Il classpath è una lista piatta di JAR in cui:

- tutte le classi pubbliche sono accessibili a tutti
- non esiste una dichiarazione affidabile delle dipendenze
- le versioni in conflitto sono comuni
- l'incapsulamento è debole o inesistente

- classi duplicate si sovrascrivono silenziosamente in base all'ordine del classpath

Questo ha portato a problemi ben noti come:

- "JAR hell"
- bug di ordinamento del classpath
- uso accidentale di API interne
- errori di runtime che non venivano rilevati in fase di compilazione

37.1.2 Esempio di un problema di classpath

Supponiamo che due librerie dipendano da versioni diverse dello stesso JAR di terze parti.

Solo una versione può essere messa sul classpath.

Quale viene scelta dipende solamente dall'ordine del classpath, non dalla appropriatezza effettiva.

> **Note**
>
> Questo problema non può essere risolto in modo affidabile con il solo strumento del classpath.

37.2 Che cos'è un modulo?

Un `modulo` è un'unità di codice nominata e auto-descrittiva.

In sintesi, un modulo è una collezione di uno o più package correlati, insieme a un file descrittore del modulo che ne definisce esplicitamente le sue dipendenze e le funzionalità che rende disponibili.

Un modulo offre quindi a chi lo utilizza un complesso ben definito e controllato di funzionalità.

Ogni modulo nominato ha un nome unico che lo identifica al compilatore e al sistema dei moduli.

Dichiara esplicitamente:

- da cosa dipende
- cosa espone agli altri moduli
- cosa mantiene nascosto

Un modulo è più forte di un package e più strutturato di un JAR.

37.2.1 Proprietà fondamentali dei moduli

Proprietà	Descrizione
Incapsulamento forte	I package sono nascosti di default
Dipendenze esplicite	Le dipendenze devono essere dichiarate
Configurazione affidabile	Dipendenze mancanti causano errori precoci
Identità nominata	Ogni modulo ha un nome unico

37.2.2 Modulo vs package vs JAR

Concetto	Scopo	Incapsulamento
Package	Raggruppamento di namespace	Debole (public ancora visibile)
JAR	Impacchettamento / deployment	Nessuno (tutte le classi visibili quando sul classpath)
Modulo	Incapsulamento + unità di dipendenza	Forte (package non esportati nascosti)

37.3 Il descrittore `module-info.java`

Ogni `modulo nominato` è definito da un file descrittore del modulo chiamato:

```
module-info.java
```

Questo file descrive il modulo al compilatore e al runtime.

37.3.1 Descrittore di modulo minimo

Un descrittore di modulo minimo dichiara solo il nome del modulo. Il nome del file deve essere esattamente `module-info.java` , e deve trovarsi nella root dell'albero dei sorgenti del modulo.

```
module com.example.hello {
}
```

> **Note**
>
> **Un modulo senza direttive non esporta nulla e non dipende da nulla**.

37.4 Struttura delle directory di un modulo

Un progetto modulare segue un layout standard di directory.

Il descrittore del modulo si trova alla root dell'albero dei sorgenti del modulo.

```
src/
└─ com.example.hello/
    ├─ module-info.java
    └─ com/
        └─ example/
            └─ hello/
                └─ Main.java
```

Punti chiave:

- Il **nome della directory corrisponde al nome del modulo**
- `module-info.java` è in cima alla root dei sorgenti del modulo
- i package seguono le regole standard di naming Java

> **Note**
>
> Nei progetti IDE e build-tool, la struttura dei file può differire (ad es. Maven usa `src/main/java`). Ciò che resta sempre vero: `module-info.java` sta nella root dell'albero dei sorgenti del modulo e i percorsi dei package seguono il naming standard Java.

37.5 Un primo programma modulare

Creiamo un'applicazione modulare minima.

37.5.1 Classe principale

```java
package com.example.hello;

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

37.5.2 Descrittore del modulo

```
module com.example.hello {
    exports com.example.hello;
}
```

La `direttiva exports` rende il `package` accessibile ad altri moduli.

Senza di essa, il package è incapsulato e inaccessibile.

37.6 Spiegazione dell'incapsulamento forte

In `JPMS`, i package NON sono accessibili di default.

Anche le classi public sono nascoste a meno che non siano esportate esplicitamente.

Nei moduli, `public` significa "public verso altri moduli *solo se* il package contenitore è esportato."

Situazione	Accessibile da un altro modulo?
Classe public in package non esportato	No
Classe public in package esportato	Sì
Membro protected in package esportato	Sì, ma solo via ereditarietà (non accesso generale)
Classe/membro package-private (qualsiasi package)	No
Membro private	No

> **Note**
>
> Questa è una differenza fondamentale rispetto al modello basato sul classpath.

37.7 Sintesi delle idee chiave

- `JPMS` introduce i moduli come unità forti di incapsulamento
- Le dipendenze sono esplicite e verificate
- `module-info.java` è il descrittore centrale
- I package sono nascosti a meno che non siano esportati
- La visibilità basata su classpath non si applica più nei moduli
- La visibilità public non è più sufficiente: le export del modulo controllano l'accessibilità

◀ 36. Interagire con l'Utente (Stream I/O Standard) | ▲ Index | 38. Compilare, Impacchettare ed Eseguire Moduli ▶

38. Compilare, Impacchettare ed Eseguire Moduli

Indice

- 38.1 Il Module Path vs il Classpath
- 38.2 Opzioni della Riga di Comando Relative ai Moduli
    - 38.2.1 Opzioni Disponibili sia in java che in javac
    - 38.2.2 Opzioni Applicabili Solo a javac
    - 38.2.3 Opzioni Applicabili Solo a java
    - 38.2.4 Distinzioni Importanti
- 38.3 Compilare un Singolo Modulo
- 38.4 Compilare Moduli Multipli Interdipendenti
- 38.5 Impacchettare un Modulo in un JAR Modulare
- 38.6 Eseguire un'Applicazione Modulare
- 38.7 Spiegazione delle Direttive del Modulo
    - 38.7.1 requires
    - 38.7.2 requires transitive
    - 38.7.3 exports
    - 38.7.4 exports-to-qualified-exports
    - 38.7.5 opens
    - 38.7.6 opens-to-qualified-opens
    - 38.7.7 Tabella delle Direttive Principali
    - 38.7.8 Exports vs Opens — Accesso a Compile-Time vs Runtime
- 38.8 Moduli Named, Automatici e Unnamed
    - 38.8.1 Moduli Named
    - 38.8.2 Moduli Automatici
    - 38.8.3 Modulo Unnamed
    - 38.8.4 Riepilogo Comparativo
- 38.9 Approccio Top-Down e Bottom-Up per modularizzare un'applicazione
    - 38.9.1 Approccio Top-Down
        - 38.9.1.1 Regole fondamentali
        - 38.9.1.2 Implicazioni pratiche
        - 38.9.1.3 Riepilogo delle regole di accesso
    - 38.9.2 Approccio Bottom-Up
        - 38.9.2.1 Strategia principale
        - 38.9.2.2 Vantaggi architetturali
    - 38.9.3 Confronto concettuale
    - 38.9.4 Considerazioni sulla migrazione
- 38.10 Ispezionare Moduli e Dipendenze
    - 38.10.1 Descrivere Moduli con java
    - 38.10.2 Descrivere JAR Modulari
    - 38.10.3 Analizzare le Dipendenze con jdeps
- 38.11 Creare Immagini Runtime Personalizzate con jlink
- 38.12 Creare Applicazioni Self-Contained con jpackage
- 38.13 Riepilogo Finale JPMS in Pratica

Una volta che un `modulo` è definito con un file `module-info.java`, deve essere compilato, impacchettato ed eseguito utilizzando strumenti consapevoli dei moduli.

Questa sezione spiega come cambia la `toolchain Java` quando sono coinvolti i moduli.

38.1 Il Module Path vs il Classpath

`JPMS` introduce un nuovo concetto: il **module path**.

Esiste accanto al tradizionale **classpath**, ma i due si comportano in modo molto diverso.

Aspetto	Classpath	Module path
Struttura	Lista piatta di JAR	Moduli con identità
Incapsulamento	Nessuno	Forte
Verifica delle dipendenze	Nessuna	Rigorosa
Split packages	Consentiti	Vietati (moduli nominati)
Ordine di risoluzione	Dipendente dall'ordine	Deterministico

> **Note**
>
> - Un JAR posizionato sul `module path` è trattato come un `module`:
>   - Se contiene un `module-info.class`, diventa un `named module`.
>   - Se non contiene un descrittore di modulo, diventa un `automatic module`.
> - Un JAR posizionato sul classpath non è trattato come un modulo.
>   - Invece, diventa parte dell unnamed module, insieme a tutte le altre entry del classpath.
> - Un JAR modulare (cioè un JAR che contiene module-info.class) può comunque essere utilizzato come un JAR regolare.
>   - Se è posizionato sul classpath invece che sul module path, è trattato come parte dell unnamed module, permettendo alle applicazioni non modulari di utilizzarlo senza adottare il module system.
> - Split packages:
>   - Sono consentiti sul classpath (più JAR possono contenere classi nello stesso package).
>   - Sono vietati per i named modules o automatic modules sul module path.

38.2 Opzioni della Riga di Comando Relative ai Moduli

Quando si lavora con il Java Module System, sia `java` che `javac` forniscono opzioni specifiche per compilare ed eseguire applicazioni modulari.

Alcune opzioni sono condivise, mentre altre sono specifiche di uno strumento.

38.2.1 Opzioni Disponibili sia in `java` che in `javac`

Queste opzioni possono essere utilizzate sia durante la compilazione sia durante l'esecuzione:

- **`--module`** o **`-m`**
  Utilizzata per compilare o eseguire solo il modulo specificato.
- **`--module-path`** o **`-p`**
  Specifica i percorsi nei quali `java` o `javac` cercheranno le definizioni dei moduli.

Il sistema dei moduli Java mette a disposizione tre opzioni speciali da linea di comando, utilizzabili sia con `javac` sia con `java`, che consentono di modificare temporaneamente le regole di accesso tra moduli senza alterare i file `module-info.java`. Queste opzioni hanno effetto solo per quella specifica esecuzione del comando e non modificano in modo permanente i descrittori dei moduli.

Le tre opzioni sono:

- `--add-reads`
- `--add-exports`
- `--add-opens`

Sono generalmente utilizzate per test, retrocompatibilità, migrazione di applicazioni esistenti oppure quando si lavora con moduli di terze parti che non possono essere modificati.

Supponiamo, ad esempio, che `moduleA` debba accedere ai tipi pubblici di `moduleB`, ma che:

- `moduleA` non dichiari `requires moduleB;`
- `moduleB` non esporti il package richiesto verso `moduleA`

Invece di modificare i file `module-info.java`, è possibile concedere temporaneamente l'accesso necessario con:

```
javac --add-reads moduleA=moduleB \
      --add-exports moduleB/com.modB.package1=moduleA \
      ...

java  --add-reads moduleA=moduleB \
      --add-exports moduleB/com.modB.package1=moduleA \
      ...
```

Significato delle opzioni:

- `--add-reads moduleA=moduleB`
  Dichiara temporaneamente che `moduleA` legge `moduleB`.
  È equivalente ad aggiungere `requires moduleB;` nel descrittore di `moduleA`.
  In questo modo, `moduleA` può accedere ai package esportati di `moduleB`.
- `--add-exports moduleB/com.modB.package1=moduleA`
  Esporta temporaneamente il package `com.modB.package1` dal modulo `moduleB` verso `moduleA`.
  È equivalente ad aggiungere:
  `exports com.modB.package1 to moduleA;`
  nel descrittore di `moduleB`.

Distinzione importante:

- `--add-reads` stabilisce la leggibilità a livello di modulo.
- `--add-exports` concede l'accesso a specifici package.
- `--add-opens` (non mostrato sopra) è simile a `--add-exports`, ma consente anche l'accesso tramite reflection profonda (deep reflection), spesso necessario per alcuni framework.

Queste opzioni non modificano i metadati compilati del modulo; si limitano ad adattare il grafo dei moduli per quella specifica esecuzione di `javac` o `java`.

38.2.2 Opzioni Applicabili Solo a `javac`

Queste opzioni si applicano solo in fase di compilazione:

- **`--module-source-path`**
  (nessuna forma abbreviata)
  Utilizzata da `javac` per individuare le definizioni dei moduli sorgente.
- **`-d`**
  Specifica la directory di destinazione nella quale verranno generati i file `.class` dopo la compilazione.

38.2.3 Opzioni Applicabili Solo a `java`

Queste opzioni si applicano solo in fase di esecuzione:

- **`--list-modules`**
  (nessuna forma abbreviata)
  Elenca tutti i moduli osservabili e quindi termina.
- **`--show-module-resolution`**
  (nessuna forma abbreviata)
  Mostra i dettagli della risoluzione dei moduli durante l'avvio dell'applicazione.
- **`--describe-module`** o **`-d`**
  Descrive un modulo specificato e quindi termina.

38.2.4 Distinzioni Importanti

L'opzione `-d` ha significati diversi a seconda dello strumento:

- In **`javac`**, `-d` definisce la directory di destinazione per i file di classe compilati.
- In **`java`**, `-d` è una forma abbreviata di `--describe-module`.

Inoltre, `-d` non deve essere confusa con **`-D`** (D maiuscola).

- **`-D`** viene utilizzata durante l'esecuzione di un programma Java per definire proprietà di sistema come coppie nome-valore nella riga di comando.

```
java -Dconfig.file=app.properties com.example.Main
```

In questo esempio, `-Dconfig.file=app.properties` imposta una proprietà di sistema che può essere letta a runtime tramite `System.getProperty("config.file")`.

38.3 Compilare un Singolo Modulo

Per compilare un modulo, devi specificare il percorso dei sorgenti del modulo e la directory di destinazione.

```
javac -d out \
src/com.example.hello/module-info.java \
src/com.example.hello/com/example/hello/Main.java
```

Un approccio più scalabile utilizza `--module-source-path`.

```
javac --module-source-path src \
      -d out \
      $(find src -name "*.java")
```

> **Note**
>
> `--module-source-path` indica a javac dove trovare più moduli contemporaneamente.

38.4 Compilare Moduli Multipli Interdipendenti

Quando i moduli dipendono l'uno dall'altro, le loro dipendenze devono essere risolvibili in fase di compilazione.

`--module-path` **mods** (directory di esempio contenente moduli interdipendenti) dovrebbe contenere JAR modulari già compilati o directory di moduli compilati (ognuna con il proprio module-info.class).

```
javac -d out \
--module-source-path src \
--module-path mods \
$(find src -name "*.java")
```

Qui:

- `--module-source-path` individua gli alberi dei sorgenti dei moduli
- `--module-path` fornisce moduli già compilati

38.5 Impacchettare un Modulo in un JAR Modulare

Dopo la compilazione, i moduli sono tipicamente impacchettati come file JAR.

Un JAR modulare contiene un `module-info.class` alla sua root.

Se `module-info.class` è presente, il JAR diventa automaticamente un `modulo nominato` e il suo `nome` è preso dal descrittore (non dal nome del file).

```
jar --create \
--file mods/com.example.hello.jar \
--main-class com.example.hello.Main \
-C out/com.example.hello .
```

> **Note**
>
> Un JAR con `module-info.class` è un `modulo nominato, non un modulo automatico`. Quando un JAR contiene un `module-info.class`, il suo nome di modulo è preso da quel file e non è dedotto dal nome del file.

38.6 Eseguire un'Applicazione Modulare

Per eseguire un'applicazione modulare, si utilizza il `module path` e si specifica il `nome del modulo`.

```
java --module-path mods \
--module com.example.hello/com.example.hello.Main
```

Puoi abbreviare usando le opzioni `-p` e `-m`.

```
java -p mods -m com.example.hello/com.example.hello.Main
```

> **Note**
>
> Quando si usano moduli nominati, il classpath è ignorato per la risoluzione delle dipendenze tra moduli.

38.7 Spiegazione delle Direttive del Modulo

Il file `module-info.java` contiene direttive che descrivono dipendenze, visibilità e servizi.

Ogni direttiva ha un significato preciso.

38.7.1 `requires`

La direttiva `requires` dichiara una dipendenza da un altro modulo.

Senza di essa, i tipi del modulo dipendente non possono essere utilizzati.

```
module com.example.app {
    requires com.example.lib;
}
```

Effetti di requires:

- La dipendenza deve essere presente a compile-time e a runtime
- I package esportati del modulo richiesto diventano accessibili

38.7.2 `requires transitive`

`requires transitive` espone una dipendenza ai moduli a valle.

Propaga la leggibilità.

```
module com.example.lib {
    requires transitive com.example.util;
    exports com.example.lib.api;
}
```

Significato:

- **Qualsiasi modulo che richiede com.example.lib legge automaticamente com.example.util**
- **I chiamanti non devono dichiarare requires com.example.util esplicitamente**

> **Note**
>
> Questo è simile alle "dipendenze pubbliche" in altri sistemi di moduli.
>
> Leggibile ≠ esportato: un requisito transitivo non esporta automaticamente i tuoi package.

38.7.3 `exports`

`exports` rende un package accessibile ad altri moduli.

Solo i package esportati sono visibili all'esterno del modulo.

```
module com.example.lib {
    exports com.example.lib.api;
}
```

I package non esportati rimangono fortemente incapsulati.

38.7.4 `exports ... to` (Export Qualificati)

Un export qualificato limita l'accesso a moduli specifici.

```
module com.example.lib {
    exports com.example.internal to com.example.friend;
}
```

Solo i moduli elencati possono accedere al package esportato.

38.7.5 `opens`

`opens` consente un accesso riflessivo profondo a un package.

È usato principalmente da framework che utilizzano reflection.

```
module com.example.app {
    opens com.example.app.model;
}
```

> **Note**
>
> opens NON rende un package accessibile a compile-time. Influenza solo la reflection a runtime.

38.7.6 `opens ... to` (Opens Qualificati)

Puoi limitare l'accesso riflessivo a moduli specifici.

```
module com.example.app {
    opens com.example.app.model to com.fasterxml.jackson.databind;
}
```

> **Note**
>
> `opens` influenza la reflection; `exports` influenza la compilazione e la visibilità dei tipi.

38.7.7 Tabella delle Direttive Principali

Direttiva	Scopo
`requires`	Dichiarare una dipendenza
`requires transitive`	Propagare una dipendenza
`exports`	Esporre un package
`exports ... to`	Esporre a moduli specifici
`opens`	Consentire reflection a runtime
`opens ... to`	Limitare l'accesso riflessivo

38.7.8 Exports vs Opens — Accesso a Compile-Time vs Runtime

Visibilità	Compile-time?	Reflection a runtime?
`exports`	Sì	No
`opens`	No	Sì
`exports ... to`	Sì (moduli limitati)	No
`opens ... to`	No	Sì (moduli limitati)

> **Important**
>
> `JPMS` aggiunge un `module path`, ma il `classpath` esiste ancora. Possono coesistere, ma i moduli nominati hanno la precedenza.

38.8 Moduli Named, Automatici e Unnamed

`JPMS` supporta differenti tipi di moduli per permettere una migrazione graduale dal classpath.

JPMS deve interoperare con codice legacy.

Per supportare l'adozione graduale, la JVM riconosce tre differenti categorie di moduli.

38.8.1 Moduli Named

Un `modulo named` possiede un `module-info.class` e una identità stabile.

- Incapsulamento forte
- Dipendenze esplicite

- Supporto completo JPMS

38.8.2 Moduli Automatici

Un JAR senza `module-info` posizionato nel `module path` diventa un `modulo automatico`.

Il suo nome è derivato dal nome del file JAR.

- Legge tutti gli altri moduli
- Esporta tutti i package
- Nessun incapsulamento forte

> **Note**
>
> I moduli automatici esistono per facilitare la migrazione. Non sono adatti come design a lungo termine.

38.8.3 Modulo Unnamed

Il codice nel classpath appartiene al `modulo unnamed`.

- Legge tutti i moduli named
- Tutti i package sono aperti
- Non può essere richiesto da moduli named

> **Note**
>
> Il `modulo unnamed` preserva il comportamento legacy del classpath.

38.8.4 Riepilogo Comparativo

Tipo di modulo	module-info presente?	Incapsulamento	Legge
`Named`	Sì	Forte	Solo dichiarati
`Automatic`	No	Debole	Tutti i moduli
`Unnamed`	No	Nessuno	Tutti i moduli

38.9 Approccio Top-Down e Bottom-Up per modularizzare un'applicazione

Quando si migra un'applicazione esistente (non modulare) verso il Java Platform Module System (JPMS), è possibile adottare due strategie principali: **top-down** e **bottom-up**.
Entrambi gli approcci richiedono una chiara comprensione delle interazioni tra **moduli nominati**, **moduli automatici** e **modulo non nominato**.

38.9.1 Approccio Top-Down

In un `approccio top-down`, si inizia **modularizzando il modulo principale dell'applicazione**, per poi migrare progressivamente le sue dipendenze.

38.9.1.1 Regole fondamentali

1. **Un JAR posizionato sul module path diventa un modulo automatico.**
    - Il suo nome è determinato:
        - Dall'eventuale voce `Automatic-Module-Name` presente nel manifest, oppure
        - Derivato dal nome del file JAR (i trattini vengono sostituiti con punti e la parte relativa alla versione viene ignorata).
          Esempio:
          `mysql-connector-java-8.0.11.jar` → `mysql.connector.java`

- Un `modulo automatico`:
  - Esporta tutti i suoi package.
  - Legge tutti gli altri moduli.

2. **Un JAR posizionato sul classpath appartiene al modulo non nominato.**
   - Il `modulo non nominato`:
     - Esporta tutti i suoi package.
     - Può leggere tutti gli altri moduli.
   - Tuttavia, non avendo un nome, nessun modulo può dichiarare una clausola `requires` nei suoi confronti.
3. **I moduli esplicitamente nominati (con file `module-info.java`)**
   - Possono dichiarare dipendenze tramite:

     ```
     requires some.module;
     ```

   - Possono dipendere da:
     - Altri moduli nominati
     - Moduli automatici
   - Non possono dipendere dal modulo non nominato (poiché privo di nome).

Conseguenza importante:

> Un `modulo nominato` può leggere un `modulo automatico`, ma non può leggere il `modulo non nominato`.

38.9.1.2 Implicazioni pratiche

Supponiamo:

- JAR dell'applicazione = `A`
- `A` dipende direttamente da `B`
- `B` dipende da `C`

Se modularizzi `A` per primo:

- `A` deve dichiarare `requires B;`
- Di conseguenza, `B` deve trovarsi sul module path (come modulo nominato o automatico)
- Se `B` diventa un modulo nominato:
  - Anche `C` dovrà essere spostato sul module path (nominato o automatico)

Quindi, in una migrazione top-down:

- Si parte dal livello dell'applicazione.
- Si modularizzano progressivamente le dipendenze verso l'esterno.
- I moduli automatici sono spesso utilizzati temporaneamente durante la fase di transizione.

38.9.1.3 Riepilogo delle regole di accesso

Tipo di modulo	Esporta	Può leggere
`Modulo nominato`	Solo export dichiarati	Solo moduli richiesti
`Modulo automatico`	Tutti i package	Tutti i moduli
`Modulo non nominato`	Tutti i package	Tutti i moduli

> **Important**
>
> - I `moduli automatici e non nominati` sono **permissivi**.
> - I `moduli nominati` impongono regole esplicite di dipendenza ed export.

38.9.2 Approccio Bottom-Up

In un `approccio bottom-up`, si inizia modularizzando le `librerie di livello più basso`, per poi risalire progressivamente verso i moduli di livello superiore, `fino all’applicazione principale`.

38.9.2.1 Strategia principale

Si convertono inizialmente le librerie fondamentali in moduli nominati correttamente definiti, dotati di un descrittore esplicito `module-info.java`.

Successivamente:

- Si modularizzano i moduli che dipendono da esse.
- Infine, anche l’applicazione principale diventa un modulo nominato.

Questo approccio enfatizza:

- Relazioni `requires` esplicite
- `exports` controllati
- Una forte incapsulazione fin dall’inizio

38.9.2.2 Vantaggi architetturali

Rispetto ai moduli automatici:

- I moduli nominati esportano solo ciò che è dichiarato esplicitamente.
- Non leggono implicitamente tutti gli altri moduli.
- I confini di incapsulamento sono chiaramente definiti.

La modularizzazione bottom-up porta generalmente a:

- Un grafo delle dipendenze più pulito
- Maggiore manutenibilità
- Confini modulari più solidi

38.9.3 Confronto concettuale

Top-Down

- Si parte dall’applicazione principale.
- Le dipendenze vengono modularizzate secondo necessità.
- Si fa spesso affidamento temporaneo sui moduli automatici.
- Migrazione iniziale più rapida.

Bottom-Up

- Si parte dalle librerie di base.
- I descrittori di modulo vengono definiti in modo rigoroso fin dall’inizio.
- La migrazione procede verso l’alto.
- Produce un’architettura modulare più disciplinata e robusta.

38.9.4 Considerazioni sulla migrazione

Nella pratica, molti progetti reali combinano entrambe le strategie:

- Una migrazione top-down consente di attivare rapidamente l’esecuzione modulare.
- Una fase successiva di raffinamento bottom-up sostituisce i moduli automatici con moduli nominati correttamente definiti.

Questo approccio ibrido consente un'adozione incrementale del JPMS, rafforzando progressivamente l'incapsulamento e la chiarezza architetturale.

38.10 Ispezionare Moduli e Dipendenze

38.10.1 Descrivere Moduli con java

```
java --describe-module java.sql
```

Questo mostra `exports`, `requires` e `services` di un modulo.

38.10.2 Descrivere JAR Modulari

```
jar --describe-module --file mylib.jar
```

38.10.3 Analizzare le Dipendenze con `jdeps`

`jdeps` analizza staticamente le dipendenze di classi e moduli.

```
jdeps myapp.jar
```

```
jdeps --module-path mods --check my.module
```

Per rilevare l'uso di API interne del JDK:

```
jdeps --jdk-internals myapp.jar
```

38.10 Creare Immagini Runtime Personalizzate con `jlink`

`jlink` costruisce un runtime Java minimale contenente solo i moduli richiesti da una applicazione.

```
jlink
--module-path $JAVA_HOME/jmods:mods
--add-modules com.example.app
--output runtime-image
```

Benefici:

- runtime più piccolo
- avvio più rapido
- nessun modulo JDK inutilizzato

38.11 Creare Applicazioni Self-Contained con `jpackage`

`jpackage` costruisce installer specifici per piattaforma o immagini applicative.

```
jpackage
--name MyApp
--input mods
--main-module com.example.app/com.example.Main
```

`jpackage` può produrre:

- .exe / .msi (Windows)
- .pkg / .dmg (macOS)
- .deb / .rpm (Linux)

38.12 Riepilogo Finale JPMS in Pratica

- `JPMS` introduce `incapsulamento forte` e dipendenze affidabili
- I `moduli` sostituiscono convenzioni fragili del classpath
- I `servizi` abilitano architetture disaccoppiate
- `Moduli automatici` e `modulo unnamed` supportano la migrazione
- `jlink` e `jpackage` abilitano modelli moderni di deployment

◀ 37. Java Platform Module System (JPMS) | ▲ Index | 39. Servizi in JPMS (Il Modello ServiceLoader) ▶

39. Servizi in JPMS (Il Modello ServiceLoader)

Indice

- 39.1 Il Problema che i Servizi Risolvono
  - 39.1.1 Ruoli nel Modello dei Servizi
  - 39.1.2 Modulo Interfaccia del Servizio
  - 39.1.3 Modulo Provider del Servizio
  - 39.1.4 Modulo Consumer del Servizio
  - 39.1.5 Caricamento dei Servizi a Runtime
  - 39.1.6 Regole di Risoluzione dei Servizi
  - 39.1.7 Livello Service Locator
  - 39.1.8 Schema Sequenziale di Invocazione
  - 39.1.9 Tabella Riassuntiva dei Componenti

`JPMS` include un meccanismo di servizio integrato che permette ai `moduli` di scoprire e utilizzare implementazioni a runtime senza codificare rigidamente dipendenze tra `provider` e `consumer`.

Questo meccanismo è basato sulla `ServiceLoader API` esistente, ma i moduli lo rendono affidabile, esplicito e sicuro.

39.1 Il Problema che i Servizi Risolvono

Talvolta un modulo necessita di utilizzare una capacità, ma non dovrebbe dipendere da una implementazione specifica.

Esempi tipici includono: - framework di logging - driver di database - sistemi plugin - provider di servizi selezionati a runtime

Senza i servizi, il consumer dovrebbe dipendere direttamente da una implementazione concreta.

Questo crea accoppiamento stretto e riduce la flessibilità.

39.1.1 Ruoli nel Modello dei Servizi

Il `modello dei servizi JPMS` coinvolge quattro ruoli distinti.

Ruolo	Descrizione
`Interfaccia del servizio`	Definisce il contratto
`Provider del servizio`	Implementa il servizio
`Consumer del servizio`	Utilizza il servizio
`Service loader`	Scopre le implementazioni a runtime

39.1.2 Modulo Interfaccia del Servizio

L’ `interfaccia del servizio` definisce l’API da cui i `consumer` dipendono.

Deve essere esportata affinché altri moduli possano vederla.

```
package com.example.service;

public interface GreetingService {
    String greet(String name);
}
```

```
module com.example.service {
    exports com.example.service;
}
```

> **Note**
>
> Il modulo dell'interfaccia del servizio non dovrebbe contenere implementazioni.

39.1.3 Modulo Provider del Servizio

Un `modulo provider` implementa l'interfaccia del servizio e dichiara di fornire il servizio.

```
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
}
```

Punti chiave: - Il `provider` dipende dall' `interfaccia del servizio` - La classe di implementazione non necessita di essere esportata - La direttiva `provides with` registra l'implementazione

39.1.4 Modulo Consumer del Servizio

Il `modulo consumer` dichiara di utilizzare un servizio, ma non nomina alcuna implementazione.

```
module com.example.consumer {
    requires com.example.service;
    uses com.example.service.GreetingService;
}
```

> **Note**
>
> `uses` dichiara l'intenzione di scoprire implementazioni a runtime.
>
> Un modulo che dichiara `uses` ma non ha provider corrispondenti nel module path compila normalmente, ma `ServiceLoader` restituisce un risultato vuoto a runtime.

39.1.5 Caricamento dei Servizi a Runtime

La `ServiceLoader API` esegue la scoperta del servizio.

Trova tutti i provider visibili al grafo dei moduli.

```
ServiceLoader<GreetingService> loader =
    ServiceLoader.load(GreetingService.class);

for (GreetingService service : loader) {
    System.out.println(service.greet("World"));
}
```

`JPMS` garantisce che solo i provider dichiarati siano scoperti.

La scoperta "accidentale" basata su classpath è prevenuta.

39.1.6 Regole di Risoluzione dei Servizi

Affinché un servizio sia individuabile da `ServiceLoader`, devono essere soddisfatte diverse condizioni:

Regola	Significato
Il modulo provider deve essere leggibile	Risolto dal grafo `requires`
L'interfaccia del servizio deve essere esportata	I consumer devono vederla
Il consumer (o il Service Locator) deve dichiarare `uses`	Altrimenti ServiceLoader fallisce
Il provider deve dichiarare `provides`	La scoperta implicita è vietata

39.1.7 Livello Service Locator

È possibile introdurre un livello aggiuntivo denominato `Service Locator`.

In questa architettura:

- Il `consumer` non utilizza direttamente `ServiceLoader`
- Il `Service Locator` è l'unico componente che dichiara `uses`
- Il `consumer` dipende dal `Service Locator`

Struttura architetturale:

```
Consumer → Service Locator → ServiceLoader → Provider
```

Modulo del 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();
    }
}
```

Modulo Consumer:

```
module com.example.consumer {
    requires com.example.locator;
}
```

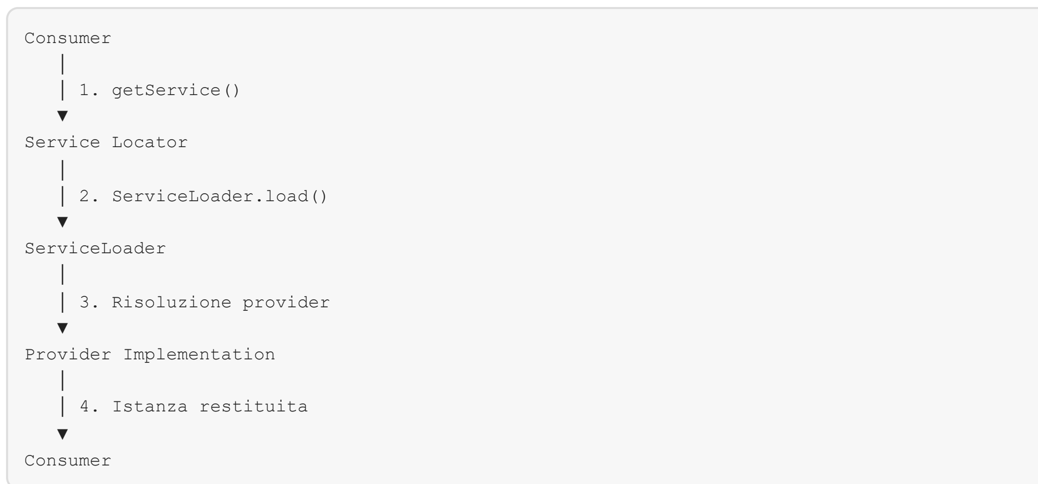
Il consumer non dichiara `uses` perché non invoca direttamente `ServiceLoader`.

39.1.8 Schema Sequenziale di Invocazione

Sequenza di esecuzione:

1. Il `Consumer` invoca `GreetingLocator.getService()`
2. Il `Service Locator` invoca `ServiceLoader.load(...)`
3. Il `ServiceLoader` consulta il grafo dei moduli
4. Il sistema individua i moduli che dichiarano `provides`
5. Viene istanziata l'implementazione del `Provider`
6. L'istanza viene restituita al `Consumer`

Schema sequenziale:



```
Consumer
    |
    | 1. getService()
    ▼
Service Locator
    |
    | 2. ServiceLoader.load()
    ▼
ServiceLoader
    |
    | 3. Risoluzione provider
    ▼
Provider Implementation
    |
    | 4. Istanza restituita
    ▼
Consumer
```

39.1.9 Tabella Riassuntiva dei Componenti

Componente	Ruolo	exports	requires	uses	provides
SPI	Definisce contratto	✅	❌	❌	❌
Provider	Implementa servizio	❌	✅	❌	✅
Service Locator	Esegue discovery	(opzionale)	✅	✅	❌
Consumer	Usa il servizio	❌	✅	❌	❌