Manuale Back-End Java

Realizzato per Colloqui di Lavoro ed il Ripasso

Alessandro Bastianoni

2 dicembre 2025

Indice

Parte I

Fondamenti e Strumenti del Mestiere

Capitolo 1

L'Ecosistema di Sviluppo: Setup Professionale

Prima di scrivere una sola riga di codice, un Back-End Developer deve saper configurare il proprio ambiente. Non si tratta solo di "installare Java", ma di gestire un ecosistema complesso fatto di versioni multiple, licenze diverse e variabili d'ambiente critiche per il funzionamento dei tool di build.

1.1 Il Kit dello Sviluppatore: JDK vs JRE

La prima distinzione fondamentale è tra l'ambiente di esecuzione e quello di sviluppo. Spesso i junior developer installano solo il runtime e si bloccano quando Maven non trova il compilatore.

- **JRE (Java Runtime Environment)**: Contiene la JVM e le librerie standard. Serve solo per *eseguire* programmi Java già compilati (es. file `.jar`).
- **JDK (Java Development Kit)**: È un superset della JRE. Include il compilatore (`javac`), i tool di debug, di monitoraggio (JConsole, JVisualVM) e le librerie per lo sviluppo.

Colloquio: Errore Classico: "javac non riconosciuto"

Domanda: "Ho installato Java, ma quando scrivo `javac -version` nel terminale il comando non viene trovato. Perché?"
Risposta: Probabilmente hai installato solo la JRE. La JRE non contiene `javac`. Devi installare il JDK completo e assicurarti che la cartella `bin` del JDK sia aggiunta alla variabile d'ambiente `PATH` del sistema operativo.

1.2 Quale Java installare? Oracle vs OpenJDK

Fino a qualche anno fa, la scelta era ovvia: Oracle JDK. Oggi, a causa dei cambi di licenza, lo standard industriale si è spostato sulle build gratuite basate su OpenJDK.

- **Oracle JDK**: Richiede una licenza commerciale per uso in produzione (in molti casi).
- **OpenJDK**: È la reference implementation open source.
- **Distribuzioni Production-Ready**: Aziende come Amazon, Microsoft e la community Eclipse forniscono build di OpenJDK stabili, gratuite e con supporto a lungo termine (LTS).
  - **Eclipse Temurin** (ex AdoptOpenJDK): La scelta più neutra e diffusa.
  - **Amazon Corretto**: Ottimizzata per girare su AWS.
  - **Azul Zulu**: Famosa per le performance e il supporto legacy.

1.3 Gestione Versioni Multiple: Mai più installazioni manuali

Un professionista lavora spesso su più progetti contemporaneamente: un monolite legacy in Java 8, un microservizio in Java 17 e un prototipo in Java 21. Disinstallare e reinstallare Java ogni volta è impensabile.

La soluzione standard su Linux/macOS è **SDKMAN!** (Software Development Kit Manager). Su Windows si può usare PowerShell o WSL.

```
# Lista tutte le versioni disponibili
sdk list java

# Installa l'ultima versione LTS di Temurin (Java 17)
sdk install java 17.0.8-tem

# Passa al volo a Java 8 per un progetto legacy
sdk use java 8.0.382-tem

# Controlla la versione attiva
java -version
```

Listing 1.1: Uso di SDKMAN per switchare versione

Questo tool gestisce automaticamente il cambio delle variabili d'ambiente, permettendoti di cambiare contesto in un secondo.

1.4 Le Variabili d'Ambiente: Il motore sotto il cofano

Perché i tool come Maven, Gradle o Tomcat funzionino, il sistema operativo deve sapere *dove* è installato Java.

1.4.1 JAVA_HOME

È la variabile d'ambiente più importante. Deve puntare alla cartella radice dell'installazione del JDK (es. `/usr/lib/jvm/java-17-openjdk`). Molti script di avvio (come quello di Tomcat o di Maven) controllano questa variabile prima di partire. Se non è settata o punta a una JRE, il build fallirà con errori criptici.

1.4.2 PATH

È una lista di directory dove il sistema operativo cerca gli eseguibili. Devi aggiungere `%JAVA_HOME%/bin` (o `$JAVA_HOME/bin`) al PATH per poter lanciare `java` e `javac` da qualsiasi cartella nel terminale.

Deep Dive: Symlinks e Wrapper

Sistemi operativi moderni spesso usano dei "trucchi" (Symlinks o Wrapper) per far funzionare il comando `java` anche senza settare il PATH manualmente. Tuttavia, affidarsi a questi meccanismi è rischioso per lo sviluppo. Configurare esplicitamente `JAVA_HOME` e `PATH` garantisce che l'IDE e i tool di build usino esattamente la versione che ti aspetti, evitando il classico problema "sulla mia macchina compila ma sulla CI fallisce".

Capitolo 2

L'Ambiente di Sviluppo Integrato (IDE)

Java è un linguaggio verboso e fortemente tipizzato. Scriverlo con un semplice editor di testo (come Notepad++ o Sublime) è tecnicamente possibile ma professionalmente suicida. L'IDE (*Integrated Development Environment*) non serve solo a colorare la sintassi: è uno strumento che comprende la struttura logica del tuo progetto, permettendo refactoring complessi e navigazione istantanea tra milioni di righe di codice.

2.1 La "Guerra" degli IDE: Cosa scegliere oggi?

Nel mercato attuale esistono tre contendenti principali. Un Senior Developer deve conoscerli per adattarsi agli standard aziendali.

2.1.1 IntelliJ IDEA (JetBrains)

Oggi è lo standard *de-facto* per lo sviluppo moderno, specialmente con Spring Boot.

- **Pro:** Indicizzazione profonda del codice (suggerimenti "intelligenti"), refactoring imbattibile, supporto nativo a Kotlin e Maven/Gradle.
- **Contro:** Pesante sulla RAM. La versione *Ultimate* (a pagamento) è necessaria per il supporto completo a Spring e Database, anche se la *Community* è sufficiente per iniziare.

2.1.2 Eclipse IDE

Il gigante storico. Ancora molto diffuso nella Pubblica Amministrazione, nelle banche e nei sistemi Legacy.

- **Pro:** Gratuito, open-source, ecosistema di plugin infinito. Utilizza un compilatore incrementale proprietario (ECJ) che permette di eseguire codice anche in presenza di errori in altre classi.
- **Contro:** Interfaccia datata, gestione dei plugin spesso instabile ("DLL hell"), più lento nell'indicizzazione rispetto a IntelliJ.

2.1.3 Visual Studio Code (Microsoft)

L'astro nascente. È un editor leggero che diventa un IDE tramite estensioni (Red Hat Java, Spring Boot Tools).

- **Pro:** Leggerissimo, avvio istantaneo, ottimo per ambienti polyglot (es. Backend Java + Frontend React nello stesso editor).

- **Contro:** Il supporto al refactoring e al debug di applicazioni Java complesse non è ancora al livello di IntelliJ.

Colloquio: Perché usiamo un IDE e non un Text Editor?

Domanda: "In un colloquio di System Design, ti chiedo di rinominare una classe pubblica usata in 50 file diversi. Come procedi?"
Risposta: "Non uso mai il *Find & Replace* testuale, perché è pericoloso (potrebbe rinominare stringhe o commenti che non c'entrano). Uso la funzione di **Refactoring** dell'IDE. L'IDE conosce l'AST (Abstract Syntax Tree) del codice: sa esattamente quali riferimenti puntano a quella classe e li aggiorna in sicurezza, gestendo anche gli import e i nomi dei file."

2.2 Configurazione Critica: Encoding e Line Endings

Prima di scrivere codice in un team, devi configurare l'IDE per evitare il caos nei commit Git.

2.2.1 Text File Encoding: UTF-8 Ovunque

Di default, Windows usa l'encoding `CP1252`, mentre Linux/Mac usano `UTF-8`. Se non forzi l'IDE a usare **UTF-8** per tutto (codice e properties file), i caratteri speciali (accenti, valute) si romperanno passando da un OS all'altro. **In IntelliJ:** `Settings -> Editor -> File Encodings -> Global/Project Encoding: UTF-8`.

2.2.2 Line Separators (LF vs CRLF)

Windows termina le righe con `CRLF` (Carriage Return + Line Feed), Unix/Mac con `LF`. Git può gestirlo automaticamente (`autocrlf`), ma è buona norma configurare l'IDE per usare solo **LF** (Unix style), evitando che i diff di Git segnino ogni riga come modificata solo perché è cambiato il carattere invisibile di fine riga.

2.3 Strumenti di Produttività

2.3.1 Decompiler Integrato

Spesso dovrai capire perché una libreria esterna lancia un errore. Non hai il codice sorgente. In IntelliJ, basta fare `CTRL+Click` su una classe di libreria: l'IDE decompila il `.class` al volo mostrandoti il codice sorgente quasi originale. È fondamentale per il debugging avanzato.

2.3.2 Live Templates

Non scrivere mai `public static void main...` a mano.

- Scrivi `psvm` + Tab → genera il main.
- Scrivi `sout` + Tab → genera `System.out.println()`.
- Scrivi `iter` + Tab → genera un ciclo for-each.

Deep Dive: Il file .gitignore e i file dell'IDE

Ogni IDE crea delle cartelle di configurazione all'interno del progetto (es. `.idea/` per IntelliJ, `.settings/` e `.classpath` per Eclipse, `.vscode/` per VS Code). **Regola d'oro:** Queste cartelle NON devono mai finire nel repository Git. Contengono configurazioni

locali (percorsi del tuo PC) che romperebbero il progetto ai tuoi colleghi. Vanno inserite tassativamente nel file `.gitignore`. La configurazione del progetto deve risiedere solo nel file di build agnostico (`pom.xml` o `build.gradle`).

Capitolo 3

Scrivere Java: Sintassi, Convenzioni e Stile

Un principio fondamentale dell'ingegneria del software è: *"Il codice è scritto per essere letto dagli umani, e solo incidentalmente per essere eseguito dalle macchine."* Java impone una struttura rigida. In questo capitolo analizzeremo l'anatomia di un file sorgente, le regole di nomenclatura (Naming Conventions) e come documentare il codice affinché sia manutenibile nel tempo.

3.1 Anatomia di un file Java

Ogni file sorgente `.java` segue una struttura gerarchica precisa.

```java
// 1. Dichiarazione del Package (Namespace)
package com.azienda.progetto.service;

// 2. Import (Dipendenze)
import java.util.List;
import java.util.ArrayList;

// 3. Definizione della Classe
public class UserService {

    // Campi (Stato)
    private static final int MAX_USERS = 100;
    private String serviceName;

    // Costruttore
    public UserService(String name) {
        this.serviceName = name;
    }

    // Metodi (Comportamento)
    public List<String> findAll() {
        return new ArrayList<>();
    }
}
```

Listing 3.1: Struttura tipica di una classe Java

3.1.1 La Regola del Nome del File

In Java esiste un vincolo fisico: una classe `public` deve essere definita in un file che ha **esattamente lo stesso nome** (case-sensitive) ed estensione `.java`. Se la classe è `public class UserService`, il file **deve** chiamarsi `UserService.java`. Se non lo fai, il compilatore lancerà un errore.

3.2 Naming Conventions: Il Codice Parla

Java ha uno standard di nomenclatura non scritto ma universalmente accettato (definito originariamente da Sun Microsystems e poi da Google/Oracle). Violarlo ti fa apparire immediatamente inesperto agli occhi di un recruiter o di un team leader.

Elemento	Convenzione	Esempio
Classi / Interfacce	**PascalCase**. Inizia con maiuscola, ogni nuova parola è maiuscola. Nomi Sostantivi.	`UserService`
Metodi	**camelCase**. Inizia con minuscola. Verbi o azioni.	`calculateTax()`
Variabili	**camelCase**. Corte ma significative.	`userName`
Costanti	**UPPER_SNAKE_CASE**. Tutte maiuscole con underscore.	`MAX_RETRY`
Package	Tutto **lowercase**. Spesso usa il dominio invertito.	`com.google.common`

Tabella 3.1: Java Naming Conventions

Colloquio: Perché le convenzioni sono importanti?

Domanda: "Il compilatore se ne frega se chiamo una classe `userService` (minuscolo) o una variabile `USER_NAME`. Perché dovrei seguirle?"
Risposta: Per ridurre il **Carico Cognitivo**. Quando un programmatore Java legge `Color.RED`, sa istantaneamente (senza guardare la definizione) che `Color` è una Classe e `RED` è una Costante. Se violi le convenzioni, costringi chi legge a fermarsi e controllare, rallentando l'intero team e aumentando il rischio di bug.

3.3 Package e Directory Structure

I package servono a evitare conflitti di nomi (Namespace) e a organizzare logicamente il codice. Ma in Java, c'è un legame diretto tra nome logico e posizione fisica.

Deep Dive: Il Classpath e le Cartelle

Se dichiari `package com.azienda.app;`, il file **deve** trovarsi fisicamente nel percorso di cartelle: `src/main/java/com/azienda/app/`.

La JVM usa il **Classpath** come punto di partenza e poi cerca le classi scendendo nelle cartelle che corrispondono ai package. Se sposti il file senza cambiare la dichiarazione

`package`, otterrai l'errore runtime: `Could not find or load main class`.

3.4 Javadoc vs Commenti: Documentare l'Intento

Scrivere codice autodocumentante è l'obiettivo, ma a volte serve spiegare il *perché* o il *contratto* di un metodo.

3.4.1 Commenti Implementativi (//)

Si usano con il doppio slash `//`. Servono a spiegare passaggi complessi **dentro** un metodo. *Best Practice:* Non commentare *cosa* fai (il codice lo dice già), commenta *perché* lo fai.

3.4.2 Javadoc (/** ... */)

Si usa per documentare Classi e Metodi pubblici (API). Viene processato da tool esterni per generare documentazione HTML.

```
/**
 * Calcola l'interesse composto.
 *
 * @param amount L'importo iniziale (non può essere negativo)
 * @param rate Il tasso di interesse annuale
 * @return L'importo finale calcolato
 * @throws IllegalArgumentException se amount è negativo
 */
public BigDecimal calculateInterest(BigDecimal amount, double rate) { ... }
```

Notare come il Javadoc definisce il **Contratto**: input, output ed eccezioni. Questo è vitale per chi userà il tuo metodo senza poterne leggere il codice interno.

Capitolo 4

Storia Moderna: Le Versioni LTS e le Feature Chiave

Java non è un linguaggio statico. Dal 2017, il ciclo di rilascio è cambiato radicalmente, passando da anni di attesa a una release ogni 6 mesi. Per un professionista, orientarsi in questa "zuppa di numeri" è fondamentale. Non devi conoscere tutte le versioni, ma devi conoscere le **LTS (Long Term Support)**.

4.1 Il Ciclo di Rilascio: LTS vs Feature Release

Oracle e la community OpenJDK hanno adottato un modello a due velocità:

- **Feature Releases (ogni 6 mesi):** Versioni intermedie (es. 18, 19, 20, 22, 23, 24). Hanno vita breve (supporto di 6 mesi). Servono per testare nuove funzionalità in produzione rapida. Sconsigliate per progetti enterprise a lungo termine.
- **LTS - Long Term Support (ogni 2-3 anni):** Versioni stabili (8, 11, 17, 21, 25). I vendor (Oracle, Amazon, Red Hat) garantiscono patch di sicurezza per anni (anche 8-10 anni). Sono lo standard industriale.

Colloquio: Che versione usare in Produzione?

Domanda: "Se dovessi iniziare un nuovo progetto oggi con Spring Boot 3, quale versione di Java sceglieresti?"
Risposta: "Sceglierei **Java 17** o **Java 21**. Spring Boot 3 richiede come minimo Java 17 (ha droppato il supporto a Java 8 e 11). Java 21 è preferibile se vogliamo sfruttare i Virtual Threads per la scalabilità, altrimenti la 17 è la scelta conservativa e ultra-stabile."

4.2 Java 8: LTS 2014 - Il Grande Spartiacque (Legacy)

Rilasciato nel 2014, è stato il cambiamento più grande nella storia del linguaggio. Ha introdotto il paradigma funzionale.

- **Lambda Expressions:** Funzioni anonime concise (`x -> x * 2`).
- **Stream API:** Elaborazione dati dichiarativa (filter, map, reduce).
- **Optional:** Un contenitore per evitare le `NullPointerException`.
- **Date/Time API:** Il pacchetto `java.time` (immutabile) che sostituisce le vecchie e buggate `Date` e `Calendar`.

Stato attuale: Ancora molto diffuso nei sistemi legacy bancari/assicurativi, ma in fase di dismissione attiva.

4.3 Java 11: LTS 2018 - La Pulizia (LTS)

La prima LTS dopo il cambio di licenza.

- **Local Variable Syntax (var):** Inferenza dei tipi per variabili locali. Rende il codice meno verboso.
- **Rimozione moduli Java EE:** JAXB, CORBA e altri moduli "enterprise" sono stati rimossi dal JDK standard (bisogna aggiungerli come dipendenze Maven).
- **HttpClient nativo:** Finalmente un client HTTP moderno e asincrono integrato nel JDK.

```
// Prima di Java 10/11
HashMap<String, List<String>> map = new HashMap<>();

// Con Java 11
var map = new HashMap<String, List<String>>(); // Il compilatore inferisce il tipo
```

Listing 4.1: Esempio di var in Java 11

4.4 Java 17: LTS 2021 - Lo Standard Moderno (LTS)

Questa è la versione base per tutto l'ecosistema moderno (Spring Boot 3, Quarkus).

- **Records (Java 14/16):** Classi immutabili "data-carrier". Sostituiscono i DTO pieni di boilerplate (getter, setter, equals, hashCode).
- **Text Blocks (Java 15):** Stringhe multilinea reali, utilissime per incollare JSON o SQL nel codice.
- **Switch Expressions:** Switch migliorato che ritorna valori e non richiede il `break`.
- **Pattern Matching per instanceof:** Evita il cast esplicito dopo il controllo del tipo.

```
// Record: Tutto (costruttore, getter, equals) in una riga
public record UserDTO(String name, String email) {}

// Text Block
String json = """
    {
      "name": "Mario",
      "city": "Roma"
    }
    """;
```

Listing 4.2: Record e Text Block in Java 17

4.5 Java 21: LTS 2023 - Il Futuro della Concorrenza

Rilasciato a Settembre 2023, sta cambiando il modo in cui gestiamo i thread.

- **Virtual Threads (Project Loom):** Thread ultra-leggeri gestiti dalla JVM, non dal sistema operativo. Permettono di avere milioni di thread simultanei, rendendo il codice bloccante scalabile quanto quello reattivo.
- **Sequenced Collections:** Nuove interfacce per accedere a primo/ultimo elemento di liste e set in modo unificato (`getFirst()`, `getLast()`).

Deep Dive: Perché la migrazione da 8 a 17 è difficile?

Passare da Java 8 a Java 17+ non è banale. Il motivo principale è il **Module System (Project Jigsaw)** introdotto in Java 9. Questo sistema incapsula le API interne del JDK (`sun.misc.*`). Molte librerie vecchie (es. vecchie versioni di Hibernate o Lombok) usavano queste API interne e si rompono sulle nuove versioni. Migrare richiede di aggiornare tutte le dipendenze del progetto.

4.6 Java 25: LTS 2025 - Stabilità, Velocità e Semplicità

Java 25 è l"ultima versione con supporto a lungo termine. Non introduce una "mega-rivoluzione" come era avvenuto con i Virtual Threads di Java 21, ma porta miglioramenti molto concreti che rendono Java più veloce, più leggero e più semplice da usare.

- **Programmi più veloci da avviare.** La JVM può preparare alcune ottimizzazioni in anticipo, così le applicazioni si "scaldano" più rapidamente. È utile per microservizi o programmi che devono partire spesso.
- **Meno memoria sprecata.** Grazie agli "header compatti", ogni oggetto occupa meno spazio. Questo aumenta l"efficienza quando ci sono tante istanze in memoria (API, batch, processing di dati).
- **Codice più semplice per programmi piccoli.** Java 25 permette di scrivere file sorgente più compatti, senza tutto il boilerplate classico. Perfetto per script, strumenti interni, esempi didattici.
- **Concorrenza più sicura e più pulita.** Con le nuove "Scoped Values" diventa più facile far circolare informazioni tra thread senza usare meccanismi complessi. Questo si integra bene con i Virtual Threads introdotti in Java 21.
- **Maggiore aiuto nel capire cosa succede dentro il programma.** I miglioramenti agli strumenti di diagnostica (come Flight Recorder) rendono più semplice monitorare prestazioni, colli di bottiglia e comportamenti sospetti.
- **Sicurezza moderna.** Arrivano nuove funzioni crittografiche pronte all"uso, utili per autenticazione, cifratura, gestione di chiavi. Non serve più dipendere da librerie esterne.

Parte II

Il cuore di Java

Capitolo 5

Introduzione e Architettura della JVM

Java non è solo un linguaggio, è un intero ecosistema. Per superare un colloquio tecnico, non basta saper scrivere codice: bisogna capire come questo codice viene eseguito, ottimizzato e gestito in memoria.

5.1 Cos'è Java? Caratteristiche Chiave

Java è un linguaggio di programmazione nato con una filosofia precisa: *"Write Once, Run Anywhere"* (Scrivi una volta, esegui ovunque). È definito come un linguaggio di **alto livello**, **orientato agli oggetti (OOP)** e **fortemente tipizzato**.

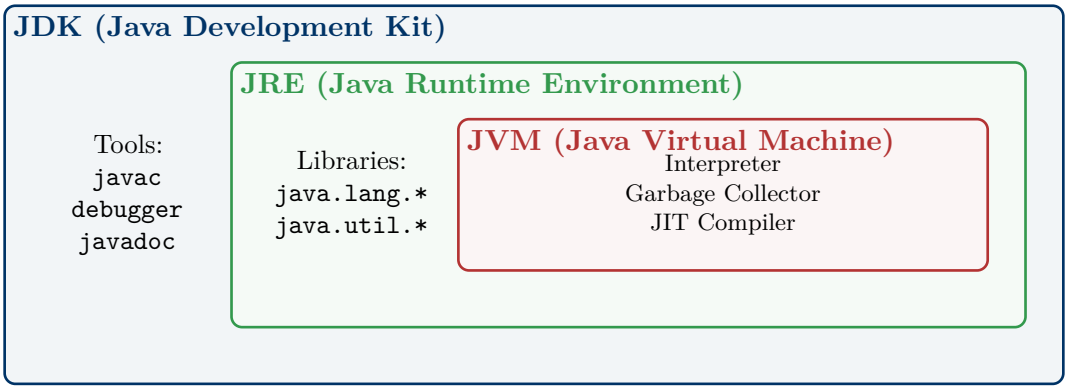
Colloquio: Quali sono le caratteristiche principali di Java?

Questa è una domanda classica per rompere il ghiaccio. Oltre alla lista standard, è importante spiegare il *perché* di queste caratteristiche:

1. **Platform Independent (WORA):** A differenza di C++, Java non compila in codice macchina specifico per la CPU, ma in **Bytecode**. Questo formato intermedio può essere eseguito su qualsiasi OS dotato di una JVM.
2. **Object-Oriented (OOP):** Tutto in Java è un oggetto (tranne i tipi primitivi, anche se esistono i Wrapper). Supporta nativamente Incapsulamento, Ereditarietà, Polimorfismo e Astrazione.
3. **Robusto e Sicuro:**
   - *Memory Management:* Gestisce automaticamente la memoria tramite il Garbage Collector (niente `malloc/free` o puntatori espliciti pericolosi come in C).
   - *Exception Handling:* Possiede un sistema rigoroso di gestione degli errori (Checked vs Unchecked).
4. **Multi-threading:** Permette l'esecuzione simultanea di più parti di codice (Thread) per massimizzare l'uso della CPU.
5. **Rich Standard Library:** Offre API predefinite per quasi ogni necessità (Networking, I/O, Collections, Concurrency, Database).

5.2 Dietro le Quinte: JDK, JRE e JVM

Per capire come Java funzioni "dietro le quinte" e risolvere problemi di configurazione ambientale, dobbiamo distinguere tre acronimi fondamentali che formano una gerarchia inclusiva.



- **JDK (Java Development Kit):** È il kit completo per gli sviluppatori.
  - Contiene il compilatore (`javac`), debugger, javadoc.
  - **Include al suo interno la JRE.**
- **JRE (Java Runtime Environment):** È l'ambiente minimo necessario per *eseguire* (non sviluppare) programmi Java.
  - Contiene le librerie standard di classe Java (es. `java.lang.*`, `java.util.*`).
  - **Include al suo interno la JVM.**
- **JVM (Java Virtual Machine):** È il "motore" che esegue fisicamente il codice.
  - Interpreta il Bytecode e lo traduce in istruzioni macchina.
  - Gestisce la memoria (Heap/Stack) e il Garbage Collection.
  - **Nota:** La JVM è *Platform Dependent* (esiste una JVM specifica per Windows, una per Linux, una per Mac), mentre il Bytecode è *Platform Independent*.

$$JDK \supset JRE \supset JVM$$

5.3 Il ciclo di vita del codice: Compile-time vs Runtime

Colloquio: Cosa succede quando compili ed esegui un programma Java?

Il processo si divide in due fasi distinte e sequenziali. Confondere errori di compilazione con errori di runtime è un segnale di inesperienza.

1. **Compile-time (Compilazione)**
   - Il programmatore scrive il codice sorgente (`.java`).
   - Il compilatore `javac` controlla la sintassi e la tipizzazione statica.
   - Se non ci sono errori, genera il **Bytecode** (file `.class`).
2. **Runtime (Esecuzione)**
   - **Class Loading:** Il *ClassLoader* carica i file `.class` necessari in memoria.
   - **Bytecode Verification:** La JVM controlla che il bytecode sia sicuro e valido.
   - **Execution:** La JVM esegue le istruzioni. Qui entra in gioco il **JIT (Just-In-Time) Compiler**.

Deep Dive: JIT Compiler e Performance

Java è spesso accusato di essere lento rispetto a C++. Tuttavia, il **JIT Compiler** mitiga questo problema. Mentre il codice viene interpretato, il JIT identifica le parti di codice eseguite più frequentemente (*Hotspots*). Compila queste parti direttamente in **codice macchina nativo** (altamente ottimizzato) e lo memorizza nella Code Cache. Risultato: Dopo una fase di "riscaldamento" (warm-up), un'applicazione Java può raggiungere performance vicine a quelle native.

5.3.1 Analisi dell'Entry Point: il metodo main

Tutto inizia da qui. Ogni parola chiave ha un ruolo specifico per la JVM.

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

Listing 5.1: Hello World standard

- **public:** Modificatore di accesso. Deve essere pubblico affinché la JVM (che è esterna al programma) possa invocare questo metodo.
- **static:** Permette alla JVM di invocare il metodo *senza dover prima istanziare un oggetto* della classe Main. Al momento dell'avvio non esistono oggetti, esiste solo la classe.
- **void:** Il main non restituisce alcun valore alla JVM; una volta terminato, il programma finisce (o termina il thread principale).
- **String[] args:** Un array di stringhe che raccoglie eventuali argomenti passati da riga di comando (es. configurazioni di avvio).

5.4 Gestione della Memoria: Primitivi vs Riferimenti

Per comprendere a fondo la differenza tra tipi primitivi e oggetti (reference types), dobbiamo guardare come Java organizza la RAM. La memoria utilizzata dalla Java Virtual Machine (JVM) è divisa principalmente in due aree logiche: lo **Stack** e lo **Heap**.

5.4.1 Lo Stack (La Pila)

Lo Stack è un'area di memoria *veloce*, ordinata secondo il principio LIFO (Last In, First Out).

- Ogni volta che viene invocato un metodo, viene creato un nuovo blocco di memoria chiamato **Stack Frame**.
- Lo Stack contiene i **tipi primitivi** locali e i **riferimenti** agli oggetti.
- Quando il metodo termina, il Frame viene distrutto e la memoria liberata immediatamente.
- È thread-safe: ogni Thread ha il proprio Stack.

5.4.2 Lo Heap (Il Mucchio)

Lo Heap è un'area di memoria molto più grande e disordinata.

- Qui risiedono **tutti gli oggetti** (istanze di classi) e gli array.
- È condiviso tra tutti i Thread.
- La memoria non viene liberata automaticamente alla fine del metodo, ma è gestita dal **Garbage Collector**.

5.4.3 Tipi Primitivi: Allocazione Diretta

Quando dichiari un tipo primitivo (es. `int`, `double`, `boolean`), Java riserva lo spazio esatto in bit direttamente nello Stack Frame corrente. Il nome della variabile è associato direttamente al valore binario.

Esempio: `int x = 42;`

- Nello Stack vengono allocati **4 byte** (32 bit).
- Il valore binario di 42 (`0000...00101010`) è scritto in quei 4 byte.
- Non c'è nessun overhead (peso aggiuntivo).

5.4.4 Oggetti: Il Concetto di Riferimento

Quando dichiari un oggetto, la situazione cambia drasticamente.

```
1 Person p = new Person("Alessandro");
```

Listing 5.2: Dichiarazione di un Oggetto

In questa singola riga avvengono due allocazioni distinte:

1. **Nello Stack (Il Riferimento):** Viene creata la variabile `p`. Questa variabile **non contiene i dati** dell'oggetto. Contiene un indirizzo di memoria (un "puntatore" o "reference") che indica dove trovare l'oggetto nello Heap.
   - Dimensione: Solitamente **4 byte**, se la JVM è 32 bit oppure usa *Compressed Oops* ovvero una tecnica che consente di memorizzare i riferimenti in formato ridotto sfruttando l'allineamento della memoria, oppure **8 byte** su JVM a 64 bit senza compressione, quando sono richiesti puntatori a 64 bit completi.
2. **Nello Heap (L'Oggetto Reale):** Viene allocato spazio per l'istanza di `Person`. Qui risiedono i campi interni (es. la stringa "Alessandro").

L'operatore `=` copia solo il contenuto dello Stack.

```
1 Person a = new Person("Mario");
2 Person b = a;
3 // b ora contiene lo STESSO indirizzo di memoria di a.
4 // Non e' stato creato un nuovo "Mario".
5 // Entrambi puntano allo stesso oggetto nello Heap.
```

Listing 5.3: Copia di riferimenti

Deep Dive: Quanto pesa davvero un Oggetto? (Byte Level)

È un errore comune pensare che un oggetto vuoto (senza campi) occupi 0 byte. In realtà, ogni oggetto allocato nello Heap porta con sé un sovraccarico strutturale (*Overhead*) gestito dalla JVM, noto come **Object Header**.

La struttura in memoria di un oggetto è composta da:

- **Mark Word (4 o 8 byte):** Contiene metadati vitali come l'hashcode, i lock di sincronizzazione e i flag per il Garbage Collector. (4 byte su 32-bit, 8 byte su 64-bit).
- **Class Pointer (4 o 8 byte):** È il puntatore che collega l'istanza alla sua classe nel Metaspace. Su JVM a 64-bit, questo puntatore viene solitamente compresso a 4 byte (*Compressed Oops*) per risparmiare memoria.
- **Instance Data:** I dati effettivi dei campi dell'oggetto (0 byte se l'oggetto è vuoto).
- **Padding:** Spazio vuoto aggiunto per rispettare l'allineamento della memoria. La JVM impone che ogni oggetto occupi un numero di byte multiplo di 8.

Esempi di calcolo per un oggetto vuoto:

1. **Architettura 32-bit:** Mark Word (4) + Class Pointer (4) = **8 byte**. Essendo un multiplo di 8, non serve padding.
2. **Architettura 64-bit (Standard - Compressed Oops attivi):** Mark Word (8) + Class Pointer (4) = 12 byte. Poiché 12 non è multiplo di 8, la JVM aggiunge 4 byte di *Padding*. **Totale: 16 byte**.
3. **Architettura 64-bit (Senza compressione):** Mark Word (8) + Class Pointer (8) = **16 byte**. Nessun padding necessario.

Stack Memory (La "Pila")	Heap Memory (Il "Mucchio")
Usata per l'esecuzione dei metodi e il flusso del codice.	Usata per l'allocazione dinamica degli oggetti.
Memorizza variabili locali primitive (`int`, `double`) e i **riferimenti** agli oggetti.	Memorizza gli **Oggetti** reali (il contenuto effettivo, es. `new String("...")`).
Gestione LIFO (Last In, First Out).	Gestione complessa tramite Garbage Collector.
Thread-Safe: Ogni Thread ha il proprio Stack privato.	**Non Thread-Safe**: Lo Heap è condiviso tra tutti i Thread (necessita sincronizzazione).
Veloce, dimensione limitata (rischio `StackOverflowError`).	Più lento, dimensione grande (rischio `OutOfMemoryError`).

Esempio Pratico:

```
void metodo() {
    int x = 10;            // 'x' e 10 sono nello Stack
    Person p = new Person();
    // Il riferimento 'p' e' nello Stack.
    // L'oggetto reale 'Person' (con i suoi campi) e' nello Heap.
}
```

5.5 Garbage Collection: Algoritmi e Tuning

La Garbage Collection (GC) è il processo automatico di gestione della memoria nello Heap. Identifica gli oggetti non più utilizzati e libera spazio. Un oggetto diventa "eleggibile" per il GC quando non è più **raggiungibile** (Unreachable) da nessun riferimento attivo nello Stack (GC Roots).

5.5.1 GC Roots e Stack: Il punto di ancoraggio

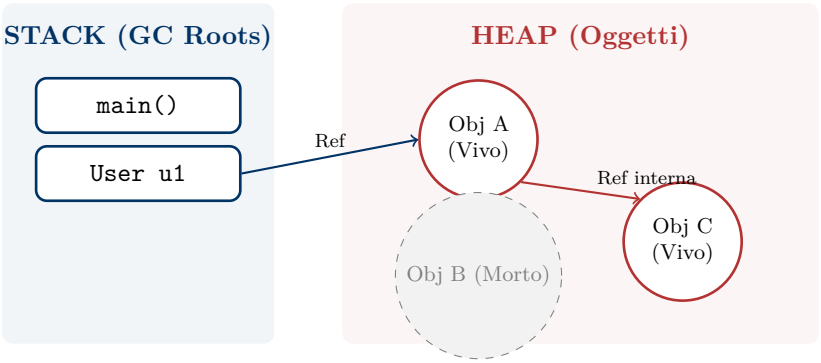
Per capire cosa sia una **GC Root**, dobbiamo guardare la struttura della memoria di un Thread Java.

1. Lo Stack (La pila di esecuzione) Ogni volta che esegui un metodo, Java crea un "Frame" nello **Stack**. Qui risiedono le variabili locali (es. `int x`, `User u`).

- Le variabili primitive contengono il valore direttamente.
- Le variabili oggetto (es. `User u`) **NON** contengono l'oggetto. Contengono solo un **indirizzo di memoria** (un puntatore).

2. Le GC Roots Quel puntatore che risiede nello Stack è una **GC Root**. È un "morsetto" che tiene ancorato l'oggetto che si trova nell'Heap. Finché il metodo è in esecuzione, quel puntatore esiste sullo Stack e l'oggetto nell'Heap è considerato "vivo".

3. Rappresentazione Grafica Ecco come il GC "vede" la memoria. Nota come l'Oggetto B è morto perché nessuno Stack lo sta puntando.



Analisi del Grafico:

1. **Oggetto A:** È vivo perché puntato direttamente da `u1` (che è nello Stack, quindi è una GC Root).
2. **Oggetto C:** È vivo per transizione. Lo Stack punta ad A, e A punta a C. Il GC segue la catena.
3. **Oggetto B:** È morto. Esiste in memoria (occupa spazio), ma nessuna freccia parte dallo Stack per arrivare a lui, né direttamente né indirettamente. Al prossimo Sweep verrà rimosso.

5.5.2 Dietro le quinte: Come decide il GC? (Mark & Sweep)

Per stabilire chi deve morire, il Garbage Collector non controlla gli oggetti uno per uno chiedendo "ti serve ancora?". Sarebbe troppo lento. Invece, usa un algoritmo chiamato **Mark & Sweep** (Marca e Spazza), che funziona in due fasi precise:

FASE 1: MARK (Marcatura - "Chi è vivo?")

Il GC fa una "fotografia" istantanea della memoria e si pone una domanda: *"Quali oggetti sto sicuramente usando in questo preciso istante?"*. Questi oggetti sicuri si chiamano **GC Roots** (Radici) e sono essenzialmente le variabili dei metodi che sono in esecuzione ora.

Il GC agisce come un verniciatore:

1. Parte dalle **Radici** e le "colora" di vivo.
2. Guarda dentro questi oggetti: se puntano ad altri oggetti, va lì e colora anche quelli.
3. Continua a seguire i riferimenti a catena (gli amici degli amici) finché non ha trovato e colorato tutto ciò che è collegato.

Alla fine di questa fase, tutto ciò che è raggiungibile è stato "marcato".

FASE 2: SWEEP (Pulizia - "Elimina il resto")

Ora il GC guarda tutta la memoria rimanente. Tutti gli oggetti che **non sono stati colorati** significano una cosa sola: non c'è nessuna strada che parte dal codice in esecuzione per arrivare a loro. Sono isolati. Quindi, vengono spazzati via brutalmente.

Esempio pratico: Se hai `Utente u = new Utente();`, il GC parte dalla variabile `u` (Radice) e marca l'oggetto in memoria. Se poi scrivi `u = null;`, spezzi l'unica strada. Al prossimo passaggio, il GC partirà dalle radici, non troverà più la strada per quell'oggetto, non lo marcherà, e nella fase di Sweep lo cancellerà.

Deep Dive: Stop-The-World Events

Durante alcune fasi del GC, la JVM deve mettere in pausa *tutti* i thread dell'applicazione (**Stop-The-World**). Se il GC parte troppo spesso o dura troppo a lungo, l'applicazione "si congela" (latenza alta). L'obiettivo del tuning è ridurre la durata di queste pause.

5.5.3 Le generazioni dello Heap (Young, Old, Metaspace)

La JVM non gestisce la memoria come un blocco unico, ma la divide in zone basandosi sull'*Ipotesi Generazionale Debole* (*Weak Generational Hypothesis*), che afferma una verità statistica fondamentale:

> *"La maggior parte degli oggetti muore giovane, mentre quelli che sopravvivono tendono a restare in vita a lungo."*

Di conseguenza, lo Heap è diviso in tre aree principali per ottimizzare il Garbage Collection (GC).

1. Young Generation (La "Nursery")

È l'area dove avvengono le allocazioni e le prime fasi di vita degli oggetti. È progettata basandosi sull'ipotesi che *la maggior parte degli oggetti muore giovane*.

- **Eden Space:** È il punto di ingresso. Ogni volta che esegui `new Object()`, l'allocazione avviene qui.
- **Survivor Spaces (S0 e S1):** Fungono da "filtro temporale". Non vengono mai usati contemporaneamente: in ogni momento uno è attivo (*From*) e l'altro è vuoto (*To*).
  1. Quando avviene un Minor GC, gli oggetti ancora vivi nell'Eden e nel Survivor attivo (*From*) vengono copiati nel Survivor vuoto (*To*).
  2. Una volta copiati, l'Eden e il *From* vengono svuotati completamente.
  3. I ruoli si invertono: chi era *To* diventa *From* e viceversa.
  4. Ad ogni passaggio, l'età dell'oggetto (age) viene incrementata.
- **Tipo di GC:** Qui avviene il **Minor GC**. È molto frequente ed estremamente veloce perché la JVM copia solo gli oggetti vivi (pochi) e ignora quelli morti (tanti).

Deep Dive: Perché il "Ping-Pong" tra S0 e S1 è utile?

Questa pratica di travasamento continuo serve a due scopi principali:

1. **Compattazione Gratuita (No Frammentazione):** Invece di cercare i "buchi" lasciati dagli oggetti morti nell'Eden, la JVM copia gli oggetti vivi uno accanto all'altro nel nuovo spazio S1 (o S0). Questo mantiene la memoria contigua e rende l'allocazione futura rapidissima.
2. **Aging (Invecchiamento):** I Survivor Spaces agiscono come un buffer per ritardare la promozione alla *Old Generation*. Se un oggetto sopravvive a un certo numero di scambi (solitamente 15, definito dal *Tenuring Threshold*), viene considerato stabile e promosso. Senza S0/S1, ogni oggetto sopravvissuto all'Eden finirebbe subito nella Old Gen, riempiendola di spazzatura temporanea e causando frequenti *Full GC* (che sono lenti).

2. Old Generation (Tenured)

Questa è l'area di memoria destinata agli oggetti a lunga vita. Solitamente è dimensionalmente più grande della Young Generation e ospita componenti strutturali dell'applicazione (es. Cache, Singleton di Spring, Connection Pools, Sessioni Utente attive).

- **Promozione (Tenuring):** Un oggetto arriva qui principalmente in due modi:
  1. Ha superato la soglia di età massima (Max Tenuring Threshold) sopravvivendo a numerosi cicli nella Young Gen.
  2. È troppo grande per essere allocato nell'Eden (allocazione diretta in Old Gen per evitare copie costose).
- **Tipo di GC:** Qui avviene il **Major GC** (spesso parte di un Full GC).
  - È un'operazione molto più lenta rispetto al Minor GC.
  - Spesso comporta uno **"Stop-The-World"**: l'applicazione si blocca completamente finché la pulizia non è terminata.
  - Deve gestire la **frammentazione**: mentre nella Young Gen si copiano gli oggetti (veloce), qui spesso si usa un algoritmo *Mark-Sweep-Compact* (segna, pulisce e ricompatta i buchi), che richiede molto calcolo CPU.

Deep Dive: Il Ciclo di Vita Completo di un Oggetto

Per visualizzare l'intero viaggio di un dato all'interno della Heap, seguiamo questo flusso:

1. **Nascita (Allocation):** L'oggetto viene istanziato nell'`Eden` (la maggior parte delle volte).
2. **Sopravvivenza (Minor GC):** Al primo GC, se l'oggetto è ancora referenziato, viene salvato e spostato nel *Survivor Space* attivo (es. S0). La sua "età" diventa 1.
3. **Invecchiamento (Aging):** Ai successivi Minor GC, l'oggetto fa "ping-pong" tra S0 e S1, incrementando la sua età ad ogni passaggio.
4. **Promozione (Promotion):** Una volta raggiunta la soglia critica (es. 15 cicli), viene considerato stabile e promosso nella **Old Generation**.
5. **Morte (Major GC):** Se l'oggetto diventa irraggiungibile (garbage) mentre risiede nella Old Gen, non verrà rimosso subito. Rimarrà lì ad occupare spazio fino al prossimo, costoso, Major GC.

3. Metaspace (Memoria Nativa)

Introdotto in Java 8 per sostituire la vecchia *PermGen* (Permanent Generation).

- **Non è nello Heap:** Risiede nella *Native Memory* del Sistema Operativo.
- **Cosa contiene:** I metadati delle classi (il "DNA" del codice: definizioni di metodi, nomi di variabili statiche, constant pool). In maniera sintetica le definizioni.
- **Vantaggio:** A differenza della PermGen che aveva una dimensione fissa (causando spesso `OutOfMemoryError`), il Metaspace può crescere automaticamente fino al limite della RAM del server (se non limitato esplicitamente).

Colloquio: PermGen vs Metaspace e Memory Leak

Domanda: "Qual è la differenza tra Heap e Metaspace e quando si riempiono?"

Risposta:

- Lo **Heap** contiene le *istanze* (i dati). Si riempie se crei troppi oggetti senza rilasciarli (es. una `List` statica che cresce all'infinito). Errore: `java.lang.OutOfMemoryError: Java heap space`.
- Il **Metaspace** contiene le *classi* (il codice). Si riempie se carichi troppe classi diverse

dinamicamente (comune con framework che usano molta Reflection o CGLIB come Spring o Hibernate). Errore: `java.lang.OutOfMemoryError: Metaspace`.

Capitolo 6

Il Ciclo di Vita del Software: Dal Codice al Runtime

Molti sviluppatori junior pensano che il lavoro finisca quando il codice compila nell'IDE. In realtà, quello è solo l'inizio. Per diventare un Software Engineer, devi comprendere l'intero viaggio che il codice compie: da semplice testo scritto da un umano, a istruzioni macchina eseguite da un processore su un server remoto.

In questo capitolo dissezioneremo questo processo, analizzando il Build System, la gestione delle dipendenze e l'ambiente di esecuzione (Runtime).

6.1 Il formato JAR e il processo di Build

Prima di poter avviare un'applicazione, il codice sorgente deve essere trasformato in un artefatto distribuibile. In Java, lo standard *de facto* è il file **JAR** (Java ARchive).

6.1.1 Che cos'è un file JAR?

Sotto il cofano, un file `.jar` non è altro che un archivio **ZIP** che contiene:

- I file `.class` (il bytecode compilato).
- Risorse statiche (immagini, file di configurazione `.properties` o `.xml`).
- I metadati all'interno della cartella `META-INF`.

Puoi letteralmente rinominare un file `app.jar` in `app.zip` e aprirlo con WinRAR o il gestore archivi del tuo OS per ispezionarne il contenuto.

6.1.2 Compilazione e Packaging Manuale

Per capire come funzionano i moderni strumenti di build, è utile vedere come si faceva "alla vecchia maniera" da riga di comando.

1. Compilazione

Il compilatore `javac` trasforma il codice leggibile dall'uomo in bytecode.

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

2. Il Manifesto (META-INF/MANIFEST.MF)

Questo è il cuore dell'eseguibile. Senza questo file (o senza configurarlo correttamente), la JVM non saprà quale classe contiene il metodo `main` da avviare.

Un file `MANIFEST.MF` minimale per un eseguibile appare così:

```
Manifest-Version: 1.0
Main-Class: com.example.Main
Created-By: 17.0.1 (Oracle Corporation)
```

Listing 6.1: Esempio di MANIFEST.MF

3. Packaging

Il comando `jar` impacchetta tutto insieme:

```
jar cvfm app.jar MANIFEST.MF -C out .
```

Dove le flag stanno per: **c**reate, **v**erbose, **f**ile (nome output), **m**anifest (file input).

6.1.3 Build Moderni: Fat JAR vs Thin JAR

Nel mondo enterprise moderno, raramente userai `javac` a mano. Strumenti come **Maven** o **Gradle** automatizzano il processo. Tuttavia, sorge un problema architetturale: come gestiamo le dipendenze esterne (librerie)?

Qui entra in gioco la distinzione fondamentale tra Fat JAR e Thin JAR.

Deep Dive: Fat JAR (o Uber-JAR) vs Thin JAR

Questa distinzione è cruciale quando si lavora con **Docker** e **Microservizi**.

Thin JAR (Il classico)

Contiene *solo* le tue classi e risorse. Le dipendenze esterne (es. driver del DB, Spring Framework) devono essere presenti nel *Classpath* della macchina che esegue il codice o in una cartella `lib` separata.

- **Pro:** Il file è piccolissimo (pochi KB). Ottimo per aggiornamenti incrementali.
- **Contro:** "Dependency Hell". È difficile assicurarsi che l'ambiente di produzione abbia le stesse versioni delle librerie dell'ambiente di sviluppo.

Fat JAR / Uber-JAR (Lo standard Cloud Native)

Contiene le tue classi **PIÙ** tutte le dipendenze (tutti i jar delle librerie vengono estratti e rimpacchettati dentro, o inseriti come jar annidati). È l'approccio standard di **Spring Boot**.

- **Pro:** "Self-contained". Basta avere la JVM installata e l'app gira ovunque (‘java -jar app.jar‘). Nessuna configurazione del classpath necessaria.
- **Contro:** Dimensioni maggiori (50MB+ per un Hello World Spring Boot).

Colloquio: Differenza tra library JAR ed executable JAR

Domanda: "Ho un file .jar, ma quando provo a lanciarlo mi da errore 'no main manifest attribute'. Perché?"

Risposta: Stai cercando di eseguire un JAR configurato come **Libreria**, non come **Eseguibile**.

- Un **Executable JAR** ha nel suo `META-INF/MANIFEST.MF` l'attributo `Main-Class` che punta alla classe contenente il `public static void main`.

- Un **Library JAR** è solo un contenitore di classi pensato per essere usato da altre applicazioni. Non ha un punto di ingresso (entry point).

Colloquio: ClassNotFoundException vs NoClassDefFoundError

- **ClassNotFoundException (Checked Exception):** Avviene quando provi a caricare una classe dinamicamente a runtime (usando Reflection o `Class.forName("...")`) e il nome fornito non esiste nel classpath. È un errore "esplicito".
- **NoClassDefFoundError (Error - Unchecked):** Molto più subdolo. Succede quando la classe c'era durante la compilazione (quindi il codice compilava), ma **non c'è più** al momento dell'esecuzione (runtime). Esempio tipico: Hai compilato usando la libreria A versione 2.0, ma sul server c'è la libreria A versione 1.0 che manca di quella classe. È un problema di configurazione ambientale o dipendenze corrotte (JAR Hell).

6.2 Il Build System: Perché usiamo Maven o Gradle?

Immagina di voler costruire una casa. Non forgi i chiodi a mano e non tagli gli alberi per fare le assi. Vai dal fornitore e compri materiali pronti. Nello sviluppo software, raramente scrivi tutto da zero. Usi librerie esterne (es. driver per il database, librerie per generare PDF, framework come Spring).

6.2.1 L'incubo del Classpath manuale

Senza un Build Tool, dovresti:

1. Andare sui siti delle librerie e scaricare i file `.jar`.
2. Metterli in una cartella `lib/`.
3. Dire a Java di includerli nel Classpath manualmente.

Esempio Pratico Visuale:
Immagina di avere una struttura di progetto organizzata in questo modo sul tuo computer:

```
IlTuoProgetto/
|-- src/
|   `-- Main.java          (Il tuo codice)
`-- lib/
    `-- commons-lang3.jar  (La libreria esterna scaricata)
```

Listing 6.2: Struttura delle directory

Per compilare ed eseguire il codice includendo manualmente il file `.jar`, dovrai usare il flag `-cp` (o `-classpath`) nel terminale:

```
# 1. COMPILAZIONE
# Diciamo a javac: "Guarda anche nella cartella lib/ per trovare le classi"
javac -cp "lib/commons-lang3.jar" src/Main.java

# 2. ESECUZIONE
# Diciamo a java: "Il classpath è composto dalla cartella corrente (.)
# PIÙ il jar esterno"
# NOTA: I separatori cambiano in base al Sistema Operativo (vedi sotto)
```

```
9 java -cp ".;lib/commons-lang3.jar" src.Main
```

Listing 6.3: Compilazione ed Esecuzione da Terminale

Deep Dive: Attenzione ai Separatori del Classpath!

Quando concateni più percorsi nel Classpath, il separatore cambia a seconda del sistema operativo. Questo è spesso causa di errori "Class Not Found" quando si sposta codice da Windows a Linux.

- **Windows**: Usa il punto e virgola (`;`)
  Es: `-cp ".;lib/libreria.jar"`
- **Mac/Linux**: Usa i due punti (`:`)
  Es: `-cp ".:lib/libreria.jar"`

Ma c'è un problema peggiore: le **Dipendenze Transitive**. Se usi la libreria A, e la libreria A ha bisogno della libreria B, devi scaricare anche B. E se B ha bisogno di C? È l'inferno delle dipendenze ("Dependency Hell").

6.2.2 Maven al salvataggio

Maven (o Gradle) risolve questo problema.

- **POM (Project Object Model)**: È il file `pom.xml`. Qui dichiari **cosa** vuoi (es. "Voglio Spring Web versione 3.0").
- **Central Repository**: È un magazzino online gigantesco che contiene quasi tutte le librerie Java del mondo.
- **Gestione Automatica**: Maven scarica la libreria che hai chiesto, controlla di cosa ha bisogno quella libreria (dipendenze transitive) e scarica pure quelle.

Colloquio: Intervista: Cos'è il ciclo di vita di Maven?

Maven non serve solo a scaricare jar. Gestisce l'intero ciclo di vita del build tramite fasi standard:

1. **Clean:** Pulisce i vecchi file compilati (cartella `target`).
2. **Compile:** Compila i sorgenti `.java` in `.class`.
3. **Test:** Esegue gli Unit Test (JUnit). Se falliscono, il build si ferma.
4. **Package:** Impacchetta i file compilati in un artefatto distribuibile (JAR o WAR).
5. **Install:** Copia l'artefatto nel tuo repository locale, rendendolo disponibile ad altri tuoi progetti.

6.3 Packaging: JAR vs WAR (e la rivoluzione Spring Boot)

Una volta che il codice è compilato e testato, dobbiamo impacchettarlo per spedirlo al server. Esistono due formati principali.

6.3.1 WAR (Web ARchive) - L'approccio Legacy

Fino a pochi anni fa (era pre-Cloud), l'architettura standard era:

- Compri un server fisico potente.
- Installi sopra un **Application Server** (come Tomcat, JBoss, WebLogic). Questo server è sempre acceso.
- Il tuo build produce un file **.war**.

- "Deploayi" il file .war dentro l'Application Server.

Analogia: È come un condominio (Application Server). Tu porti solo i mobili (il file WAR) e abiti in un appartamento pre-costruito.

6.3.2 JAR (Java ARchive) - L'approccio Moderno (Cloud Native)

Spring Boot ha reso popolare l'approccio "Fat Jar" (o Uber Jar).

- Il build produce un file **.jar**.
- Questo file contiene il tuo codice, le tue dipendenze E ANCHE il server web (Tomcat Embedded).
- Per avviarlo non serve installare Tomcat sulla macchina. Basta avere Java: `java -jar miaopp.jar`.

Analogia: È come un Camper. Ha il motore e la casa tutti insieme. È autonomo e puoi spostarlo ovunque (Docker, Kubernetes) senza dipendere dall'infrastruttura esterna.

6.4 Avvio di un'applicazione Java: Sotto il cofano

Capire cosa succede esattamente quando digitiamo `java -jar app.jar` nel terminale è fondamentale per un ingegnere Back-End. Non si tratta solo di eseguire codice, ma di attivare una complessa macchina virtuale che gestisce memoria, sicurezza e ottimizzazioni in tempo reale.

Il ciclo di vita dell'avvio può essere suddiviso in fasi distinte: dal caricamento del binario del sistema operativo fino all'esecuzione del metodo `main`.

6.4.1 Il comando `java` e la JNI

Quando lanci il comando:

```
java -jar app.jar
```

Il Sistema Operativo avvia un processo. Questo processo non è ancora la tua applicazione, ma il **launcher** della JVM (scritto solitamente in C/C++).

- Il launcher analizza gli argomenti della riga di comando (es. opzioni `-Xmx`, `-Dproperty`).
- Carica la libreria dinamica della JVM (es. `jvm.dll` su Windows o `libjvm.so` su Linux).
- Inizializza la **Java Native Interface (JNI)**, il ponte che permette al mondo C++ di creare un'istanza della Java Virtual Machine.

6.4.2 Class Loading: Portare il Codice in Vita

Per capire il *Class Loading*, dobbiamo prima sfatare un mito: quando avvii un programma Java, la JVM **non** carica tutto il tuo codice in memoria immediatamente. Se il tuo progetto ha 10.000 classi ma ne usi solo 3, caricarle tutte sarebbe uno spreco enorme di RAM e tempo di avvio.

Cos'è quindi il Class Loading?

Immagina i file `.class` (il bytecode compilato) come dei libri chiusi in una biblioteca (il tuo Hard Disk). La JVM è lo studioso.

1. Il "Caricamento" avviene quando lo studioso prende un libro dallo scaffale e lo apre sulla scrivania (la Memoria RAM).
2. Solo quando il libro è aperto, lo studioso può leggerne le istruzioni e creare oggetti.

Questo approccio si chiama **Lazy Loading** (Caricamento Pigro): Java carica una classe in memoria solo nell'esatto istante in cui viene menzionata per la prima volta nel codice (es. quando fai `new MioOggetto()`).

Deep Dive: Come funziona la Gerarchia (Chi carica cosa?)

Java non butta tutte le classi in un unico calderone. Usa dei "magazzinieri" specializzati chiamati **ClassLoader**, organizzati gerarchicamente.

Il principio fondamentale è la **Delega al Genitore**: quando serve una classe, il magazziniere più in basso (il tuo) chiede prima al suo capo se ce l'ha lui. Se il capo ce l'ha, bene. Se il capo non ce l'ha, il magazziniere prova a cercarla da solo.

Ecco i tre livelli, dal "Capo Supremo" all'ultimo arrivato:

1. **Bootstrap ClassLoader (Il Capo Supremo)** Carica le classi vitali per la sopravvivenza di Java (es. `String`, `Integer`, `List`).
   - **Curiosità:** È scritto in C++ (nativo), non in Java. Per questo, se provi a stampare il ClassLoader di una Stringa, ottieni `null`: la JVM non sa rappresentarlo come oggetto Java.
2. **Platform ClassLoader (Il Manager)** È figlio del Bootstrap. Carica le librerie standard aggiuntive che fanno parte del JDK ma non del nucleo vitale (es. `java.sql.*` per i database o classi XML).
3. **Application ClassLoader (L'Operaio)** È figlio del Platform. Questo è quello che carica **il tuo codice** e le librerie esterne (i `.jar`) che hai aggiunto al progetto (il famoso *Classpath*).

Colloquio: Perché la delega è importante?

Domanda tipica: *"Perché Java usa questo modello a delega?"*
Risposta: Per **Sicurezza**. Se tu creassi nel tuo progetto una classe maligna chiamandola `java.lang.String`, il sistema di delega proteggerebbe la JVM.

Quando il tuo codice chiede "dammi `String`", la richiesta sale fino al *Bootstrap ClassLoader*. Lui troverà la *vera* Stringa di Java e restituirà quella, ignorando completamente la tua versione maligna. Senza delega, potresti sovrascrivere le classi base del linguaggio!

Esempio Pratico: Questo codice interroga le classi per chiedere: "Chi ti ha portato in memoria?".

```
1  public class ChiMiHaCaricato {
2      public static void main(String[] args) {
3
4          // 1. Chiediamo a una classe creata da noi
5          // Risposta attesa: AppClassLoader (perché è codice nostro)
6          System.out.println("Mio Codice: " +
7              ChiMiHaCaricato.class.getClassLoader());
8
9          // 2. Chiediamo a una libreria standard (es. SQL)
10         // Risposta attesa: PlatformClassLoader (estensione JDK)
11         System.out.println("SQL: " +
12             java.sql.SQLData.class.getClassLoader());
13
14         // 3. Chiediamo a una classe base (String)
15         // Risposta attesa: null (Bootstrap, scritto in C++)
16         System.out.println("String: " +
```

```
            String.class.getClassLoader());
    }
}
```

Listing 6.4: Analisi dei ClassLoader a Runtime

6.4.3 Linking (Collegamento)

Dopo che una classe è stata caricata (il file `.class` è stato letto), deve passare attraverso la fase di Linking, divisa in tre sotto-fasi:

1. **Verification (Verifica)**

   È la fase più complessa e importante per la sicurezza. Il *Bytecode Verifier* controlla che il codice non violi le regole della JVM (es. non saltare a istruzioni inesistenti, non causare overflow dello stack, rispetto dei tipi). Se questa fase fallisce, viene lanciato un `java.lang.VerifyError`.

2. **Preparation (Preparazione)**

   La JVM alloca la memoria per le variabili statiche della classe e le inizializza ai valori di default (0 per gli interi, `null` per i riferimenti, `false` per i booleani). **Nota:** In questa fase non viene ancora eseguito il tuo codice di inizializzazione, solo i default della JVM.

3. **Resolution (Risoluzione)**

   I riferimenti simbolici nel bytecode (nomi di classi, interfacce, campi) vengono sostituiti con riferimenti diretti alla memoria (puntatori).

6.4.4 Initialization (Inizializzazione)

Questa è la prima fase in cui il tuo codice viene effettivamente eseguito. La JVM esegue il blocco di inizializzazione statico della classe (`<clinit>`).

```
public class StartupExample {
    // 1. Questo viene eseguito durante la fase di Initialization
    static {
        System.out.println("Blocco statico: Classe caricata e inizializzata.");
    }

    // 2. Il main viene chiamato solo DOPO l'inizializzazione della classe
    public static void main(String[] args) {
        System.out.println("Metodo Main: Inizio esecuzione applicazione.");
    }
}
```

Listing 6.5: Ordine di esecuzione all'avvio

6.4.5 Esecuzione: Interprete e JIT

Dopo aver inizializzato la classe principale, la JVM invoca il metodo `public static void main(String[] args)`. Inizialmente, il bytecode viene letto ed eseguito da un **Interprete** (riga per riga). È veloce da avviare, ma lento nell'esecuzione prolungata.

Qui entra in gioco il **Just-In-Time (JIT) Compiler**:

- La JVM monitora quali metodi vengono eseguiti più spesso ("Hot Spots").
- Il JIT compila questi "pezzi caldi" di bytecode in codice macchina nativo ottimizzato per la CPU specifica.
- Da quel momento in poi, viene eseguito direttamente il codice nativo, bypassando l'interprete.

Colloquio: Java è compilato o interpretato?

Questa è una domanda trabocchetto classica. La risposta corretta è: **Entrambi**.

- Il codice sorgente (`.java`) viene **compilato** staticamente da `javac` in Bytecode (`.class`).
- Il Bytecode è intermedio e indipendente dalla piattaforma.
- A runtime, la JVM agisce come un **interprete** per il bytecode.
- Tuttavia, grazie al compilatore **JIT** (Just-In-Time), le parti di codice eseguite frequentemente vengono compilate dinamicamente in codice macchina nativo per massimizzare le performance.

6.4.6 Shutdown (Terminazione)

L'applicazione Java continua a girare finché esiste almeno un **Non-Daemon Thread** attivo.

- Il thread `main` è un thread non-daemon.
- Se il `main` finisce, ma hai avviato altri thread utente (non-daemon), la JVM non si chiude.
- Quando l'ultimo thread non-daemon termina, la JVM inizia la procedura di shutdown, eseguendo eventuali *Shutdown Hooks* (usati ad esempio da Spring Boot per chiudere le connessioni al database gentilmente) e rilasciando la memoria al sistema operativo.

Deep Dive: System.exit(0) vs Runtime Exception

- `System.exit(n)`: Forza la chiusura immediata della JVM. L'argomento `n` è lo status code (0 = ok, altro = errore). Questo comando ferma tutto, anche se ci sono thread attivi.
- `RuntimeException`: Se un'eccezione non gestita risale fino al `main`, quel thread muore. Se è l'unico thread non-daemon, la JVM si spegne con uno stack trace.

Capitolo 7

Variabili, Tipi di Dati e Operatori

Una comprensione superficiale dei tipi di dati porta a bug subdoli (come `NullPointerException` o errori di precisione). In questo capitolo analizziamo come Java gestisce i dati a basso livello.

7.1 Tipi Primitivi vs Reference: La gestione in memoria

Java è un linguaggio **Strongly Typed**: ogni variabile deve avere un tipo dichiarato. Esistono due macro-categorie:

7.1.1 Tipi Primitivi (Primitive Data Types)

Sono i "mattoni" del linguaggio. Non sono oggetti, contengono direttamente il valore nello Stack e hanno performance elevate.

Tipo	Bit	Range / Valore	Default (Campi)
`boolean`	-	`true`, `false`	`false`
`byte`	8	-128 a 127	0
`short`	16	-32,768 a 32,767	0
`char`	16	Unicode (unsigned)	`'\u0000'`
`int`	32	$\approx \pm 2$ miliardi	0
`long`	64	Molto grande (suffisso `L`)	0L
`float`	32	Virgola mobile (suffisso `f`)	0.0f
`double`	64	Virgola mobile doppia prec.	0.0d

Tabella 7.1: Tipi Primitivi in Java

Regola di Inizializzazione:

- **Campi di istanza (Fields):** Se non inizializzati, assumono il valore di default (vedi tabella).
- **Variabili Locali (dentro i metodi): Non** hanno valore di default. Il compilatore darà errore se provi a usarle senza aver assegnato un valore.

7.2 Numeri Decimali: Float, Double e BigDecimal

In Java, i tipi primitivi per i numeri con la virgola sono `float` (32 bit) e `double` (64 bit). Tuttavia, il loro comportamento nasconde insidie che ogni sviluppatore Back-End deve conoscere, specialmente quando si tratta di precisione.

Deep Dive: Sotto il cofano: IEEE 754 e la Mantissa

I computer non salvano i numeri decimali come facciamo noi (es. 12.5), ma utilizzano uno standard chiamato **IEEE 754**. Questo standard rappresenta i numeri in notazione scientifica binaria:

$$V = (-1)^S \times M \times 2^E$$

Dove:

- **S (Sign):** 1 bit per il segno (+ o -).
- **E (Exponent):** L'esponente che "sposta" la virgola (8 bit per float, 11 per double).
- **M (Mantissa):** La parte frazionaria significativa (23 bit per float, 52 per double).

Il problema della precisione: Così come in base 10 non possiamo scrivere esattamente 1/3 (che diventa 0.3333...), in base 2 (binario) non è possibile rappresentare esattamente numeri apparentemente semplici come **0.1** o **0.2**. Questi numeri diventano periodici in binario. Poiché la mantissa ha un numero finito di bit, il computer è costretto a "tagliare" e arrotondare il numero, introducendo un piccolo errore di approssimazione.

7.2.1 Perché usiamo la 'f'?

In Java, qualsiasi numero decimale scritto nel codice (es. `3.14`) viene interpretato di default come **double**. Un `double` occupa 64 bit, mentre un `float` ne occupa solo 32. Se proviamo ad assegnare un double a un float senza specificarlo, il compilatore da errore perché stiamo cercando di infilare un dato grande in un contenitore piccolo (possibile perdita di dati). Il suffisso `f` (o `F`) dice esplicitamente al compilatore: *"Tratta questo letterale come un float a 32 bit"*.

```
// Errore di compilazione: 3.14 e' double
float pi = 3.14;

// Corretto: forziamo il tipo float
float pi = 3.14f;
```

7.2.2 Il problema di Float e Double

A causa del funzionamento della mantissa spiegato sopra, le operazioni aritmetiche possono dare risultati inaspettati.

```
double a = 0.1;
double b = 0.2;
double somma = a + b;

System.out.println(somma);
// STAMPA: 0.30000000000000004 (Non e' 0.3!)
```

Listing 7.1: Errore di precisione aritmetica

7.2.3 La Soluzione: BigDecimal

Per calcoli che richiedono precisione assoluta (come calcoli finanziari, tasse, coordinate GPS precise), Java offre la classe `java.math.BigDecimal`.

Cosa ha di speciale?

- **Precisione Arbitraria:** Non usa l'approssimazione binaria IEEE 754. Memorizza il numero come un *intero non scalato* (un `BigInteger` enorme) e un intero che rappresenta

la *scala* (dove mettere la virgola). È come se memorizzasse "123" e "sposta la virgola di 2 a sinistra" per ottenere 1.23.

- **Immutabilità:** Come le Stringhe, le operazioni su BigDecimal non modificano l'oggetto corrente ma ne restituiscono uno nuovo.
- **Controllo sugli arrotondamenti:** Permette di specificare esattamente come arrotondare (per eccesso, per difetto, verso pari, ecc.).

```
// ATTENZIONE: Usare sempre il costruttore Stringa!
// new BigDecimal(0.1) porterebbe dentro l'errore del double.
BigDecimal bd1 = new BigDecimal("0.1");
BigDecimal bd2 = new BigDecimal("0.2");

BigDecimal risultato = bd1.add(bd2);

System.out.println(risultato); // STAMPA: 0.3
```

Listing 7.2: Uso corretto di BigDecimal

Colloquio: Float/Double vs BigDecimal per i soldi

Domanda: "In un e-commerce, quale tipo di dato useresti per rappresentare il prezzo dei prodotti?"

Risposta: "Non userei mai `float` o `double` a causa degli errori di arrotondamento della virgola mobile (IEEE 754), che su grandi volumi portano a perdite finanziarie. Utilizzerei `BigDecimal` per la massima precisione decimale, oppure in alcuni casi un `long` rappresentando il valore in centesimi (es. 1000 invece di 10.00) per performance migliori, convertendo solo in fase di visualizzazione."

7.3 Gestire i Soldi con Long (Centesimi)

Una valida alternativa a `BigDecimal` per le performance è l'uso dei tipi interi (`long`), rappresentando tutto nell'unità più piccola della valuta (es. centesimi per Euro/Dollari, Yen interi, Satoshi per Bitcoin).

Esempio: €10.50 diventa `1050` (long).

Tuttavia, questo approccio si scontra con un muro matematico durante le divisioni: la divisione intera tronca i decimali.

7.3.1 Il problema del centesimo sparito

Immaginiamo di dover dividere €1.00 (100 centesimi) tra 3 persone.

```
long totale = 100;
long persone = 3;

long quota = totale / persone;
// Risultato: 33 (divisione intera)

long sommaFinale = quota * 3;
// Risultato: 99. Manca 1 centesimo!
```

In finanza, il denaro non può essere distrutto né creato. Quel centesimo deve andare a qualcuno.

Deep Dive: Algoritmo di Allocazione del Resto

Quando la divisione non è intera, la soluzione standard è calcolare il **resto** (operatore modulo `%`) e distribuirlo un centesimo alla volta ai beneficiari finché non finisce.

La formula logica è:

1. **Base Share:** `totale / n` (quanto spetta a tutti come minimo).
2. **Remainder:** `totale % n` (i centesimi avanzati).
3. **Distribuzione:** I primi *Remainder* utenti ricevono `Base + 1`, gli altri solo `Base`.

Ecco come implementare una funzione utility sicura per dividere importi monetari interi:

```
/**
 * Divide un importo in centesimi in N parti.
 * Distribuisce il resto ai primi beneficiari.
 */
public static long[] dividiImporto(long amount, int parti) {
    long[] risultati = new long[parti];

    long quotaBase = amount / parti; // Es. 100 / 3 = 33
    long resto = amount \% parti;     // Es. 100 % 3 = 1

    for (int i = 0; i < parti; i++) {
        // Assegna la quota base
        risultati[i] = quotaBase;

        // Se c'e' ancora resto, dai 1 centesimo in piu' a questo utente
        if (resto > 0) {
            risultati[i]++;
            resto--;
        }
    }

    return risultati;
    // Ritorna: [34, 33, 33] -> Totale: 100. Nulla e' andato perso.
}
```

Listing 7.3: Divisione Monetaria con Distribuzione del Resto

7.3.2 Percentuali e Moltiplicazioni con i Long

Quando si calcolano percentuali (es. sconti o IVA) usando i `long`, l'ordine delle operazioni è fondamentale: bisogna sempre **moltiplicare prima** e **dividere poi**. Se si dividesse prima, si perderebbe immediatamente la precisione a causa del troncamento intero.

Prendiamo come esempio il calcolo del 15% di €50.05, che in centesimi corrisponde a `5005`:

1. **Moltiplicazione:** $5005 \times 15 = 75075$.
2. **Divisione (Problematico):** $75075/100 = 750$.

Il risultato matematico esatto sarebbe 750.75. La divisione intera di Java tronca sempre verso zero, restituendo `750` (€7.50). Tuttavia, in ambito commerciale, 750.75 dovrebbe essere arrotondato a `751` (€7.51).

La tecnica del "+ Metà Divisore" Per ottenere l'arrotondamento aritmetico classico (*Round Half Up*: da .5 in su si arrotonda per eccesso) lavorando solo con interi positivi, si aggiunge la metà del divisore al numeratore *prima* di dividere. Poiché dividiamo per 100 (percentuale), la metà è 50.

$$\text{Risultato} = \frac{(\text{Valore} \times \text{Percentuale}) + 50}{100}$$

Applicandolo al nostro esempio:

$$\text{Risultato} = \frac{75075 + 50}{100} = \frac{75125}{100} = 751$$

Ora il risultato è **751** centesimi, ovvero l'arrotondamento corretto.

Deep Dive: Nota sulla Terminologia

Questa formula implementa l'**Arrotondamento Commerciale** (Round Half Up). Spesso si confonde con lo "Standard Bancario" (*Round Half to Even*), che invece arrotonda al numero pari più vicino (es. 2.5 diventa 2, 3.5 diventa 4) per ridurre l'accumulo di errori statistici. Per ottenere il vero comportamento bancario o gestire numeri negativi, questa formula semplice non basta e serve una logica condizionale più complessa.

Colloquio: Long vs BigDecimal: Cosa scegliere?

Domanda: "In un sistema ad alta frequenza (HFT), useresti BigDecimal?"

Risposta: "Probabilmente no. `BigDecimal` crea molti oggetti in memoria (è immutabile) e ha un overhead di CPU. In sistemi Real-Time o Low-Latency, si preferisce usare `long` (centesimi o millesimi di centesimo) per sfruttare le istruzioni native della CPU ed evitare la Garbage Collection frequente. Tuttavia, per applicazioni Enterprise standard (gestionali, e-commerce), la sicurezza e la leggibilità di `BigDecimal` vincono sulle micro-ottimizzazioni."

Colloquio: Perché non si usa mai `double` per i soldi?

I tipi a virgola mobile (`float`, `double`) seguono lo standard IEEE 754, che è pensato per la velocità scientifica, non per la precisione assoluta. Esempio classico:

```
double a = 0.1;
double b = 0.2;
System.out.println(a + b); // Stampa 0.3000000000000004
```

Quell'errore infinitesimale, su milioni di transazioni, crea buchi di bilancio. **Soluzione:** Usare sempre **`java.math.BigDecimal`**.

- **Costruttore:** Usare sempre il costruttore **String** o `valueOf`. Mai quello double.

```
    BigDecimal safe = new BigDecimal("0.1"); // Corretto
    BigDecimal unsafe = new BigDecimal(0.1); // Pericoloso (include l'
    ↪ imprecisione del double)
    BigDecimal preferred = BigDecimal.valueOf(0.1); // Corretto (chiama toString
    ↪ internamente)

```

- **Immutabilità:** `BigDecimal` è immutabile (come String). Le operazioni restituiscono nuovi oggetti.

```
    BigDecimal b1 = new BigDecimal("10");
    b1.add(new BigDecimal("5")); // Errore! Il risultato viene perso
    b1 = b1.add(new BigDecimal("5")); // Corretto

```

- **Confronti:** Mai usare `equals()` per confrontare valori numerici, perché controlla anche la scala (numero di decimali).

```
    BigDecimal x = new BigDecimal("1.0");
    BigDecimal y = new BigDecimal("1.00");
    x.equals(y);      // FALSE (Scale diverse: 1 vs 2)
    x.compareTo(y);   // 0 (Sono matematicamente uguali) -> USARE QUESTO

```

7.3.3 Anatomia di BigDecimal: Grandezza e Scala

Spesso si pensa a `BigDecimal` come a un "contenitore magico", ma internamente è composto da due campi molto semplici che lavorano insieme per determinare il valore e la posizione della virgola.

Deep Dive: Struttura Interna: Unscaled Value e Scale

Un oggetto `BigDecimal` è definito da questa formula matematica esatta:

$$\text{Valore} = \text{UnscaledValue} \times 10^{-\text{Scale}}$$

Internamente, Java memorizza questi due dati:

1. **Unscaled Value (Valore non scalato):** Un oggetto `BigInteger`. Questo contiene il numero "intero", ignorando completamente la virgola. Essendo un `BigInteger`, può avere una grandezza arbitraria, limitata solo dalla RAM del computer.
2. **Scale (Scala):** Un primitivo `int` (32 bit). Questo numero dice "di quanti posti dobbiamo spostare la virgola verso sinistra".

Esempio pratico: Se creiamo `new BigDecimal("123.456")`, in memoria avremo:

- `intVal` (Unscaled): **123456**
- `scale`: **3**

Il calcolo è: $123456 \times 10^{-3} = 123.456$.

Quanto può essere grande?

Mentre `double` esplode (diventa `Infinity`) se supera circa 1.8×10^{308}, `BigDecimal` è mostruosamente più capiente.

- **Limite delle Cifre:** Il "numero di cifre" è gestito dal `BigInteger`. Teoricamente può contenere miliardi di cifre, finché hai memoria RAM libera.
- **Limite della Virgola (Scale):** La posizione della virgola è gestita da un `int`, quindi la scala può andare da circa **-2 miliardi** a **+2 miliardi** ($\pm 2^{31} - 1$).

Questo significa che puoi rappresentare un numero con 2 miliardi di cifre dopo la virgola, o un numero intero così grande da avere 2 miliardi di zeri finali. Per contestualizzare: il numero di atomi nell'universo è stimato a 10^{80}. Un `BigDecimal` può gestire numeri enormemente più grandi.

```
BigDecimal numero = new BigDecimal("98.765");

// Il numero "puro" senza virgola
BigInteger unscaled = numero.unscaledValue();
// Risultato: 98765

// Dove mettere la virgola
int scala = numero.scale();
```

```
// Risultato: 3 (sposta la virgola di 3 a sinistra su 98765)
```

Listing 7.4: Ispezione della struttura interna

Colloquio: Scale Negativa?

Domanda: "Cosa succede se la scale è negativa in un BigDecimal?"

Risposta: "Se la scale è positiva indica i decimali. Se è **negativa**, indica che il numero deve essere moltiplicato per potenze di 10 (aggiunge zeri alla fine). Ad esempio, un Unscaled Value di `5` con scale `-3` vale $5 \times 10^{-(-3)} = 5 \times 10^3 = 5000$. Questo permette di gestire numeri interi enormi risparmiando spazio."

7.4 Operazioni con Scale Diverse

Cosa succede se sommiamo un numero con 1 decimale (`10.5`) a uno con 2 decimali (`0.25`)? E nella moltiplicazione? Le regole cambiano in base all'operazione matematica.

7.4.1 Somma e Sottrazione: La regola del Max Scale

Nella somma e nella sottrazione, la regola d'oro è l'allineamento. Java deve rendere i due numeri comparabili portandoli alla stessa scala. La scala del risultato sarà sempre il **massimo** tra le scale degli operandi:

$$\text{Scale}_{\text{risultato}} = \max(\text{Scale}_1, \text{Scale}_2)$$

Deep Dive: Dietro le quinte: L'Allineamento degli Unscaled Value

Immagina di sommare:

- **A:** `10.5` (Unscaled: 105, Scale: 1)
- **B:** `0.25` (Unscaled: 25, Scale: 2)

Java non può fare $105 + 25$. Prima deve "gonfiare" il numero con la scala più piccola per pareggiare quella più grande:

1. Identifica la scala target: **2** (la maggiore).
2. Prende **A**. La differenza di scala è $2 - 1 = 1$.
3. Moltiplica l'unscaled value di A per 10^1: $105 \times 10 = 1050$.
4. Ora A è virtualmente `10.50` (Unscaled: 1050, Scale: 2).
5. Esegue la somma degli interi: $1050 + 25 = 1075$.

Il risultato è un nuovo BigDecimal con unscaled **1075** e scale **2** $\rightarrow$ `10.75`.

7.4.2 Moltiplicazione: La somma delle Scale

Nella moltiplicazione non serve allineare. La regola è puramente matematica: il numero di cifre decimali del risultato è la somma delle cifre decimali dei fattori.

$$\text{Scale}_{\text{risultato}} = \text{Scale}_1 + \text{Scale}_2$$

```
BigDecimal a = new BigDecimal("1.2");  // Scale 1
BigDecimal b = new BigDecimal("3.45"); // Scale 2

// Scale risultante: 1 + 2 = 3
BigDecimal res = a.multiply(b);
System.out.println(res); // 4.140
// Nota: 12 * 345 = 4140. La virgola si sposta di 3.
```

7.4.3 Divisione: Il terreno minato

La divisione è l'operazione più pericolosa con i `BigDecimal`.

Se il risultato della divisione ha un numero finito di decimali (es. $1/2 = 0.5$), Java calcola la scala necessaria (spesso `scale1 - scale2`, ma si adatta per contenere il risultato). Tuttavia, se il risultato è un numero periodico o infinito (es. $1/3 = 0.333...$), `BigDecimal` cerca di calcolare la rappresentazione **esatta**. Non potendolo fare con una memoria finita, lancia un'eccezione.

Colloquio: ArithmeticException: Non-terminating decimal expansion

Problema: Molti junior dev scrivono questo codice che funziona nei test (es. $10/2$) ma crasha in produzione (es. $10/3$).

```
BigDecimal a = new BigDecimal("1");
BigDecimal b = new BigDecimal("3");

// CRASH! ArithmeticException
BigDecimal c = a.divide(b);
```

Soluzione: Quando si divide, è **obbligatorio** specificare due cose:

1. La `Scale` desiderata (quanti decimali voglio tenere).
2. Il `RoundingMode` (come arrotondare l'ultima cifra).

```
// Corretto: Voglio 2 decimali, arrotondando per eccesso se serve
BigDecimal c = a.divide(b, 2, RoundingMode.HALF_UP);
System.out.println(c); // 0.33
```

7.4.4 Wrapper Classes e Autoboxing

Per ogni primitivo esiste una classe corrispondente (Wrapper) che permette di trattarlo come un oggetto (es. `int` → `Integer`). Questo è necessario per usare i Generics (es. `List<Integer>`, non esiste `List<int>`).

Deep Dive: Autoboxing - Unboxing - Performance

Dal Java 5, la conversione è automatica:

```
Integer x = 10; // Autoboxing (da int a Integer)
int y = x;      // Unboxing (da Integer a int)
```

Attenzione ai rischi:

1. **NullPointerException:** Se `x` fosse `null`, la riga `int y = x;` lancerebbe una NPE, perché la JVM prova a invocare `x.intValue()` su un riferimento nullo.
2. **Performance:** L'autoboxing crea nuovi oggetti nello Heap. In cicli intensivi, usare `Integer` invece di `int` può causare un overhead inutile e pressione sul Garbage Collector.

Colloquio: Il trabocchetto dell'Integer Cache

Cosa stampa questo codice?

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

```

```
Integer c = 200; Integer d = 200;
System.out.println(c == d); // false (???)
```

Risposta: Java mantiene una **Cache** per gli oggetti `Integer` (e altri wrapper) tra **-128 e 127**.

- Per 100, Java riutilizza lo stesso oggetto dalla cache → riferimenti uguali.
- Per 200, Java crea due nuovi oggetti nello Heap → riferimenti diversi.

Morale: Confrontare sempre i Wrapper Objects con `.equals()`, mai con `==`.

7.5 Pass-by-Value vs Pass-by-Reference: Sfatiamo il mito

Questa è probabilmente la domanda teorica più sbagliata dai candidati Junior/Mid.

Deep Dive: Java è rigorosamente Pass-by-Value

In Java, il passaggio dei parametri avviene **sempre per valore (copia)**. Non esiste il "Pass-by-Reference" come in C++.

Cosa viene copiato?

- **Primitivi:** Viene copiato il valore effettivo (i bit). Modificare il parametro nel metodo non influenza la variabile originale.
- **Oggetti (Reference):** Viene copiato **il valore del riferimento** (l'indirizzo di memoria).

Questo significa che:

1. Puoi modificare lo **stato** dell'oggetto puntato (es. `p.setName("Mario")`), e la modifica sarà visibile fuori.
2. **NON** puoi cambiare l'oggetto a cui punta la variabile originale riassegnando il riferimento.

Esempio di prova:

```
public void swap(Object a, Object b) {
    Object temp = a;
    a = b;
    b = temp;
}
// Questo metodo NON scambia gli oggetti nel main.
// Ha solo scambiato le copie locali dei riferimenti 'a' e 'b'.
```

7.6 Operatori e Precedenza

Oltre agli operatori aritmetici base, è fondamentale conoscere le sfumature logiche e di concatenazione.

7.6.1 Short-Circuit Evaluation (&& vs &)

- `&&` (AND logico): Se il primo operando è `false`, non valuta nemmeno il secondo. (Più efficiente e sicuro contro le NPE).
- `&` (AND bitwise/booleano): Valuta **sempre** entrambi gli operandi.

Esempio critico: `if (obj != null && obj.isValid())` è sicuro. Usando `&`, lancerebbe NPE se `obj` è null.

7.6.2 Concatenazione di Stringhe

L'operatore + è sovraccaricato per le stringhe. La valutazione avviene da sinistra a destra.

```
System.out.println(1 + 2 + "Java"); // Stampa "3Java" (1+2=3, poi concatena)
System.out.println("Java" + 1 + 2); // Stampa "Java12" (String+1="Java1", poi "Java1
    ↪ "+2)
```

7.7 Control Flow: Best Practices e Modern Java

7.7.1 Switch vs If-Else

- **Switch Classico:** Funziona con byte, short, char, int, String ed Enum. Richiede `break` per evitare il fall-through.
- **If-Else:** Necessario per condizioni complesse o range di valori.

7.7.2 Java 14+: Switch Expressions

Java moderno ha introdotto una sintassi più pulita che può ritornare valori e non soffre di fall-through accidentali.

```
// Sintassi "Arrow" ->
String tipoGiorno = switch (giorno) {
    case LUN, MAR, MER, GIO, VEN -> "Lavorativo";
    case SAB, DOM -> "Weekend";
    default -> throw new IllegalArgumentException("Giorno non valido");
};
```

Questa feature mostra al colloquio che sei aggiornato sulle ultime versioni.

7.7.3 Cicli e Recursion

- **For-Loop:** Usalo quando sai a priori il numero di iterazioni o ti serve l'indice.
- **For-Each:** (`for (Tipo t : collezione)`) Usalo sempre per leggibilità su Array e Liste, a meno che tu non debba modificare la struttura (rimuovere elementi) mentre iteri.
- **Recursion Risk:** Ogni chiamata ricorsiva aggiunge un frame allo **Stack**. Troppe chiamate causano `StackOverflowError`. In Java, l'iterazione è quasi sempre preferibile alla ricorsione per problemi profondi.

7.8 L'Operatore Ternario

L'operatore ternario (`? :`) è una scorciatoia sintattica per l'istruzione `if-else`. È l'unico operatore in Java che accetta tre operandi.

La sua sintassi generale è:

`condizione ? espressione_se_vera : espressione_se_falsa`

A differenza di un blocco `if`, l'operatore ternario è un'**espressione**, il che significa che **restituisce sempre un valore**. Pertanto, non può essere utilizzato come un'istruzione a sé stante (void), ma il suo risultato deve essere assegnato a una variabile, restituito da un metodo o passato come argomento.

7.8.1 Esempio Pratico

Vediamo come semplificare il codice sostituendo un classico `if-else` con l'operatore ternario.

```java
public class TernaryExample {
    public static void main(String[] args) {
        int score = 75;
        String result;

        // Approccio Classico (Verboso)
        if (score >= 60) {
            result = "Promosso";
        } else {
            result = "Bocciato";
        }

        // Approccio con Operatore Ternario (Conciso)
        // Leggilo come: "Se score >= 60 allora 'Promosso', altrimenti 'Bocciato'"
        String quickResult = (score >= 60) ? "Promosso" : "Bocciato";

        System.out.println(quickResult);
    }
}
```

Listing 7.5: Confronto If-Else vs Operatore Ternario

Colloquio: Il tranello della "Numeric Promotion"

Una domanda comune nei test scritti riguarda il comportamento dell'operatore ternario quando i due valori di ritorno hanno **tipi numerici diversi**.

Cosa stampa il seguente codice?

```java
Object o1 = true ? new Integer(1) : new Double(2.0);
System.out.println(o1);
```

Risposta: Stampa `1.0`, non `1`.

Spiegazione: Java applica la *Binary Numeric Promotion*. Se uno degli operandi è `double` (o `Double`) e l'altro è convertibile in un numero (come `Integer`), Java converte entrambi al tipo più ampio (in questo caso `double`) prima di restituire il valore. Quindi l'`Integer(1)` viene promosso a `1.0`.

Deep Dive: Quando NON usare l'operatore ternario

Sebbene sia molto utile per assegnazioni semplici, l'abuso dell'operatore ternario può rendere il codice illeggibile.

Evita il "Nesting" (annidamento):

```java
// CATTIVA PRATICA: Difficile da leggere
String status = (age > 18) ? (hasLicense ? "Driver" : "Passenger") : "Child";

// BUONA PRATICA: Usa if-else o variabili intermedie
if (age <= 18) {
    status = "Child";
} else {
    status = hasLicense ? "Driver" : "Passenger";
}
```

La regola d'oro è: se devi fermarti a riflettere per capire l'ordine di valutazione, probabilmente dovresti usare un `if-else` standard.

7.9 Modificatori: Visibilità e Stato

In Java, la definizione di una variabile (o metodo/classe) è preceduta da parole chiave che ne determinano la visibilità e il ciclo di vita.

7.9.1 Modificatori di Accesso (Visibilità)

Controllano *chi* può vedere e usare i tuoi campi e metodi. Questo è il cuore dell'**Incapsulamento**.

Modificatore	Classe	Package	Sottoclasse	Mondo
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	**X**
`default` (nessuno)	✓	✓	**X**	**X**
`private`	✓	**X**	**X**	**X**

Tabella 7.2: Livelli di visibilità in Java

Colloquio: Differenza tra Default e Protected

Questa domanda serve a verificare se conosci la gestione dei package.

- **Default (Package-Private):** Se non scrivi nulla, il membro è visibile solo alle classi dello **stesso package**.
- **Protected:** Come il default, MA è visibile anche alle **sottoclassi** che si trovano in **package diversi**. È pensato per l'ereditarietà.

7.9.2 Modificatori di Stato (Non-Access Modifiers)

Questi modificatori cambiano il comportamento della variabile in memoria.

1. static

Rende la variabile appartenente alla **Classe** e non all'istanza (oggetto).

- C'è una sola copia della variabile in memoria, condivisa da tutte le istanze.
- Se un oggetto modifica una variabile statica, la modifica è visibile a tutti gli altri oggetti.
- **Uso comune:** Costanti, contatori globali, Singleton.

2. final

Indica che il valore non può essere cambiato una volta inizializzato.

- **Su primitive:** Il valore è costante (es. `final int MAX = 10;`).
- **Su oggetti:** Il **riferimento** è costante (non puoi riassegnare l'oggetto), ma lo **stato interno** dell'oggetto può cambiare (a meno che la classe stessa non sia immutabile).

3. volatile (Concetto Avanzato Back-End)

Utilizzato nella programmazione concorrente (Thread). Indica alla JVM che il valore della variabile deve essere letto/scritto sempre direttamente dalla **Main Memory** (RAM) e non dalla cache della CPU. Garantisce la visibilità delle modifiche tra thread diversi, ma non garantisce l'atomicità.

Capitolo 8

Classi, Oggetti e Membri

Java è un linguaggio puramente orientato agli oggetti. Comprendere l'anatomia di una classe non significa solo sapere scrivere `class Pippo`, ma capire come la JVM costruisce gli oggetti in memoria e in che ordine esegue il codice.

8.1 Anatomia di una Classe: Campi, Metodi, Costruttori

8.1.1 Ordine di Inizializzazione (The Initialization Block Trap)

Una delle domande più frequenti nei test di certificazione (OCP) e nei colloqui riguarda l'ordine esatto in cui vengono eseguiti i blocchi di codice quando si istanzia un oggetto.

L'ordine rigido è:

1. **Static Blocks & Fields:** Eseguiti **una sola volta** quando la classe viene caricata dal ClassLoader (prima ancora di creare istanze). Ordine di apparizione nel file.
2. **Parent Constructor:** Se la classe estende qualcos'altro, viene prima completata la costruzione del genitore.
3. **Instance Initializer Blocks & Fields:** Blocchi di codice tra graffe `{ ... }` fuori dai metodi e inizializzazione variabili di istanza. Eseguiti **ogni volta** che si crea un oggetto, in ordine di apparizione.
4. **Constructor:** Il corpo del costruttore è l'**ultimo** ad essere eseguito.

Colloquio: Cosa stampa questo codice?

```java
class Test {
    static { System.out.println("Static Block"); }

    { System.out.println("Instance Block"); }

    public Test() { System.out.println("Constructor"); }

    public static void main(String[] args) {
        new Test();
        new Test();
    }
}
```

Risposta:

```
Static Block     (Solo una volta al caricamento classe)
Instance Block   (Per il primo oggetto)
```

```
Constructor      (Per il primo oggetto)
Instance Block   (Per il secondo oggetto)
Constructor      (Per il secondo oggetto)
```

8.2 Specificatori di Accesso (Access Modifiers)

L'incapsulamento inizia qui. Java controlla la visibilità di classi, metodi e variabili attraverso quattro livelli di accesso. Saper scegliere quello giusto è il primo passo per un design sicuro.

Colloquio: Quali sono i modificatori di accesso in Java e le loro differenze?

Spesso i candidati dimenticano il livello "Default". Ecco la gerarchia dal più restrittivo al più aperto:

1. **private:** Visibile **solo** all'interno della stessa classe. È lo standard per i campi (variabili di istanza).
2. **Default (Package-Private):** Si applica quando **non** scrivi nessun modificatore. Visibile solo alle classi dello **stesso package**.
3. **protected:** Visibile nello stesso package **E** nelle **sottoclassi** (anche se si trovano in package diversi).
4. **public:** Visibile ovunque.

8.2.1 La Matrice di Visibilità

Memorizzare questa tabella è essenziale.

Modificatore	Stessa Classe	Stesso Pkg	Sottoclasse (Diff. Pkg)	Mondo
`public`	SÌ	SÌ	SÌ	SÌ
`protected`	SÌ	SÌ	SÌ	NO
`default` (no mod.)	SÌ	SÌ	NO	NO
`private`	SÌ	NO	NO	NO

Tabella 8.1: Matrice dei permessi di accesso

Deep Dive: Il trabocchetto di 'protected'

Molti pensano che `protected` significhi "solo sottoclassi". **Falso.** `protected` include **anche** l'accesso *package-private*. Se la classe `A` ha un metodo `protected`, e la classe `B` è nello stesso package (ma non estende A), `B` può comunque chiamare quel metodo.

8.2.2 Best Practice: Principle of Least Privilege

La regola d'oro per i colloqui e il codice pulito: **"Rendi tutto il più privato possibile"**.

- Inizia sempre con `private`.
- Se serve accesso ai test unitari nello stesso package, rimuovi il modificatore (usa `default`).
- Usa `protected` solo se stai progettando una libreria pensata per essere estesa.
- Usa `public` solo per i metodi dell'API (il "contratto" della classe).

8.3 La keyword `static`: Implicazioni e Memory Leak

Il modificatore `static` disaccoppia un membro dall'istanza dell'oggetto. Appartiene alla classe (o meglio, al Class Object nel Metaspace/Heap).

8.3.1 Static Methods

- Non possono accedere a variabili di istanza (non-static) perché non esiste il riferimento `this`.
- Sono risolti a **Compile-Time** (Static Binding).

Deep Dive: Static e il Method Hiding

In Java, i metodi statici **NON** supportano l'Overriding (polimorfismo a runtime). Se una sottoclasse ridefinisce un metodo statico con la stessa firma del padre, si parla di **Method Hiding**.

Quale metodo viene eseguito dipende dal **tipo della variabile di riferimento**, non dall'oggetto reale.

Di seguito un esempio pratico che dimostra come il compilatore risolva la chiamata basandosi sul tipo del riferimento (`Parent`) e non sull'istanza creata (`Child`). Quella che segue è una **BAD PRACTICE** ove si richiama un metodo statico tramite l'istanza di un oggetto. In questi casi è una scorciatoia che si attua implicando che il metodo statico sia legato alla classe del tipo della variabile che lo ha chiamato. Questo spiega nell'esempio il paradosso del Method Hiding.

```
class Parent {
    // Metodo statico nella superclasse
    public static void print() {
        System.out.println("Statico in Parent");
    }
}

class Child extends Parent {
    // HIDING: Questo metodo nasconde quello del padre, non lo sovrascrive
    public static void print() {
        System.out.println("Statico in Child");
    }
}

public class Main {
    public static void main(String[] args) {
        // Riferimento di tipo Parent, Oggetto di tipo Child
        Parent p = new Child();

        // Poiche' il metodo e' statico, Java guarda il TIPO DEL RIFERIMENTO (Parent)
        // Il polimorfismo non si applica qui.
        p.print();
    }
}
```

Listing 8.1: Dimostrazione del Method Hiding

Output del programma:

```
Statico in Parent
```

Colloquio: Perché il metodo statico non va in Override?

Durante un colloquio potresti dover spiegare il "perché".
Risposta: L'Overriding si basa sul *dynamic binding* (risoluzione a runtime dell'oggetto reale). I metodi statici, invece, sono risolti tramite *static binding* (al momento della compilazione) e sono legati alla Classe, non all'istanza. Per questo motivo il compilatore guarda solo il tipo della variabile dichiarata.

8.3.2 Static Variables e Memory Leaks

Le variabili statiche sono, di fatto, variabili globali. **Il Rischio:** Poiché le variabili statiche sono legate alla classe (che solitamente non viene mai scaricata dal GC finché l'app gira), gli oggetti referenziati da variabili statiche rimangono in memoria **per sempre**.
Scenario Classico: Una `static List<Object> cache` che continua a crescere senza mai essere svuotata causerà un **OutOfMemoryError**. Le variabili statiche sono considerate **GC Roots**.

8.4 Il riferimento `this` e `super`

8.4.1 Constructor Chaining

Una pratica essenziale per evitare duplicazione di codice nei costruttori e rispettare il principio **DRY** (*Don't Repeat Yourself*). Invece di riscrivere la logica di inizializzazione in ogni costruttore, si crea una "catena" di chiamate.

- `this(...)`: Chiama un altro costruttore della **stessa** classe.
- `super(...)`: Chiama un costruttore della **superclasse** (classe padre). Se non esplicito con extends si riferisce ad Object. Non esplicitare un metodo costruttore implicitamente richiama il costruttore della classe genitore.

```java
public class ServerConfig {
    private String host;
    private int port;
    private int timeout;

    // 1. Costruttore "Master": contiene l'unica logica di assegnazione
    public ServerConfig(String host, int port, int timeout) {
        this.host = host;
        this.port = port;
        this.timeout = timeout;
        System.out.println("Configurazione creata.");
    }

    // 2. Costruttore parziale: delega al Master fornendo un timeout di default
    public ServerConfig(String host, int port) {
        this(host, port, 3000); // Chiama il costruttore sopra
    }

    // 3. Costruttore vuoto: delega fornendo valori di default per tutto
    public ServerConfig() {
        this("localhost", 8080); // Chiama il costruttore parziale
    }
}
```

Listing 8.2: Esempio di Constructor Chaining con this()

Nell'esempio sopra, se instanziamo `new ServerConfig()`, Java eseguirà la catena:

1. `ServerConfig()` chiama...
2. `ServerConfig(String, int)` che chiama...
3. `ServerConfig(String, int, int)` (dove avviene l'inizializzazione reale).

La Regola d'Oro

La gestione della prima istruzione è rigida in Java. Ecco come affrontarla in sede di colloquio.

Colloquio: Posso usare this() e super() nello stesso costruttore?

No, è impossibile.

La chiamata a `this(...)` o `super(...)` deve essere tassativamente la **prima istruzione** del costruttore.

- **Motivo tecnico:** Java deve garantire che l'oggetto sia inizializzato correttamente (partendo dalla classe padre Object in giù) prima che si possa accedere a qualsiasi variabile d'istanza.
- Se provassi a scriverli entrambi, il secondo non sarebbe la prima istruzione, generando un errore di compilazione: *"Constructor call must be the first statement in a constructor"*.

Nota: Se non scrivi nulla, il compilatore inserisce implicitamente `super()` (senza argomenti) come prima riga.

Deep Dive: Chaining e Eccezioni

Se i costruttori nella catena lanciano **Checked Exceptions**, il costruttore chiamante deve dichiarare di lanciare le stesse eccezioni (o eccezioni padre). Non è possibile inserire un blocco `try-catch` attorno a una chiamata `super()` o `this()` perché, dovendo essere la prima istruzione, non può essere preceduta dalla keyword `try`.

8.5 Tassonomia delle Classi

In Java, una classe non è solo un file `.java`. Possiamo classificare le classi in base ai modificatori (cosa permettono di fare) e al loro ambito di definizione (dove si trovano).

8.5.1 Classi Top-Level e Modificatori

Le classi standard definite direttamente in un file package-level. Oltre alla dichiarazione base, possono avere modificatori che ne alterano drasticamente l'uso:

- **Abstract Class:** Una classe che non può essere istanziata direttamente (`new` non è permesso), ma serve solo come base per sottoclassi. Può contenere metodi astratti (senza corpo).
- **Final Class:** Una classe che **non può essere estesa**. Utile per garantire l'immutabilità (es. `String`) e la sicurezza, impedendo a chiunque di alterarne il comportamento tramite override.
- **Sealed Class (Java 17+):** Una via di mezzo. Permette l'ereditarietà, ma solo a un set specifico di classi permesse (`permits`).

8.5.2 Nested Classes (Classi Annidate)

Java permette di definire classi dentro altre classi. La distinzione cruciale è la parola chiave `static`.

A. Static Nested Class

Dichiarata come `static class Nested { ... }` all'interno di una classe esterna.

- **Comportamento:** È a tutti gli effetti una classe Top-Level, ma "impacchettata" dentro un'altra per coesione logica.
- **Accesso: Non ha accesso** ai membri d'istanza (non-statici) della classe esterna.
- **Istanziazione:** Non richiede un'istanza dell'outer class.

```
Outer.Nested n = new Outer.Nested(); // Sintassi pulita
```

B. Inner Class (Non-static / Member Class)

Dichiarata senza `static` all'interno di una classe.

- **Legame:** È legata indissolubilmente a una specifica **istanza** della classe esterna.
- **Accesso:** Vede **tutto** della classe esterna, inclusi i campi `private`.
- **Istanziazione:** Sintassi "aliena" che evidenzia la dipendenza dall'oggetto padre.

```
Outer o = new Outer();
Outer.Inner i = o.new Inner(); // Serve 'o' per creare 'i'
```

Deep Dive: Deep Dive: Il Memory Leak silenzioso

Quando compili una Inner Class, il compilatore genera un costruttore nascosto che accetta un riferimento alla classe esterna (spesso chiamato `this$0`).

Il Pericolo: Se passi l'oggetto `Inner` a un componente che vive a lungo (es. una cache o un listener statico), l'intero oggetto `Outer` rimarrà "vivo" in memoria, perché l'Inner ne detiene un riferimento forte. Il Garbage Collector non potrà pulire l'Outer finché l'Inner esiste.

Regola d'oro: Se la classe interna non ha bisogno di accedere ai campi dell'esterna, dichiarala **sempre** `static`.

8.5.3 Local Class (Classi Locali)

Classi definite **all'interno di un metodo**.

- Visibili solo all'interno di quel metodo.
- Possono accedere alle variabili locali del metodo solo se sono **final** o **effectively final** (non vengono mai modificate dopo l'inizializzazione).

Colloquio: Perché le variabili locali devono essere Final?

Domanda: Perché una Local/Anonymous Class può usare una variabile del metodo che la contiene solo se è `final`?

Risposta: Per una questione di *Lifecycle*. Il metodo viene eseguito sullo **Stack** e le sue variabili muoiono quando il metodo termina. L'istanza della classe locale viene creata nell'**Heap** e può sopravvivere molto più a lungo. Java effettua una "cattura" (copia) del valore della variabile dentro l'oggetto nell'Heap. Se la variabile sullo Stack cambiasse valore dopo la copia, ci sarebbe disallineamento. Imponendo `final`, Java garantisce la

coerenza.

8.5.4 Anonymous Inner Class

Definisce e istanzia una classe "al volo" senza darle un nome. Usata storicamente per i Callback.

```
// Vecchio stile (Java 7)
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonymous!");
    }
};
```

Sebbene le **Lambda Expressions** (Java 8) abbiano sostituito questo pattern per le interfacce funzionali (metodo singolo), le Anonymous Class servono ancora quando:

1. Devi fare override di **più metodi**.
2. Devi aggiungere **campi di stato** (variabili) all'istanza.
3. La classe da estendere non è un'interfaccia ma una classe astratta o concreta senza costruttori adatti alle lambda.

8.6 Enum: non sono semplici costanti

In alcuni linguaggi (come C) un enum è solo un numero mascherato. In Java, invece, un `enum` è una vera e propria **classe speciale**.

La differenza fondamentale è questa:

> Una classe normale può avere infinite istanze create con `new`. Un enum, invece, ha un numero **fisso e limitato di istanze**, deciso a priori nel codice.

Queste istanze:

- vengono create automaticamente dalla JVM
- hanno un nome preciso (es: `CREATO`, `PAGATO`, `SPEDITO`)
- non possono essere duplicate o create manualmente

8.6.1 Caratteristiche principali degli Enum

- Non possono essere estesi (sono implicitamente `final`)
- Possono avere **campi**, **costruttori** e **metodi**
- Rendono il codice più sicuro (*Type Safety*): non puoi usare valori che non esistono nell'enum

```
public enum StatoOrdine {

    CREATO(1),
    PAGATO(2),
    SPEDITO(3),
    CONSEGNATO(4);

    // Dato interno dell'enum
    private final int codice;

    // Costruttore (chiamato dalla JVM per ogni valore dell'enum)
    StatoOrdine(int codice) {
```

```
        this.codice = codice;
    }

    // Metodo: comportamento legato allo stato
    public boolean isCompletato() {
        return this == CONSEGNATO;
    }

    public int getCodice() {
        return codice;
    }
}

// Utilizzo
StatoOrdine s = StatoOrdine.PAGATO;

if (!s.isCompletato()) {
    System.out.println("Ordine in corso, codice: " + s.getCodice());
}
```

Listing 8.3: Enum con stato (dati) e comportamento (metodi)

Deep Dive: Singleton Pattern con Enum

Joshua Bloch (autore di *Effective Java*) suggerisce di usare un enum quando si vuole creare un vero **Singleton**.

Se un enum ha **un solo valore**, allora esisterà **una sola istanza** di quella classe in tutta l'applicazione.

```
public enum DatabaseConnection {
    INSTANCE;

    public void connect() {
        // Logica di connessione
    }
}

// Uso
DatabaseConnection.INSTANCE.connect();
```

Questa è la versione più sicura possibile del Singleton perché:

1. **Thread-safe:** la JVM crea l'istanza una sola volta in modo sicuro, anche con più thread.
2. **Sicurezza con la serializzazione:** Java impedisce che vengano create copie durante la deserializzazione.
3. **Sicurezza contro reflection:** non puoi forzare la creazione di nuove istanze di un enum.

Colloquio: Metodi automatici degli Enum

La JVM aggiunge automaticamente due metodi statici utilissimi:

- `values()` : restituisce un array con tutti i valori dell'enum
- `valueOf(String name)` : converte una stringa in un valore dell'enum (lancia un errore se non esiste)

Capitolo 9

I Pilastri della OOP

L'Object Oriented Programming si regge su quattro pilastri. Conoscerli a memoria non basta; bisogna saper spiegare *come* Java li implementa e quali limitazioni impone.

9.1 I 4 Pilastri della Programmazione ad Oggetti (OOP)

La Programmazione Orientata agli Oggetti (OOP) non è solo una sintassi particolare; è un **paradigma di pensiero**. L'idea di base è modellare il software come se fosse il mondo reale: non più una lunga lista di istruzioni procedurali, ma un insieme di "Oggetti" che interagiscono tra loro scambiandosi messaggi.

Per definire un sistema OOP "puro", ci si basa su quattro concetti fondamentali, noti come i "Pilastri". Durante un colloquio, è essenziale saperli definire concettualmente, slegandosi dal codice.

9.1.1 1. Astrazione (Abstraction)

L'astrazione è l'arte di **semplificare la realtà**. Nel mondo reale, ogni oggetto è infinitamente complesso. Quando programmiamo, non ci serve modellare ogni singolo atomo di un'automobile. Ci interessa modellare solo ciò che serve al nostro sistema (es. *marca*, *modello*, *velocità*).

- **Concetto:** Nascondere la complessità dei dettagli implementativi e mostrare solo le funzionalità essenziali all'utente.
- **Analogy:** Quando guidi un'auto, interagisci con il volante e i pedali (l'interfaccia astratta). Non hai bisogno di sapere come funziona l'iniezione elettronica o la combustione interna (i dettagli complessi) per poter guidare.

9.1.2 2. Incapsulamento (Encapsulation)

L'incapsulamento è un meccanismo di **protezione e controllo**. Serve a raggruppare dati (variabili) e metodi (comportamenti) in un'unica unità (la Classe) e a limitarne l'accesso dall'esterno.

- **Concetto:** Impedire che i dati interni di un oggetto vengano modificati arbitrariamente da altri oggetti. L'oggetto diventa una "Black Box": protegge il proprio stato e permette modifiche solo attraverso metodi controllati.
- **Analogy:** Pensa a un conto bancario. Non puoi entrare nel database della banca e modificare il tuo saldo a mano. Devi usare uno sportello o un'app (metodi pubblici) che verificano se hai i fondi prima di prelevare. L'incapsulamento garantisce l'integrità del saldo.

9.1.3 3. Ereditarietà (Inheritance)

L'ereditarietà è un meccanismo di **riutilizzo e gerarchia**. Permette di creare nuove classi basandosi su classi esistenti, ereditandone caratteristiche e comportamenti senza dover riscrivere il codice.

- **Concetto:** Stabilire una relazione "Is-A" (È-Un) tra concetti generali e concetti specifici. Evita la duplicazione del codice (principio DRY - Don't Repeat Yourself).
- **Analogy:** In biologia, un "Cane" eredita le caratteristiche di un "Mammifero", che a sua volta eredita da "Animale". Il cane ha tutte le proprietà dell'animale (respira, mangia) più le sue specificità (abbaia).

9.1.4 4. Polimorfismo (Polymorphism)

Il termine deriva dal greco "molte forme". È la capacità di oggetti di tipo diverso di rispondere allo stesso messaggio (metodo) in modi differenti. È il pilastro che garantisce la **flessibilità** del sistema.

- **Concetto:** Un'unica interfaccia per molte implementazioni. Posso trattare un gruppo di oggetti diversi come se fossero lo stesso tipo generico, e lasciare che ognuno esegua l'azione a modo suo.
- **Analogy:** Prendi un telecomando universale col tasto "Play".
  - Se lo punti verso un lettore DVD, il tasto "Play" fa girare il disco.
  - Se lo punti verso un lettore MP3, il tasto "Play" avvia la musica digitale.
  - Se lo punti verso una console, il tasto "Play" avvia il gioco.

  L'azione (l'interfaccia "Play") è la stessa, ma il risultato (l'implementazione) cambia a seconda dell'oggetto che la riceve.

Colloquio: Come spiegheresti i 4 pilastri in 30 secondi?

Domanda: "Senza scrivere codice, riassumimi i vantaggi dell'OOP."

Risposta:

1. **Astrazione:** Riduce la complessità concentrandosi su "Cosa fa" l'oggetto, non su "Come lo fa".
2. **Incapsulamento:** Protegge i dati da modifiche non autorizzate, garantendo stabilità.
3. **Ereditarietà:** Elimina la ridondanza organizzando il codice in gerarchie logiche.
4. **Polimorfismo:** Rende il sistema estendibile, permettendo di aggiungere nuovi tipi di oggetti senza rompere il codice esistente che li utilizza.

9.2 Incapsulamento: Oltre i Getter e Setter

L'incapsulamento non serve solo a "nascondere i dati", ma a **proteggere gli invarianti** della classe. Un invariante è una condizione che deve essere sempre vera (es. `età >= 0`). Se il campo fosse `public`, chiunque potrebbe impostarlo a -1 rompendo la logica.

Deep Dive: Defensive Copying (La copia difensiva)

Un errore comune nei colloqui riguarda la gestione di oggetti mutabili nei getter/setter.

```
1 public class Periodo {
2     private Date fine;
3
```

```
    public Date getFine() {
        return fine; // ERRORE DI SICUREZZA!
    }
}
```

Poiché `Date` è mutabile, chi chiama `getFine()` riceve il riferimento all'oggetto originale e può cambiarne il valore (es. `p.getFine().setYear(1900)`), modificando lo stato interno di `Periodo` a sua insaputa. **Soluzione:** Restituire sempre una copia (clone) o usare classi immutabili (es. `LocalDate` di Java 8).

```
public Date getFine() {
    return new Date(fine.getTime()); // Copia difensiva
}
```

9.3 Ereditarietà: Single vs Multiple

Java supporta l'**Ereditarietà Singola** per le classi (`extends`) ma l'**Ereditarietà Multipla** per le interfacce (`implements`).

Colloquio: Cos'è il Diamond Problem e come lo gestisce Java?

Problema: Immagina una classe `C` che estende sia `A` che `B`. Entrambe (`A` e `B`) hanno un metodo `print()`. Se chiami `c.print()`, quale versione viene eseguita? Quella di A o di B? Questa ambiguità è il "Diamond Problem".

Soluzione in Java:

- **Classi:** Java proibisce l'ereditarietà multipla di classi (`extends A, B` è illegale). Il problema è risolto alla radice.
- **Interfacce (Java 8+):** Con i *Default Methods* (metodi delle interfacce che presentano un implementazione a patto di usare la key-word default), il problema è tornato. Se implementi due interfacce che hanno lo stesso metodo default, il compilatore genera un **errore di compilazione**.
- **Risoluzione:** Devi obbligatoriamente fare Override del metodo nella tua classe e specificare quale chiamare: `InterfaceA.super.print();`

9.4 Polimorfismo: Overloading vs Overriding

9.4.1 1. Overloading (Compile-time Polymorphism)

Stesso nome metodo, parametri diversi (firma diversa). Risolto dal compilatore (Static Binding). **Nota:** Cambiare solo il tipo di ritorno **non** è overloading valido.

9.4.2 2. Overriding (Runtime Polymorphism)

Stessa firma, implementazione diversa nella sottoclasse. Risolto dalla JVM a runtime (Dynamic Binding) basandosi sull'oggetto reale.

Deep Dive: Regole dell'Overriding (Covarianza ed Eccezioni)

Quando fai override di un metodo, devi rispettare il contratto del padre, ma con alcune libertà:

1. **Access Modifier:** Non puoi essere più restrittivo (da `public` a `private`), ma puoi essere più permissivo.
2. **Return Type (Covarianza):** Puoi restituire il tipo originale o una sua **sottoclasse**.
   - Padre: `public Number getValore()`
   - Figlio: `public Integer getValore()` → **Valido!**
3. **Exceptions:**
   - Puoi non lanciare eccezioni.
   - Puoi lanciare le stesse eccezioni (o sottoclassi).
   - **NON** puoi lanciare nuove *Checked Exceptions* non previste dal padre.
   - Puoi lanciare qualsiasi *Unchecked Exception*.

9.5 Astrazione: Classi Astratte vs Interfacce

L'astrazione è uno dei pilastri della OOP, ma la distinzione tra Classi Astratte e Interfacce va ben oltre la sintassi. La scelta tra le due determina la flessibilità futura della tua applicazione.

9.5.1 La differenza concettuale: "Chi sono" vs "Cosa so fare"

- **Classe Astratta (Is-A):** Definisce l'identità fondamentale dell'oggetto. Un `Cane` *è un* `Animale`. Le classi astratte sono ideali per definire un **template** di base con una logica parzialmente implementata e, soprattutto, uno **stato** condiviso.
- **Interfaccia (Can-Do / Behavior):** Definisce una capacità o un contratto. Un `Cane` *può essere* `Addestrabile`, ma anche un `Robot` può esserlo. Le interfacce disaccoppiano il comportamento dalla gerarchia delle classi.

Deep Dive: Perché le Classi Astratte esistono ancora dopo Java 8?

Con l'introduzione dei `default methods` nelle interfacce (Java 8), molti si chiedono se le classi astratte siano obsolete. La risposta è **NO**.

La differenza critica è lo **Stato (State)**.

- Le interfacce **non possono avere variabili d'istanza** (stato mutabile). Possono avere solo costanti (`static final`).
- Le classi astratte possono avere campi (`private`, `protected`) che mantengono lo stato dell'oggetto e costruttori per inizializzarlo.

Usa una classe astratta quando hai bisogno di condividere codice che manipola lo stato interno dell'oggetto.

9.5.2 Tabella Comparativa Aggiornata

9.5.3 Esempio Pratico: Il problema dello Stato

Per capire perché non possiamo usare sempre e solo le interfacce, proviamo a modellare un videogioco. Vogliamo che le nostre entità abbiano una **barra della vita (energia)** che diminuisce quando fanno azioni.

Classe Astratta	Interfaccia
Relazione: Gerarchica stretta (Is-A).	**Relazione:** Contratto trasversale (Can-Do).
Stato: Può avere variabili d'istanza (non statiche, non final).	**Stato:** Solo costanti (`public static final`). Niente stato interno.
Costruttori: Sì, invocati dalle sottoclassi via `super()`.	**Costruttori:** No.
Visibilità: I metodi possono essere `public`, `protected`, `private`.	**Visibilità:** Default `public`. Da Java 9 supporta `private` per uso interno.
Ereditarietà: Singola.	**Ereditarietà:** Multipla (una classe può implementare N interfacce).

- **Il problema:** Se definissimo `energia` in un'interfaccia, sarebbe implicitamente `public static final` (una costante). Non potremmo mai scrivere `energia -= 10`, perché le costanti non cambiano!
- **La soluzione:** Usiamo una **Classe Astratta** per contenere la variabile `energia` (lo Stato) e un'**Interfaccia** per definire abilità extra come `Volante` (il Comportamento).

```java
// 1. CLASSE ASTRATTA: Gestisce lo STATO (Chi sono e come sto)
public abstract class Personaggio {
    // Questo è lo "Stato Interno".
    // Un'interfaccia NON può avere questo campo modificabile.
    protected int energia = 100;

    // Metodo concreto che manipola lo stato
    public void riposa() {
        this.energia = 100;
        System.out.println("Energia ripristinata.");
    }
}

// 2. INTERFACCIA: Gestisce il COMPORTAMENTO (Cosa so fare)
public interface Volante {
    void vola();
}

// 3. CLASSE CONCRETA
public class Drago extends Personaggio implements Volante {

    @Override
    public void vola() {
        // Posso accedere e modificare lo stato ereditato
        if (this.energia >= 10) {
            System.out.println("Il drago sta volando in alto!");
            this.energia -= 10; // MODIFICA DELLO STATO (Possibile solo grazie alla
    ↪ classe astratta)
        } else {
            System.out.println("Troppo stanco per volare.");
        }
    }
}
```

Listing 9.1: Gestione dello Stato: Classe Astratta vs Interfaccia

Deep Dive: In sintesi

Se provassi a mettere `int energia = 100;` dentro l'interfaccia `Volante`, Java lo trasformerebbe automaticamente in una costante (`final`). Alla riga `this.energia -= 10` riceveresti un errore di compilazione: *"Cannot assign a value to final variable energy"*. Ecco perché per gestire dati che cambiano nel tempo (stato) serve una Classe (Astratta o no).

9.6 Evoluzione delle Interfacce (Java 8, 9, 17+)

Le interfacce, in origine, erano semplici contratti: dichiaravano **cosa** fare, ma non **come** farlo. Con Java 8, 9 e 17, le interfacce si sono evolute in strumenti molto più potenti, capaci di:

- far evolvere le API senza rompere il codice esistente (*Backward Compatibility*)
- incorporare comportamento di base
- controllare precisamente chi può implementarle

9.6.1 Default Methods (Java 8) e il Diamond Problem

Prima di Java 8, aggiungere un nuovo metodo a un'interfaccia significava dover modificare **tutte** le classi che la implementavano.

Con l'introduzione dei **default methods**, un'interfaccia può fornire una implementazione di base:

```
interface Veicolo {
    default void accendi() {
        System.out.println("Il veicolo si accende.");
    }
}
```

Listing 9.2: Esempio semplice di Default Method

Se una classe implementa questa interfaccia ma non ridefinisce il metodo, utilizza automaticamente quello fornito.

Il problema del diamante (Diamond Problem)

Il problema nasce quando più interfacce forniscono un *default method* con la stessa firma.

```
interface Veicolo {
    default void accendi() {
        System.out.println("Il veicolo si accende.");
    }
}

interface Allarme {
    default void accendi() {
        System.out.println("L'allarme si attiva.");
    }
}

// ERRORE: ambiguità sui default methods
```

```java
public class AutoBlindata implements Veicolo, Allarme {

    @Override
    public void accendi() {
        // Scelta esplicita: utilizzo il metodo dell'interfaccia Veicolo
        Veicolo.super.accendi();
        System.out.println("Sistemi di sicurezza attivi.");
    }
}
```

Listing 9.3: Conflitto tra default methods

La classe è obbligata a risolvere il conflitto definendo quale metodo utilizzare.

9.6.2 Regole di risoluzione dei Default Methods

Quando esiste un conflitto tra più metodi con la stessa firma, Java applica le seguenti regole, in ordine:

1. **Class wins:** Se la classe (o una superclasse) definisce il metodo, questa implementazione ha la priorità su qualsiasi default method.
2. **Sub-interface wins:** Se un'interfaccia estende un'altra e ridefinisce il metodo, vince sempre l'interfaccia più specifica (la sottinterfaccia).
3. **Ambiguità:** Se non esiste alcuna relazione tra le interfacce, la classe concreta deve fornire una propria implementazione ed esplicitare la scelta.

9.6.3 Metodi Statici e Privati (Java 8 & 9)

Metodi statici (Java 8) Un'interfaccia può contenere metodi `static`, utilizzabili senza creare un oggetto:

```java
interface Matematica {
    static int max(int a, int b) {
        return a > b ? a : b;
    }
}
```

Listing 9.4: Metodo statico in un'interfaccia

Utilizzo:

```java
int valore = Matematica.max(4, 10);
```

Caratteristiche principali:

- Non sono ereditati dalle classi
- Non sono sovrascrivibili (override)
- Servono come metodi di utilità o factory

Esempio reale: `Stream.of(...)` è un metodo `static` dell'interfaccia `Stream`.

Metodi private (Java 9) I metodi `private` all'interno di un'interfaccia servono per evitare duplicazione di codice tra più *default methods*.

```java
interface Logger {

    default void info(String msg) {
        stampa("INFO", msg);
    }
```

```
6
7       default void error(String msg) {
8           stampa("ERROR", msg);
9       }
10
11      private void stampa(String livello, String msg) {
12          System.out.println("[" + livello + "] " + msg);
13      }
14  }
```

Listing 9.5: Metodo private in un'interfaccia

Questi metodi:

- Sono visibili solo all'interno dell'interfaccia
- Non sono accessibili dalle classi implementanti
- Non sono ereditati

9.6.4 Sealed Interfaces (Java 17)

Con Java 17 è possibile dichiarare un'interfaccia come **sealed** (sigillata), limitando esplicitamente quali classi possono implementarla.

```
1  public sealed interface Forma permits Cerchio, Rettangolo {
2      double calcolaArea();
3  }
4
5  final class Cerchio implements Forma { ... }
6  final class Rettangolo implements Forma { ... }
7
8  // ERRORE: non è permesso
9  // class Triangolo implements Forma { ... }
```

Listing 9.6: Esempio di sealed interface

Questo meccanismo permette:

- maggiore controllo sulla gerarchia
- maggiore sicurezza del dominio applicativo
- supporto a pattern matching avanzato negli `switch`

9.6.5 Nota importante: uso di `extends` nelle interfacce

Quando un'interfaccia eredita da un'altra (o più) interfaccia, utilizza sempre la parola chiave **`extends`**, non `implements`.

```
1  interface A {
2      void metodoA();
3  }
4
5  interface B {
6      void metodoB();
7  }
8
9  // Le interfacce utilizzano EXTENDS
10 interface C extends A, B {
11     void metodoC();
12 }
```

Listing 9.7: Interfaccia che estende altre interfacce

Caratteristiche:
- Un'interfaccia può estendere una o più interfacce
- Questo è un tipo sicuro di ereditarietà multipla
- Le classi invece usano `implements`

Riassunto finale:
- Java 8 → *default* e *static methods*
- Java 9 → *private methods*
- Java 17 → *sealed interfaces*
- Le interfacce usano `extends`, le classi usano `implements`

Capitolo 10

Gestione dei Dati: Identità, Uguaglianza e Immutabilità

In Java, ogni classe estende implicitamente `java.lang.Object`. Questo significa che non esistono oggetti "nudi": ogni istanza porta con sé un'eredità di metodi fondamentali che definiscono come l'oggetto si comporta nelle collezioni, nella sincronizzazione e nel debug.

Ignorare il funzionamento di questi metodi è la causa principale di memory leak nelle cache (es. HashMap) e di bug logici difficili da tracciare.

10.1 La Classe java.lang.Object

Prima di approfondire i contratti specifici, ecco una panoramica di tutti i metodi esposti da `Object`. Durante i colloqui, una domanda frequente per testare la conoscenza della libreria standard è: *"A parte equals e toString, quali altri metodi della classe Object conosci?"*.

Metodo	Descrizione
`equals(Object obj)`	Determina l'uguaglianza logica tra due oggetti.
`hashCode()`	Restituisce un intero rappresentativo, fondamentale per le Hash Tables.
`toString()`	Restituisce la rappresentazione stringa dell'oggetto. Essenziale per il logging e il debug.
`getClass()`	Restituisce l'oggetto `Class<?>` a runtime (Reflection). È un metodo `final`, quindi non sovrascrivibile.
`clone()`	Crea e restituisce una copia dell'oggetto. Spesso sconsigliato in favore dei costruttori di copia.
`wait()`, `notify()`, `notifyAll()`	Primitive di basso livello per la sincronizzazione dei Thread.
`finalize()`	**DEPRECATO** (da Java 9). Non usarlo mai per rilasciare risorse; non c'è garanzia che venga chiamato dal Garbage Collector.

10.2 Identità vs Uguaglianza

Prima di scrivere codice, bisogna distinguere filosoficamente due concetti che spesso i principianti confondono:

1. **Identità (Reference Equality):** Due variabili puntano allo stesso indirizzo di memoria fisico? Sono lo stesso oggetto?
2. **Uguaglianza (Logical Equality):** Due oggetti, pur essendo distinti in memoria, contengono gli stessi dati significativi (es. stesso ID utente)?

Colloquio:

Domanda: "Qual è la differenza tra l'operatore `==` e il metodo `.equals()`?"

Risposta:

- `==` controlla l'**Identità**. Restituisce `true` solo se i due riferimenti (puntatori) puntano esattamente alla stessa locazione di memoria nello Heap.
- `.equals()` controlla l'**Uguaglianza Logica**.
  - L'implementazione di default in `Object` usa `==` (quindi controlla l'identità).
  - Nelle classi Java standard (es. `String`, `Integer`) e nelle tue classi di dominio (DTO, Entity), deve essere sovrascritto per confrontare il **contenuto** (i campi interni).

Deep Dive: String Pool e l'inganno del

Le stringhe in Java sono speciali.

- `String a = "ciao";`
- `String b = "ciao";`

In questo caso `a == b` potrebbe restituire `true` grazie allo *String Pool* (ottimizzazione della JVM che riusa le stringhe letterali). Tuttavia, se scrivi `String c = new String("ciao");`, allora `a == c` sarà **false**, mentre `a.equals(c)` sarà **true**.

Regola d'oro: Per confrontare oggetti (incluse le Stringhe), usa **sempre** `.equals()`. Usa `==` solo per i tipi primitivi o per controllare se un riferimento è `null`.

10.3 Il Contratto HashCode e le Collezioni

Se decidi di sovrascrivere `equals()`, firmi un contratto implicito ma vincolante: **DEVI sovrascrivere anche `hashCode()`**.

Le strutture dati più usate in Java (`HashMap`, `HashSet`, `Hashtable`) basano la loro efficienza su questo legame. Se lo rompi, le collezioni smettono di funzionare correttamente (perdono i dati), anche se il codice compila senza errori.

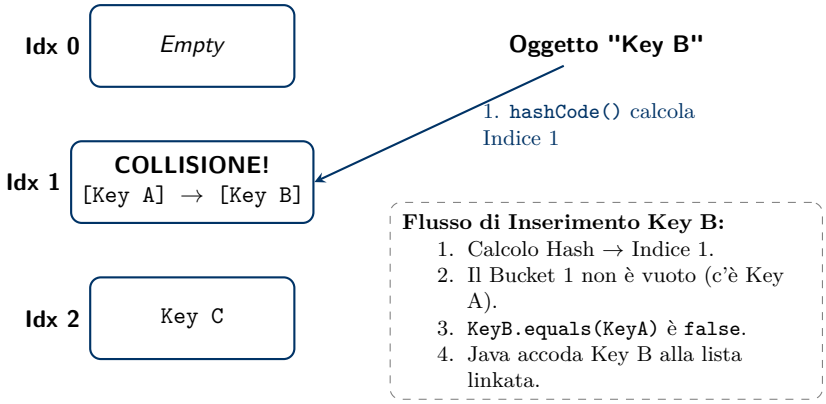
10.3.1 Come funziona una HashMap (Dietro le quinte)

Una HashMap non è magia: è essenzialmente un **Array** di liste (o alberi, da Java 8). Ogni posizione dell'array è chiamata **Bucket** (Secchio).

Il processo di inserimento/ricerca segue questi step:

1. **Hashing:** Viene invocato `key.hashCode()`. Il risultato viene trasformato in un indice dell'array (es. `hash % arrayLength`).
2. **Individuazione:** Si va al bucket corrispondente.
3. **Confronto:**
   - Se il bucket è vuoto: inserisce l'oggetto.

- Se c'è già qualcosa (**Collisione**): scorre la lista di oggetti presenti usando `equals()` per vedere se la chiave esiste già (aggiornamento) o se è nuova (aggiunta in coda).



Deep Dive: Le 3 Regole del Contratto HashCode

Secondo la specifica Java, un'implementazione corretta deve rispettare queste regole:

1. **Coerenza (Consistency):** Se chiamo `hashCode()` più volte sullo stesso oggetto (senza modificarne i campi), deve restituire sempre lo stesso numero.
2. **Uguaglianza implica Hash uguale:** Se `a.equals(b)` è `true`, allora `a.hashCode()` DEVE essere uguale a `b.hashCode()`. *(Questa è la regola più violata).*
3. **Hash uguale NON implica Uguaglianza:** Se `a.hashCode() == b.hashCode()`, non è detto che `equals` sia true. Questo fenomeno si chiama **Collisione** ed è gestito dalla HashMap, ma troppe collisioni degradano le performance da $O(1)$ a $O(n)$.

Colloquio: Il bug della HashMap silenziosa

Scenario: Hai una classe `Utente` con campo `id`. Fai l'override solo di `equals` (dicendo che due utenti sono uguali se hanno lo stesso ID).

- Inserisci: `map.put(new Utente(1), "Dati");`
- Cerchi: `map.get(new Utente(1));`

Domanda: Cosa restituisce la mappa?

Risposta: Restituisce `null`. **Perché?** Non avendo sovrascritto `hashCode`, i due oggetti (pur avendo ID 1) hanno hashcode nativi diversi (basati sull'indirizzo di memoria). La HashMap guarda nel "secchio" sbagliato e non trova l'oggetto, senza nemmeno provare a chiamare `equals`.

10.4 Immutabilità

Una classe si definisce **Immutabile** quando il suo stato non può essere modificato in alcun modo dopo che l'oggetto è stato costruito. Esempi classici nel JDK sono `String`, `Integer`, `BigDecimal`.

10.4.1 Perché è fondamentale?

Ci sono due motivi principali per cui i DTO (Data Transfer Objects) e le Value Classes dovrebbero essere immutabili:

1. **Thread Safety:** Un oggetto immutabile è automaticamente thread-safe. Può essere letto da mille thread contemporaneamente senza bisogno di sincronizzazione, perché nessuno può cambiarne lo stato.
2. **Chiavi delle HashMap:** Questa è la regola d'oro. **Le chiavi di una Mappa non devono mai cambiare.**

Colloquio: Il problema delle chiavi mutabili

Domanda: "Cosa succede se uso un oggetto mutabile come chiave di una HashMap e poi ne modifico un campo?"

Risposta: Perdi l'oggetto.

- Quando inserisci l'oggetto, viene calcolato l'hash (es. 10) e l'oggetto finisce nel Bucket 10.
- Se modifichi un campo dell'oggetto, il suo `hashCode()` cambia (es. diventa 55).
- Quando cerchi l'oggetto, la Map calcola il nuovo hash (55), guarda nel Bucket 55, ma l'oggetto è fisicamente rimasto nel Bucket 10!
- Risultato: `get()` restituisce `null`, anche se l'oggetto è dentro la mappa. Si crea un memory leak.

10.4.2 Come creare una Classe Immutabile (La Checklist)

Per rendere una classe immutabile, non basta togliere i setter. Bisogna seguire una ricetta precisa per garantire la "Deep Immutability".

1. Dichiarare la classe `final` (per impedire che una sottoclasse malevola sovrascriva i metodi e aggiunga stato mutabile).
2. Rendere tutti i campi `private` e `final` (devono essere assegnati nel costruttore).
3. Non fornire nessun metodo **Setter**.
4. **Defensive Copying:** Se la classe contiene campi che sono oggetti mutabili (come `Date` o `List`), non bisogna mai restituire o assegnare il riferimento diretto.

Deep Dive: Deep Dive: Defensive Copying

Molti candidati cadono su questo punto. Se la tua classe ha un campo `private final Date data;`, il campo è final (il riferimento non cambia), ma l'oggetto `Date` interno è mutabile (ha il metodo `setTime()`).

Se restituisci `return this.data;` nel getter, chi chiama il metodo può modificare la data interna del tuo oggetto "immutabile"!

Soluzione:

- **In Ingresso (Costruttore):** Crea una copia dei dati ricevuti.
- **In Uscita (Getter):** Restituisci una copia dei dati interni.

```
1  public final class ImmutableUser {
2      private final String username;     // String è immutabile, ok.
3      private final Date registration;  // Date è MUTABILE! Attenzione.
4
5      public ImmutableUser(String username, Date registration) {
6          this.username = username;
7          // COPY IN: Se il chiamante modifica la sua variabile 'date' dopo
8          // aver chiamato il costruttore, noi non ne risentiamo.
9          this.registration = new Date(registration.getTime());
10     }
11
```

```
12      public String getUsername() {
13          return username;
14      }
15
16      public Date getRegistration() {
17          // COPY OUT: Se il chiamante usa setTime() sulla data ricevuta,
18          // modifica la sua copia, non il nostro stato interno.
19          return new Date(this.registration.getTime());
20      }
21  }
```

Listing 10.1: Immutabilità Corretta con Defensive Copy

10.5 Java Records (Modern Java)

Fino a Java 14, creare una semplice classe immutabile per trasportare dati (DTO) richiedeva decine di righe di codice ripetitivo (*boilerplate*): campi privati, costruttore, getter, `equals`, `hashCode` e `toString`.

Java 14 (diventato standard in Java 16) ha introdotto i **Record**. Un Record è una classe speciale il cui scopo principale è essere un "*trasportatore trasparente di dati immutabili*".

10.5.1 Sintassi e Funzionalità

Con una sola riga di codice, il compilatore genera tutto ciò che serve.

```
1  // Sintassi concisa: definisce campi e costruttore in una riga
2  public record UserRecord(String username, int age) {}
```

Listing 10.2: Definizione di un Record

Dietro le quinte, il compilatore trasforma quella riga in una classe `final` con:

- Due campi `private final` (`username`, `age`).
- Un **Costruttore Canonico** che inizializza tutti i campi.
- Metodi di accesso (senza prefisso `get`, es. `.username()`, `.age()`).
- Implementazioni corrette e performanti di `equals()`, `hashCode()` e `toString()`.

Colloquio: Record vs Classi Tradizionali

Domanda: "Posso aggiungere metodi o logica a un Record?"

Risposta: Sì, ma con limitazioni strutturali per preservare l'immutabilità.

- Puoi aggiungere metodi statici o di istanza per logica di business leggera.
- Puoi implementare interfacce.
- **NON** puoi estendere altre classi (il Record estende già implicitamente `java.lang.Record`).
- **NON** puoi aggiungere campi di istanza extra che non siano elencati nella dichiarazione del record.

10.5.2 Validazione: Il Compact Constructor

Un dubbio comune è: *"Se il costruttore è automatico, come faccio a validare che l'username non sia null?"*. I Record supportano una sintassi speciale chiamata **Compact Constructor**.

Non devi ripetere i parametri e le assegnazioni (`this.x = x`). Scrivi solo la logica di controllo.

```java
public record UserRecord(String username, int age) {

    // Niente parentesi con argomenti qui!
    public UserRecord {
        Objects.requireNonNull(username, "L'username è obbligatorio");

        if (age < 0) {
            throw new IllegalArgumentException("L'età non può essere negativa");
        }

        // Non serve scrivere this.age = age; lo fa Java in automatico alla fine.
    }
}
```

Listing 10.3: Compact Constructor per la validazione

Deep Dive: Record vs Lombok

Molti progetti usano la libreria **Lombok** (`@Data`, `@Value`) per ridurre il codice. Tuttavia, i Record sono ora la scelta preferibile perché:

1. Sono **Nativi**: Non richiedono plugin nell'IDE o dipendenze nel `pom.xml`.
2. Sono **Semantici**: Java sa che un Record è solo dati. Questo permette ottimizzazioni future e serializzazione sicura.

10.6 Copia degli Oggetti

In Java, l'operatore di assegnazione `=` non crea mai una copia dell'oggetto, ma copia solo il **riferimento**.

```
User u1 = new User("Mario");
User u2 = u1; // u2 e u1 puntano allo STESSO oggetto in memoria
```

Se modifichi `u2`, modifichi anche `u1`. Quando serve duplicare un oggetto (es. per il pattern Defensive Copy), abbiamo due strade: il metodo `clone()` o i Costruttori di Copia.

10.6.1 Perché evitare clone()

Il metodo `clone()` e l'interfaccia `Cloneable` sono considerati un errore di design nelle prime versioni di Java (persino da Joshua Bloch, autore di *Effective Java*).

I problemi principali sono:

1. **Contratto confuso:** `Cloneable` è un'interfaccia vuota (Marker Interface), ma se non la implementi, il metodo `clone()` lancia un'eccezione controllata (`CloneNotSupportedException`).
2. **Shallow Copy (Copia Superficiale):** Questa è la trappola mortale. L'implementazione di default di `clone()` copia i valori dei campi bit-per-bit.

Deep Dive: Shallow Copy vs Deep Copy

Immagina un oggetto `Manager` che contiene una lista di `Impiegati`.

- **Shallow Copy (clone standard):** Crea un nuovo oggetto `Manager`, ma copia il *riferimento* alla lista degli impiegati. *Risultato:* Il clone e l'originale condividono la **stessa** lista. Se il clone aggiunge un impiegato, questo appare anche nella lista

dell'originale. Disastroso.

- **Deep Copy (Copia Profonda):** Crea un nuovo `Manager` E crea anche una **nuova** lista di impiegati, copiando i dati uno a uno. *Risultato:* I due oggetti sono totalmente disaccoppiati.

10.6.2 La Soluzione: Copy Constructor

L'approccio standard e sicuro nell'industria è ignorare `clone()` e creare un costruttore che accetta un'istanza della stessa classe. Questo ti dà il controllo totale sulla profondità della copia.

```
public class Project {
    private String name;
    private List<String> tasks; // Oggetto mutabile!

    // Costruttore standard
    public Project(String name, List<String> tasks) {
        this.name = name;
        this.tasks = new ArrayList<>(tasks);
    }

    // COPY CONSTRUCTOR
    public Project(Project other) {
        this.name = other.name; // String è immutabile, ok copiare il riferimento

        // DEEP COPY manuale della lista
        // Creiamo una NUOVA lista contenente gli stessi elementi
        this.tasks = new ArrayList<>(other.tasks);

        // Nota: Se la lista contenesse oggetti complessi (non String),
        // dovremmo ciclare e clonare anche quelli!
    }
}
```

Listing 10.4: Copy Constructor (Best Practice)

Colloquio: Librerie per la Deep Copy

Domanda: "Se ho un oggetto molto complesso con 50 campi annidati, devo scrivere il Copy Constructor a mano?"

Risposta: No. In produzione si usano spesso scorciatoie basate sulla serializzazione:

1. **JSON:** Serializzi l'oggetto in una stringa JSON e lo deserializzi subito in un nuovo oggetto (usando Jackson o Gson). È lento ma sicuro.
2. **Librerie:** Usare `Apache Commons Lang` (`SerializationUtils.clone()`).

Capitolo 11

Java Core Essentials

In questo capitolo analizziamo i mattoni fondamentali del linguaggio. Non si tratta semplici contenitori di dati, ma di classi con comportamenti di memoria e performance specifici. Ogni sviluppatore Senior deve conoscere cosa succede "sotto il cofano" di Stringhe, Wrapper e Array per evitare colli di bottiglia e bug silenziosi.

11.1 Stringhe: Architettura e Memoria

In Java, la classe `String` è speciale. Pur essendo un oggetto (Reference Type), gode di un supporto linguistico dedicato (letterali tra virgolette, operatore `+`). Tuttavia, la sua gestione della memoria è unica nel suo genere.

11.1.1 Anatomia Interna: Compact Strings (Java 9+)

Molti pensano ancora che una Stringa sia un array di `char`. Sebbene vero fino a Java 8, la struttura è cambiata per ottimizzare la RAM.

Deep Dive: Da char[] a byte[]

In Java, un `char` occupa sempre **2 byte** (UTF-16). Questo significava che una stringa ASCII come "CIAO" occupava 8 byte, di cui 4 erano zeri inutili.

Da **Java 9** in poi, la JVM usa le **Compact Strings**:

- Internamente usa un `byte[]` (1 byte per slot) + un flag `coder`.
- Se la stringa contiene solo caratteri Latin-1 (ISO-8859-1), usa **1 byte** per carattere.
- Se contiene caratteri speciali (es. Emoji, Kanji), il `coder` cambia e usa **2 byte** (UTF-16).

Impatto: Risparmio medio del 10-15% di Heap Memory nelle applicazioni Enterprise.

11.1.2 Perché la Stringa è Immutabile?

L'immutabilità non è un caso, ma una scelta architetturale precisa per quattro motivi:

1. **String Pool:** Se le stringhe fossero mutabili, modificare una variabile cambierebbe il valore per tutte le altre referenze che puntano alla stessa stringa nel Pool. Sarebbe il caos.
2. **HashCode Caching:** Le stringhe sono le chiavi più usate nelle `HashMap`. Essendo immutabili, il loro `hashCode()` viene calcolato **una volta sola** (alla prima chiamata) e salvato in una variabile interna `hash`. Questo rende i lookup nelle mappe istantanei.

3. **Sicurezza:** Le stringhe sono usate per caricare classi, aprire file e connessioni database. Se fossero mutabili, un attaccante potrebbe modificare il path del file dopo i controlli di sicurezza ma prima della lettura.
4. **Thread-Safety:** Un oggetto immutabile è automaticamente thread-safe.

11.1.3 Lo String Constant Pool

Per risparmiare memoria, la JVM mantiene un'area speciale dello Heap (o nel Metaspace in vecchie versioni) chiamata **String Pool**.

```
String s1 = "Java";             // Va nel POOL
String s2 = "Java";             // Riusa l'oggetto nel POOL
String s3 = new String("Java"); // Crea un NUOVO oggetto nello HEAP

System.out.println(s1 == s2); // true (Stesso indirizzo)
System.out.println(s1 == s3); // false (Indirizzi diversi)
```

Listing 11.1: Pool vs Heap

Deep Dive: Il metodo intern()

È possibile spostare manualmente una stringa nel pool invocando `s3.intern()`. Questo metodo cerca se la stringa esiste già nel pool:

- Se sì, restituisce il riferimento del pool.
- Se no, aggiunge la stringa al pool e ne restituisce il riferimento.

Utile per la **deduplicazione** quando si caricano milioni di stringhe ripetitive da DB o file CSV.

11.1.4 Concatenazione: Performance

Colloquio: String vs StringBuilder vs StringBuffer

Domanda: "Qual è la differenza e quando usare quale?"

Risposta:

1. **String:** Immutabile. Ogni `+` crea un nuovo oggetto. Lento nei loop.
2. **StringBuffer:** Mutabile e **Synchronized**. Thread-safe ma lento a causa dei lock. Legacy (da non usare quasi mai).
3. **StringBuilder:** Mutabile e **Non-Synchronized**. Veloce. È lo standard per manipolare testo.

Attenzione ai Loop: Concatenare con `+` dentro un ciclo `for` ha complessità $O(n^2)$ perché ricopia l'array ad ogni iterazione. Usare `StringBuilder` porta la complessità a $O(n)$.

11.2 Wrapper Classes e Autoboxing

Java non è un linguaggio a oggetti "puro" (come Ruby o Smalltalk) perché mantiene i **tipi primitivi** (`int`, `boolean`, `double`) per motivi di performance. Tuttavia, i Generics di Java (`List<T>`, `Map<K,V>`) non supportano i primitivi. Non puoi scrivere `List<int>`.

Per colmare questo divario, Java fornisce le **Wrapper Classes** (`Integer`, `Boolean`, `Double`, ecc.), che sono oggetti che "avvolgono" il valore primitivo.

11.2.1 Autoboxing e Unboxing

Dalla versione 5, Java esegue la conversione automatica tra primitivo e wrapper.

- **Autoboxing:** Conversione da primitivo a oggetto. `Integer a = 10;` $\xrightarrow{\text{compila in}}$ `Integer a = Integer.valueOf(10);`
- **Unboxing:** Conversione da oggetto a primitivo. `int b = a;` $\xrightarrow{\text{compila in}}$ `int b = a.intValue();`

Deep Dive: Il rischio NullPointerException (NPE)

L'Unboxing nasconde un'insidia mortale.

```
Integer a = null;
// Sembra un'assegnazione sicura, ma lancia NPE!
int b = a;

```

Poiché l'unboxing invoca `a.intValue()`, se `a` è `null`, l'applicazione va in crash. Questo è un bug frequente quando si lavora con database che ammettono valori NULL su colonne numeriche.

11.2.2 La Trappola della Integer Cache

Questa è una domanda da colloquio "filtro". Serve a capire se il candidato conosce la gestione della memoria o va a caso.

Colloquio: Perché 128 diverso da 128?

Domanda: "Analizza questo codice. Cosa viene stampato e perché?"

```
Integer a = 100;
Integer b = 100;
System.out.println(a == b); // Stampa TRUE

Integer c = 1000;
Integer d = 1000;
System.out.println(c == d); // Stampa FALSE

```

Risposta:

- **Il caso 100:** Java mantiene una **Cache** interna per gli oggetti `Integer` piccoli (da **-128 a 127**). Quando scrivi `Integer a = 100`, Java vede che 100 è nel range e ti restituisce l'istanza già esistente nella cache. Quindi `a` e `b` puntano allo stesso oggetto fisico.
- **Il caso 1000:** 1000 è fuori dal range di cache. Java è costretto a creare due **nuovi** oggetti nello Heap. Quindi `c` e `d` hanno indirizzi di memoria diversi.

Lezione: Mai usare `==` per confrontare Wrapper Objects. Usa sempre `.equals()`, che funziona per tutti i valori.

11.2.3 Costo in Memoria: Primitivi vs Wrapper

Usare i Wrapper ha un costo significativo in termini di RAM (Footprint).

- **`int`:** Occupa **4 byte** sullo Stack (o nel corpo dell'oggetto).
- **`Integer`:** È un oggetto completo.

- 16 byte di Header (su 64-bit JVM).
- 4 byte per il valore int.
- Padding per allineamento.
- **Totale:** Circa **24 byte** + il riferimento (4-8 byte).

Conclusione: Un `int[]` di 1 milione di elementi occupa $\approx$ 4 MB. Un `Integer[]` (o `ArrayList<Integer>`) ne occupa oltre 24 MB. Nelle applicazioni ad alte performance, preferire array di primitivi o librerie come *Eclipse Collections* che offrono liste di primitivi.

11.3 Enums: Molto più di costanti

In linguaggi come C o C++, un'enumerazione è poco più di una lista di interi glorificata. In Java, `enum` è una **Classe** vera e propria (che estende implicitamente `java.lang.Enum`).

Questo significa che le Enum possono avere:

- **Campi di istanza** (stato).
- **Costruttori** (sempre privati).
- **Metodi** (concreti o astratti).
- Possono implementare **Interfacce**.

11.3.1 Enum con Stato e Comportamento

Un caso d'uso tipico nel backend è mappare codici di stato del database o codici di errore HTTP.

```
public enum OrderStatus {
    PENDING(1, "In attesa di pagamento"),
    SHIPPED(2, "Spedito"),
    DELIVERED(3, "Consegnato"),
    CANCELLED(99, "Annullato");

    // Campi final (Best Practice: le enum dovrebbero essere immutabili)
    private final int dbCode;
    private final String description;

    // Costruttore (Implicitamente private)
    OrderStatus(int dbCode, String description) {
        this.dbCode = dbCode;
        this.description = description;
    }

    // Metodo di utility per lookup inverso (da DB a Enum)
    public static OrderStatus fromCode(int code) {
        for (OrderStatus status : values()) {
            if (status.dbCode == code) return status;
        }
        throw new IllegalArgumentException("Codice non valido: " + code);
    }
}
```

Listing 11.2: Enum Avanzata

11.3.2 Il Singleton Pattern con Enum

Questa è una delle domande "Senior" più apprezzate.

Colloquio: Qual è il modo più sicuro per creare un Singleton?

Domanda: "Come implementi un Singleton che sia Thread-Safe e resistente alla Serializzazione?"

Risposta: Non usare il *Double-Checked Locking* o la *Lazy Initialization* manuale. Usa una **Enum a singolo elemento**.

```
public enum DatabaseConnection {
    INSTANCE; // L'unica istanza esistente

    public void executeQuery(String sql) {
        // Logica di connessione...
    }
}

```

Perché è migliore?

1. **Thread-Safety Gratuita:** La JVM garantisce che le istanze delle Enum vengano create una volta sola, in modo atomico, al caricamento della classe.
2. **Serializzazione Sicura:** Se serializzi e deserializzi un Singleton classico, potresti creare una seconda istanza. Le Enum sono protette nativamente dalla JVM contro questo problema (e anche contro gli attacchi via Reflection).

11.3.3 Enum e il Polimorfismo (Strategy Pattern)

Le Enum possono avere metodi astratti. Questo permette a ogni costante di definire un comportamento diverso. È un modo molto elegante di implementare lo **Strategy Pattern** senza creare decine di file separati.

```
public enum Operation {
    PLUS   { double apply(double x, double y) { return x + y; } },
    MINUS  { double apply(double x, double y) { return x - y; } },
    TIMES  { double apply(double x, double y) { return x * y; } },
    DIVIDE { double apply(double x, double y) { return x / y; } };

    // Ogni costante DEVE implementare questo metodo
    abstract double apply(double x, double y);
}

// Uso:
double result = Operation.PLUS.apply(2, 3); // 5.0
```

Listing 11.3: Enum con Metodi Astratti

11.4 Array: Basso livello

Gli array sono la struttura dati più semplice e vicina all'hardware. In Java, sono oggetti a dimensione fissa che memorizzano elementi dello stesso tipo in modo sequenziale in memoria. Nonostante la comodità delle 'ArrayList', gli array sono ancora imbattibili per performance pura.

11.4.1 Efficienza in Memoria: Primitivi vs Oggetti

La differenza di layout in memoria tra un array di primitivi e uno di oggetti è drammatica.

- `int[]`: Contiene i valori grezzi (es. 10, 20, 30) stoccati in un blocco di memoria **contiguo**. La CPU li adora perché può caricarli nella Cache L1 in blocco, massimizzando la velocità di iterazione.
- `Integer[]`: È un array di **riferimenti** (puntatori).
  - L'array contiene solo indirizzi di memoria.
  - Gli oggetti `Integer` reali sono sparsi ("scattered") casualmente nello Heap.
  - Iterare su questo array causa continui salti di memoria (*Cache Miss*), rallentando drasticamente l'esecuzione.

11.4.2 Il trabocchetto della Covarianza

Questa è una domanda teorica avanzata che mette in crisi molti sviluppatori.

Colloquio: Array Covarianti vs Generics Invarianti

Domanda: "Perché questo codice compila ma fallisce a runtime?"

```
1 Object[] objArr = new String[10]; // Compila!
2 objArr[0] = 100; // Lancia ArrayStoreException a Runtime
3
```

Risposta:

- Gli **Array sono Covarianti:** Se `String` è sottotipo di `Object`, allora `String[]` è considerato sottotipo di `Object[]`. Java ti permette di trattarlo come un array generico, ma a runtime controlla ogni inserimento. Se provi a mettere un `Integer` in un array che fisicamente è di `String`, la JVM lancia un'eccezione.
- I **Generics (`List<T>`) sono Invarianti:** `List<String>` **NON** è sottotipo di `List<Object>`. Il compilatore blocca subito il codice: `List<Object> l = new ArrayList<String>(); // ERRORE DI COMPILAZIONE` Questo garantisce la **Type Safety** a tempo di compilazione, che è molto più sicura.

11.4.3 Utility Class: java.util.Arrays

Mai reinventare la ruota. La classe `Arrays` offre algoritmi altamente ottimizzati.

- `sort()`:
  Usa **Dual-Pivot Quicksort** per i primitivi (veloce, $O(n \log n)$, instabile).
  Usa **Timsort** (MergeSort + InsertionSort) per gli oggetti (stabile, garantisce che oggetti uguali mantengano l'ordine relativo).
- `binarySearch()`:
  Esegue una ricerca logaritmica ($O(\log n)$) su array **già ordinati**. Se l'array non è ordinato, il risultato è indefinito.

Deep Dive: La trappola di Arrays.asList()

Il metodo `Arrays.asList(1, 2, 3)` restituisce una `List`, ma non è una `java.util.ArrayList` standard! È una classe interna (`Arrays$ArrayList`) che fa da "wrapper" (vista) sull'array originale.

Conseguenze:

- La lista è a **dimensione fissa**.
- `get()` e `set()` funzionano (modificano l'array sottostante).
- **`add()` e `remove()` lanciano `UnsupportedOperationException`.**

Soluzione: Se vuoi una lista modificabile, avvolgila: `new`

`ArrayList<>(Arrays.asList(...))`.

11.5 Date e Tempo (Java Time API)

Fino a Java 7, la gestione del tempo era affidata alle classi `java.util.Date` e `java.util.Calendar`. Queste classi sono note per essere **problematiche** sotto ogni punto di vista. Java 8 ha introdotto il package `java.time` (basato su Joda-Time), che risolve definitivamente questi problemi.

11.5.1 Perché java.util.Date è "il male"?

Se un intervistatore ti chiede perché non usi `Date`, ecco i motivi tecnici:

1. **Mutabilità:** `Date` è mutabile. Se passi una data a un metodo, quel metodo può cambiarla (usando `setTime()`) influenzando il chiamante. Questo rompe l'incapsulamento e crea bug di concorrenza.
2. **Non Thread-Safe:** La classe `SimpleDateFormat` (usata per il parsing) non è thread-safe. Se condivisa in una variabile statica tra più thread, fallisce miseramente lanciando eccezioni strane.
3. **Design Confuso:** Gli anni partono da 1900 (quindi l'anno 2023 è 123), i mesi partono da 0 (Gennaio = 0).

11.5.2 La Rivoluzione di Java 8: java.time

Le nuove classi sono **Immutabili**, **Thread-Safe** e semanticamente chiare. Bisogna scegliere la classe giusta in base al contesto.

Classe	Cosa Rappresenta
`LocalDate`	Solo una data (Anno-Mese-Giorno). Senza orario, senza fuso orario. *Esempio: Compleanno, Scadenza fattura.*
`LocalTime`	Solo un orario (Ore-Minuti-Secondi). Senza data. *Esempio: Orario di apertura negozio.*
`LocalDateTime`	Data + Orario. Senza fuso orario. *Esempio: Un evento nel calendario personale.*
`ZonedDateTime`	Data + Orario + Fuso Orario. *Esempio: Una riunione internazionale (zoom call).*
`Instant`	Un punto preciso sulla linea temporale (Timestamp UTC). Rappresenta i secondi passati dal 1970 (Epoch). *Uso: Log di sistema, salvataggio su DB.*

11.5.3 Aritmetica delle Date: Period vs Duration

Poiché le classi sono immutabili, ogni operazione di modifica restituisce una **nuova istanza** (stile `String`).

```
LocalDate oggi = LocalDate.now();
// Chaining fluente
LocalDate scadenza = oggi.plusDays(30).plusMonths(1);

```

```
// Parsing Thread-Safe
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate data = LocalDate.parse("25/12/2023", fmt);
```

Listing 11.4: Manipolazione Date

Colloquio: Period vs Duration

Domanda: "Come calcolo la differenza tra due date?"

Risposta: Dipende se vuoi ragionare in termini "umani" o "macchina".

- **Period:** Misura il tempo in Anni, Mesi e Giorni (Human Time). Tiene conto delle irregolarità (anni bisestili, mesi di 30/31 giorni). *Esempio: Tra oggi e il prossimo compleanno.*
- **Duration:** Misura il tempo in Secondi e Nanosecondi (Machine Time). È una misura fisica esatta. *Esempio: Quanto tempo è durata l'esecuzione di questo metodo?*

11.6 Optional e Null Safety

Il `NullPointerException` (NPE) è stato definito dal suo inventore, Tony Hoare, come il suo "errore da un miliardo di dollari". Prima di Java 8, l'unico modo per evitare crash era riempire il codice di controlli difensivi (`if (x != null)`).

Java 8 ha introdotto `java.util.Optional<T>`, un contenitore che può contenere un valore non-null oppure essere vuoto. L'obiettivo non è eliminare il `null` dal linguaggio, ma permettere alle API di esprimere chiaramente l'intento: *"Questo metodo potrebbe non restituire nulla, gestiscilo!"*.

11.6.1 L'approccio Funzionale (The Right Way)

L'errore del principiante è usare `Optional` come se fosse un normale controllo null (usando `isPresent()` e `get()`). L'approccio corretto è usare la catena funzionale.

```
// VECCHIO STILE (Prono a errori)
User user = repo.findById(1);
String city = "Sconosciuta";
if (user != null) {
    Address addr = user.getAddress();
    if (addr != null) {
        city = addr.getCity();
    }
}

// STILE MODERNO (Optional)
String city = repo.findById(1) // Ritorna Optional<User>
    .map(User::getAddress)     // Se c'è user, prendi address. Se no, salta.
    .map(Address::getCity)     // Se c'è address, prendi city.
    .orElse("Sconosciuta");    // Se qualcosa mancava, usa default.
```

Listing 11.5: Vecchio Stile vs Optional

11.6.2 Best Practices e Anti-Patterns

L'`Optional` va usato con disciplina.

Deep Dive: Dove NON usare Optional

1. **Mai come parametro di un metodo:** `void method(Optional<String> s)` è sbagliato. Obbliga il chiamante a wrappare i dati e non ti salva dal fatto che l'`Optional` stesso potrebbe essere `null`. Usa l'overloading dei metodi.
2. **Mai come campo di una classe:** `Optional` non è serializzabile. Se lo metti in un DTO o in una Entity JPA, avrai problemi di serializzazione e spreco di memoria. Usa `null` per i campi privati e `Optional` solo nel getter pubblico (se necessario).
3. **Mai nelle Collezioni:** `List<Optional<String>>` è un abominio. Non mettere valori assenti in una lista; semplicemente non aggiungerli.

Regola: Usa `Optional` quasi esclusivamente come **valore di ritorno** di un metodo.

11.6.3 Performance: orElse() vs orElseGet()

Questa è una domanda sottile sulle performance.

Colloquio: Differenza tra orElse e orElseGet

Entrambi forniscono un valore di default se l'Optional è vuoto.

- **`orElse(T other)`:** Il valore di default viene valutato (istanziato) **sempre**, anche se l'Optional è pieno.
- **`orElseGet(Supplier<? extends T> s)`:** Il valore di default viene valutato **lazy** (solo se l'Optional è effettivamente vuoto).

Esempio:

```
// "Calcolo pesante..." viene eseguito SEMPRE, anche se opt è pieno!
opt.orElse(metodoPesante());

// "Calcolo pesante..." viene eseguito SOLO se opt è vuoto.
opt.orElseGet(() -> metodoPesante());
```

Consiglio: Se il default è una stringa costante (""), usa `orElse`. Se è una chiamata a DB o un oggetto da istanziare, usa `orElseGet`.

11.6.4 Metodi da conoscere

- `ifPresent(Consumer)`: Esegui un'azione solo se c'è un valore (es. loggare).
- `orElseThrow()`: Restituisci il valore o lancia un'eccezione (preferibile a `get()`, che lancia `NoSuchElementException` senza un messaggio chiaro).
- `filter(Predicate)`: Se il valore c'è ma non soddisfa la condizione, l'Optional diventa vuoto.

Capitolo 12

Persistenza su File System

Sebbene i Database Relazionali siano lo standard per i dati strutturati e relazionali, il File System rimane la scelta primaria per:

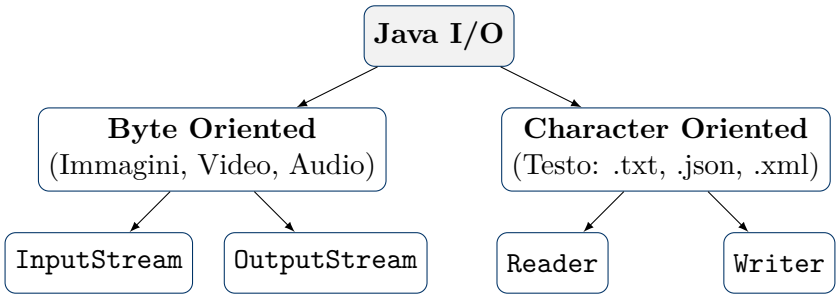
- **Configurazioni:** (es. `application.properties`, XML, YAML).
- **Dati Non Strutturati (BLOB):** Immagini, PDF, Video. Salvarli nel DB è spesso un anti-pattern; meglio salvare il percorso (path) nel DB e il file fisico su disco (o su Cloud Storage come S3).
- **Log:** Registrazione eventi e monitoraggio.
- **Data Exchange:** CSV per l'import/export massivo.

Java ha subito una profonda evoluzione nella gestione dei file. Dobbiamo distinguere tra il vecchio mondo (`java.io`) e il nuovo mondo (`java.nio`).

12.1 Java IO(Legacy): Stream

Quando leggiamo o scriviamo su file, non carichiamo (quasi) mai tutto in blocco. Usiamo gli **Stream** (flussi). Immagina un tubo dell'acqua: i dati scorrono byte dopo byte.

La distinzione fondamentale in Java è tra **Byte** e **Caratteri**.



Colloquio: Perché esistono Reader e Writer?

Domanda: "Perché non usiamo sempre InputStream anche per i file di testo?"

Risposta: A causa dell'**Encoding** (UTF-8, ISO-8859-1). Un carattere (come una lettera accentata o un simbolo speciale) può occupare più di 1 byte.

- `InputStream` legge byte crudi (01010100). Non sa cosa sia una lettera.
- `Reader` è un wrapper intelligente: legge i byte, decodifica il Charset e ti restituisce caratteri `char` o `String` leggibili.

12.2 Il Grande Divario: IO vs NIO

La differenza principale sta nel modo in cui i dati vengono movimentati.

12.2.1 Java IO (Legacy): Stream Oriented

Il pacchetto `java.io` (es. `FileInputStream`) è basato sugli **Stream**.

- **Flusso Continuo:** Leggi un byte alla volta. Non puoi andare avanti e indietro nel file (a meno di non chiudere e riaprire).
- **Blocking:** Quando chiedi di leggere un dato, il thread si blocca finché il dato non arriva. Se la rete è lenta o il disco è occupato, il thread rimane "appeso", sprecando risorse.

12.2.2 Java NIO (New IO): Block Oriented

Introdotto per alta scalabilità, `java.nio` gestisce i dati a **Blocchi**.

- **Buffer e Channel:** I dati vengono letti da un canale (Channel) e scaricati in un blocco di memoria (Buffer). Puoi muoverti avanti e indietro nel buffer liberamente.
- **Non-Blocking:** Un thread può chiedere al canale di leggere dati e, intanto che aspetta, fare altro.

Caratteristica	Java IO (Stream)	Java NIO (Buffer)
Unità di dati	Byte o Caratteri singoli	Blocchi di dati
Navigazione	Sequenziale (Solo avanti)	Casuale (Avanti/Indietro nel Buffer)
Modello Thread	Bloccante (Wait)	Non Bloccante (Selector)
Utilizzo	File piccoli, semplicità	Server ad alto traffico, File enormi

Tabella 12.1: Confronto architetturale IO vs NIO

12.3 Le Interfacce Chiave di NIO

Mentre in IO usavamo solo Stream, in NIO abbiamo tre componenti che lavorano insieme:

1. **Channel (Il Canale):** È l'equivalente moderno dello Stream, ma bidirezionale. Immaginalo come una galleria ferroviaria che collega il tuo programma al file (o al socket). Un `FileChannel` può sia leggere che scrivere.
2. **Buffer (Il Vagone):** È un contenitore di memoria (un array wrapper). Il Channel scarica i dati qui dentro. Tu non leggi mai direttamente dal Channel, leggi dal Buffer. Esistono `ByteBuffer`, `CharBuffer`, ecc.
3. **Selector (Il Controllore):** Permette a un singolo thread di monitorare molteplici Channel. È fondamentale nei server (es. Tomcat, Netty), meno nell'uso semplice dei file su disco.

12.4 L'API Moderna: java.nio.file (NIO.2)

Per lo sviluppatore che deve "solo salvare un file", usare direttamente Channel e Buffer è complesso. Java 7 ha introdotto NIO.2, che nasconde la complessità offrendo API di alto livello basate su `Path`.

12.4.1 Path vs File

- **File (Legacy):** Rappresentava un percorso ma aveva metodi limitati. Se un'operazione falliva, spesso restituiva `false` senza dire perché.
- **Path (Moderno):** È l'interfaccia che sostituisce `File`. È immutabile, più flessibile e supporta i collegamenti simbolici.

```
// Creazione del path (indipendente dal sistema operativo)
Path path = Paths.get("dati", "export.csv");

// Verifica esistenza
if (Files.notExists(path)) {
    try {
        // Crea file (lancia eccezione dettagliata se fallisce!)
        Files.createFile(path);

        // Copia
        Path backup = path.resolveSibling("export_bkp.csv");
        Files.copy(path, backup, StandardCopyOption.REPLACE_EXISTING);

    } catch (IOException e) {
        // Qui sappiamo ESATTAMENTE cosa è andato storto (es. AccessDenied)
        e.printStackTrace();
    }
}
```

Listing 12.1: Esempio: Operazioni con NIO.2

12.5 Performance: Buffered vs Unbuffered

Questa è la distinzione più importante per le performance di I/O. Spesso vedrai usare `FileReader` (standard) o `BufferedReader`. Qual è la differenza?

12.5.1 Reader/Writer Standard (Unbuffered)

Ogni chiamata a `read()` o `write()` invoca direttamente il Sistema Operativo. Se devi scrivere 1000 caratteri:

- Il tuo programma chiama la JVM.
- La JVM chiama il Kernel (System Call).
- Il Kernel scrive 1 byte sul disco.

Risultato: **1000 System Call**. Lento e inefficiente.

12.5.2 BufferedReader/BufferedWriter (Buffered)

Queste classi usano un'area di memoria interna (buffer, solitamente 8KB) come "zona di sosta". Quando scrivi 1000 caratteri:

- Il carattere viene copiato nel buffer in RAM (velocissimo).
- Solo quando il buffer è pieno (o fai `flush()`), la JVM chiama il Kernel **una sola volta** per scaricare tutto.

Risultato: **1 System Call**.

Deep Dive: System Call Overhead

Passare dallo spazio Utente allo spazio Kernel (Context Switch) è costoso per la CPU. Usare i Buffer riduce questi viaggi, migliorando le performance anche di 100 volte.

```
Path logPath = Path.of("server.log");

// Files.newBufferedReader crea un Reader bufferizzato efficiente
try (BufferedReader reader = Files.newBufferedReader(logPath)) {

    String line;
    // Leggiamo riga per riga (impossibile con FileReader base)
    while ((line = reader.readLine()) != null) {
        if (line.contains("ERROR")) {
            System.out.println("Trovato errore: " + line);
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Listing 12.2: Best Practice: Lettura Bufferizzata con NIO

Colloquio: Quando NON usare i Buffer?

Domanda: "Esiste un caso in cui è meglio non usare il buffering?"

Risposta: Sì, ma raro. Se devi scrivere un sistema di log Real-Time critico (es. scatola nera aerea) dove i dati devono essere su disco istantaneamente. Se usi il buffer e l'app crasha prima del `flush()`, perdi gli ultimi dati rimasti in RAM. In questi casi si usa `FileChannel.force(true)` per obbligare la scrittura fisica immediata.

12.6 Scrittura Rapida (Java 11+)

Per casi semplici (file di configurazione, piccoli JSON), Java 11 ha introdotto metodi che gestiscono buffer e chiusura internamente:

```
Path p = Path.of("config.json");
// Scrive tutto in un colpo solo (usa buffer internamente)
Files.writeString(p, "{ \"version\": 1 }");

// Legge tutto in memoria
String content = Files.readString(p);
```

12.7 Gestione delle Risorse e Locking

Quando lavoriamo con i file, stiamo toccando risorse del Sistema Operativo. Due concetti sono vitali:

1. **Chiusura (Close):** I "file handles" (descrittori di file) sono limitati. Se non chiudi gli stream, il file rimane bloccato (su Windows non potrai cancellarlo) e alla fine l'app crasherà con l'errore "Too many open files". Usa sempre il pattern **try-with-resources**.

2. **File Locking:** Se due processi Java (o Java e un editor di testo esterno) provano a scrivere lo stesso file contemporaneamente, i dati si corrompono. Java NIO offre il `FileLock` per bloccare un file a livello di OS.

```
// Apre il canale e tenta di acquisire il lock esclusivo
try (FileChannel channel = FileChannel.open(path, StandardOpenOption.WRITE);
     FileLock lock = channel.lock()) { // Blocca finché non ottiene accesso

    // Qui scriviamo in sicurezza...
    System.out.println("Ho il controllo esclusivo del file!");

} // Il lock viene rilasciato automaticamente qui
```

Listing 12.3: Locking di un file

12.8 Serializzazione Java

La serializzazione è il meccanismo che permette di convertire lo stato di un oggetto (i valori delle sue variabili di istanza) in un flusso di byte. Questo stream può essere:

- Salvato su disco (persistenza).
- Inviato attraverso una rete (es. RMI, socket).
- Memorizzato in cache o in un database.

Per rendere una classe serializzabile, essa deve implementare l'interfaccia "marker" (senza metodi) `java.io.Serializable`.

12.8.1 Esempio di implementazione

Ecco come avviene la scrittura (serializzazione) e la lettura (deserializzazione) utilizzando gli stream di I/O:

```
// Scrittura su file (Serializzazione)
try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("user.ser"))) {
    User user = new User("Alessandro", "passwordSegreta");
    oos.writeObject(user);
}

// Lettura da file (Deserializzazione)
try (ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("user.ser"))) {
    User userCaricato = (User) ois.readObject();
    // Nota: Il costruttore NON viene invocato durante la deserializzazione!
}
```

Colloquio: A cosa serve esattamente `serialVersionUID`?

È un identificativo univoco di versione per la classe `Serializable`. La JVM lo usa per verificare che il mittente e il destinatario abbiano caricato classi compatibili per quell'oggetto.

Il problema: Se non lo dichiari, Java ne calcola uno automaticamente basandosi su metodi, campi e nome della classe. Se cambi anche solo una virgola o aggiungi un metodo, l'hash cambia.

Lo scenario di errore:

1. Serializzi un oggetto `User` su file.
2. Aggiungi un campo `private String email` alla classe `User`.
3. Provi a leggere il file vecchio.

Senza un UID esplicito, otterrai una `java.io.InvalidClassException`.

La soluzione: Definendo `private static final long serialVersionUID = 1L;`, dici alla JVM: "Fidati, questa classe è compatibile con la versione precedente", permettendo la deserializzazione (i nuovi campi assumeranno il valore di default).

12.8.2 La keyword `transient`

Se un campo contiene dati sensibili (come password) o riferimenti a oggetti che non possono essere serializzati (come thread, stream aperti o connessioni al DB), deve essere marcato come `transient`.

Durante il processo di serializzazione, i campi `transient` vengono ignorati.

```
public class User implements Serializable {
    private static final long serialVersionUID = 1L;

    private String username;
    private transient String password; // Non finisce nel flusso di byte

    // Al momento della deserializzazione:
    // username -> avra' il valore salvato.
    // password -> sara' null (o il default per il tipo primitivo).
}
```

Deep Dive: Perché la Serializzazione Java è "Deprecated" nell'industria?

Sebbene sia ancora parte del JDK, la serializzazione nativa è sconsigliata nei nuovi progetti per due motivi principali:

1. **Sicurezza (Deserialization Vulnerabilities):** È uno dei vettori di attacco più gravi. Se un attaccante riesce a manipolare il flusso di byte in ingresso, può istanziare classi arbitrarie presenti nel classpath, portando spesso a *Remote Code Execution* (RCE).
2. **Performance e Accoppiamento:** Il formato binario di Java è verboso, lento e lega strettamente i client al linguaggio Java.

Alternative moderne: Oggi si preferisce serializzare in formati testuali agnostici come **JSON** (usando librerie come Jackson o Gson) o formati binari efficienti come **Protocol Buffers** o **Avro**.

Parte III

Java Avanzato e Moderno

Capitolo 13

Exception Handling

Gestire gli errori non significa solo evitare che l'applicazione vada in crash. Significa fornire informazioni utili per il debug e garantire che lo stato del sistema rimanga coerente anche dopo un fallimento.

13.1 Gerarchia: Checked vs Unchecked Errors

Tutto parte dalla classe madre `java.lang.Throwable`. Da qui si diramano due rami principali:

1. **Error:** Problemi gravi della JVM (es. `OutOfMemoryError`, `StackOverflowError`). L'applicazione non può recuperarli. **Non vanno mai catchati**.
2. **Exception:** Problemi che l'applicazione potrebbe gestire. Si dividono in:
   - **Checked Exceptions (Compile-time):** Estendono `Exception` (ma non `RuntimeException`). Il compilatore **obbliga** a gestirle (`try-catch` o `throws`). Rappresentano errori esterni previsti (es. File non trovato, Network down).
   - **Unchecked Exceptions (Runtime):** Estendono `RuntimeException`. Il compilatore non impone nulla. Rappresentano errori di logica del programmatore (es. `NullPointerException`, `IndexOutOfBounds`).

Colloquio: Spring e le Checked Exceptions

In Java esistono due grandi categorie di eccezioni: *Checked* e *Unchecked*. Le *Checked Exceptions* (come `SQLException`) sono obbligatorie da gestire: il compilatore forza chi scrive il codice a usare un `try/catch` oppure a dichiararle con `throws`.

Questo comportamento, però, crea un effetto a catena: se un metodo lancia una Checked Exception, anche tutti i metodi che lo chiamano devono dichiararla, anche quando non sono in grado di gestirla realmente. Questo porta a:

- firme dei metodi piene di `throws`
- codice più rumoroso e meno leggibile
- livelli dell'applicazione che conoscono dettagli che non dovrebbero conoscere

In altre parole, le Checked Exceptions tendono a **sporcare il codice** e a **rompere l'incapsulamento**, perché costringono livelli come Service e Controller a sapere che sotto esiste, ad esempio, un database SQL.

Per questo motivo, framework moderni come Spring e Hibernate convertono le *Checked Exceptions* in *Unchecked Exceptions*. Ad esempio, una `SQLException` viene trasformata in una `DataAccessException`, che è una `RuntimeException`.

Questo approccio permette di:

- mantenere le firme dei metodi pulite

- rispettare l'incapsulamento tra i livelli
- centralizzare la gestione degli errori in un punto più appropriato

L'errore non viene nascosto: continua a propagarsi, ma senza imporre vincoli artificiali a tutto il codice circostante.

Deep Dive: Il Costo delle Eccezioni

Lanciare un'eccezione è un'operazione **costosa** in termini di performance. La parte più pesante non è la creazione dell'oggetto, ma il metodo `fillInStackTrace()`, che deve scansionare lo stack dei thread per costruire la traccia dell'errore. **Consiglio:** Non usare le eccezioni per il controllo del flusso (es. uscire da un loop). Usa le eccezioni solo per situazioni davvero eccezionali.

13.2 Best Practices: Throw early, catch late

Questa è la filosofia d'oro per un codice robusto.

13.2.1 1. Throw Early (Fallire presto)

Valida gli input all'inizio del metodo. Se i dati non sono validi, lancia subito un'eccezione. Evita di eseguire metà logica per poi fallire.

```
public void processa(String input) {
    if (input == null) {
        throw new IllegalArgumentException("Input non puo' essere null");
    }
    // ... logica ...
}
```

13.2.2 2. Catch Late (Gestire tardi)

Non catturare un'eccezione se non sai come gestirla. È meglio lasciarla risalire fino a un livello superiore (es. un `@ControllerAdvice` in Spring) che può loggarla correttamente e mostrare un errore sensato all'utente. **Anti-Pattern (Swallowing):**

```
try {
    // code
} catch (Exception e) {
    e.printStackTrace(); // MAI FARLO IN PRODUZIONE! Logga su StdOut e perde il
    ↪ contesto.
    // Oppure peggio: blocco vuoto. L'errore viene "ingoiato".
}
```

13.2.3 3. Exception Chaining (Incapsulamento)

Quando catturi un'eccezione di basso livello per lanciarne una di alto livello, **passa sempre l'eccezione originale** nel costruttore. Altrimenti perdi lo stack trace originale (Root Cause).

```
try {
    fileReader.read();
} catch (IOException e) {
    // Corretto: 'e' viene passata come 'cause'
```

```
    throw new DatabaseException("Errore lettura config", e);
}
```

13.3 Try-with-resources e AutoCloseable

Fino a Java 6, la chiusura delle risorse (Stream, Connessioni DB) si faceva nel blocco `finally`, portando a codice verboso e rischio di leak se il programmatore dimenticava la chiusura.

Da Java 7, esiste il **Try-with-resources**. Qualsiasi oggetto che implementa l'interfaccia `AutoCloseable` può essere dichiarato tra le parentesi tonde del `try`.

```
// La risorsa viene chiusa AUTOMATICAMENTE alla fine del blocco
try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
} catch (IOException e) {
    logger.error("Errore I/O", e);
}
// Non serve il finally per il close()
```

Listing 13.1: Gestione Risorse Moderna

Deep Dive: Suppressed Exceptions

Cosa succede se il blocco `try` lancia un'eccezione E1, e poi la chiusura automatica (`close()`) lancia un'altra eccezione E2? Nel vecchio blocco finally, E2 avrebbe sovrascritto E1, nascondendo la vera causa dell'errore. Nel *try-with-resources*, E1 viene lanciata, e E2 viene aggiunta come **Suppressed Exception** (eccezione soppressa) a E1. Puoi recuperarla con `e.getSuppressed()`.

13.4 Custom Exceptions

Creare eccezioni personalizzate (es. `UserNotFoundException`) migliora la leggibilità. **Regola:** Estendi `RuntimeException` (Unchecked) per errori di business, estendi `Exception` (Checked) solo se vuoi forzare il client a tentare un'azione di recupero specifica.

Capitolo 14

Generics

Le Collection sono il cuore della gestione dati in memoria. Scegliere la struttura dati sbagliata può trasformare un'applicazione veloce in un sistema lento e non scalabile.

14.1 Generics: Type Safety e Flessibilità

I Generics, introdotti in Java 5, non sono semplice "zucchero sintattico", ma un meccanismo fondamentale per garantire la **Compile-time Type Safety** (sicurezza dei tipi a tempo di compilazione).

14.2 L'Era pre-Generics e i Raw Types

Prima dell'avvento dei Generics, le collezioni in Java lavoravano sui `Object`. Questo approccio, noto come utilizzo di **Raw Types**, delegava interamente al programmatore la responsabilità di inserire i tipi corretti.

```
List lista = new ArrayList(); // Raw Type (sconsigliato)
lista.add("Ciao");
lista.add(10);      // Il compilatore accetta tutto (e' Object)

// Il problema emerge SOLO a runtime
String s = (String) lista.get(1); // ClassCastException!
```

Listing 14.1: Il problema dei Raw Types

I difetti di questo approccio sono evidenti:

1. **Casting Esplicito:** Ogni operazione di lettura richiede un cast manuale.
2. **Insicurezza:** Gli errori di tipo vengono scoperti solo durante l'esecuzione (Runtime), causando crash dell'applicazione.

14.3 Il Concetto di Type Parameter

Con i Generics, spostiamo il controllo dei tipi dal Runtime al **Compile-time**. Dichiarando `List<String>`, informiamo il compilatore che quella lista deve contenere esclusivamente stringhe.

```
List<String> nomi = new ArrayList<>();
nomi.add("Anna");
// nomi.add(10); // ERRORE DI COMPILAZIONE: "incompatible types"
```

La sintassi `<T>` definisce un **Type Parameter**. È un segnaposto che verrà sostituito da un tipo concreto al momento dell'utilizzo.

```
// Definizione generica
public class Box<T> {
    private T value;
    public void set(T value) { this.value = value; }
    public T get() { return value; }
}

// Utilizzo concreto
Box<Integer> intBox = new Box<>(); // T diventa Integer
Box<String> strBox = new Box<>();  // T diventa String
```

14.4 Type Erasure: La Magia Sotto il Cofano

Una delle domande tecniche più frequenti riguarda l'implementazione interna dei Generics. Java utilizza la **Type Erasure** (Cancellazione del Tipo) per mantenere la retro-compatibilità con le versioni precedenti a Java 5.

Cosa succede realmente? Il compilatore esegue i controlli di tipo e poi **rimuove** le informazioni generiche dal bytecode generato.

- `List<String>` diventa `List` (Raw type).
- `T` viene sostituito dal suo bound più alto (solitamente `Object`).
- Il compilatore inserisce automaticamente i **Cast** necessari nel bytecode.

A Runtime, la JVM non sa la differenza tra `List<String>` e `List<Integer>`: per lei sono entrambe solo `List`.

Deep Dive: Limitazioni della Type Erasure

Poiché `T` non esiste a Runtime, alcune operazioni sono vietate:

1. **Istanziazione:** `new T()` è illegale. La JVM non saprebbe quale costruttore chiamare.
2. **Array Generici:** `new T[10]` è illegale.
3. **Instanceof:** `if (obj instanceof T)` è illegale.

Workaround: Se devi istanziare un tipo generico, devi passare esplicitamente l'oggetto `Class<T>` a runtime e usare la Reflection:

```
public Box(Class<T> clazz) {
    this.value = clazz.getDeclaredConstructor().newInstance();
}
```

14.5 Invarianza e Wildcards

A differenza degli array, i Generics in Java sono **Invarianti**.

> Anche se `Integer` estende `Number`, `List<Integer>` **NON** è un sottotipo di `List<Number>`.

Se Java permettesse la covarianza implicita, si romperebbe la Type Safety:

```
List<Integer> interi = new ArrayList<>();
List<Number> numeri = interi; // SE fosse legale...
numeri.add(3.14); // ...inserirei un Double in una lista di Integer!
```

Per gestire casi in cui serve flessibilità, Java introduce le **Wildcards** (`?`).

14.5.1 Bounded Wildcards

1. **Upper Bounded (`? extends T`):** Accetta T o qualsiasi sua sottoclasse.
2. **Lower Bounded (`? super T`):** Accetta T o qualsiasi sua superclasse.

Colloquio: Il Principio PECS (Producer Extends - Consumer Super)

Per ricordare quale wildcard usare, memorizza l'acronimo **PECS**:

- **Producer Extends:** Se la tua collezione **Produce** dati (devi solo leggerli), usa `? extends T`.
  - *Puoi leggere:* Sì (come T).
  - *Puoi scrivere:* No (non sai il tipo specifico).
- **Consumer Super:** Se la tua collezione **Consuma** dati (devi scriverci dentro), usa `? super T`.
  - *Puoi leggere:* Solo come `Object`.
  - *Puoi scrivere:* Sì (istanze di T e sottoclassi).

```
// PRODUCER: Leggo numeri (Number o figli)
public double somma(List<? extends Number> list) {
    double sum = 0;
    for (Number n : list) sum += n.doubleValue();
    // list.add(10); // ERRORE! Non posso scrivere
    return sum;
}

// CONSUMER: Scrivo interi (Integer o padri)
public void riempi(List<? super Integer> list) {
    list.add(10); // OK
    list.add(20); // OK
    // Integer n = list.get(0); // ERRORE! Ottengo solo Object
}
```

Listing 14.2: Esempio Pratico PECS

Capitolo 15

Collections Framework

15.1 Architettura e Fondamenta

Prima di analizzare le singole implementazioni (come ArrayList o HashMap), è essenziale comprendere il disegno architettonico del Java Collections Framework (JCF). Il framework non è solo un insieme di classi, ma una gerarchia rigorosa di interfacce.

15.1.1 La Gerarchia: Iterable, Collection vs Map

Il mondo delle collezioni in Java è diviso in due grandi imperi distinti. Un errore comune è pensare che tutto erediti da un unico padre. Non è così.

L'albero di Collection (java.util.Collection)

Alla base di Liste, Set e Code c'è l'interfaccia `Iterable`.

- `Iterable<T>`: È l'interfaccia "nonno". Definisce un solo metodo: `iterator()`. Qualsiasi classe la implementi può essere usata nel "for-each loop" di Java.
- `Collection<T>`: Estende `Iterable`. Definisce il contratto base per un gruppo di oggetti (add, remove, size, contains).

Sotto `Collection` troviamo le tre sotto-interfacce principali:

1. `List`: Ordine sequenziale, indici, duplicati ammessi.
2. `Set`: Unicità matematica, nessun duplicato.
3. `Queue`: Code (FIFO) e Pile, pensate per l'elaborazione (hold elements prior to processing).

L'impero di Map (java.util.Map)

Le Mappe **NON** estendono `Collection`. Sono un ramo parallelo.

Colloquio: Perché Map non estende Collection?

Domanda: Sarebbe stato logico dire che una Map è una "collezione di coppie". Perché in Java sono separate?

Risposta: Perché i contratti sono incompatibili sintatticamente e semanticamente.

- `Collection` lavora su singoli elementi: `add(E element)`.
- `Map` lavora su coppie: `put(K key, V value)`.

Se `Map` estendesse `Collection`, come implementerebbe il metodo `add(E element)`? Dovrebbe accettare solo chiavi? O solo valori? O oggetti Entry? Questa ambiguità ha portato i progettisti di Java a separare le due gerarchie. Tuttavia, le Mappe forniscono delle "viste" che sono Collezioni:

- `keySet()` → Restituisce un `Set<K>`.
- `values()` → Restituisce una `Collection<V>`.
- `entrySet()` → Restituisce un `Set<Map.Entry<K,V>>`.

15.1.2 Concetti Chiave: Mutabilità e Iteratori

Mutabilità vs Immutabilità

Fino a Java 8, le collezioni erano quasi sempre mutabili. Da Java 9 in poi, è cruciale distinguere:

- **Mutable:** `new ArrayList<>()`. Posso aggiungere/rimuovere elementi sempre.
- **Immutable:** `List.of("A", "B")`. Creata una volta, non cambia più. Tentare di modificarla lancia `UnsupportedOperationException`.
- **Unmodifiable View:** `Collections.unmodifiableList(list)`. È solo un guscio protettivo. Se la lista originale sotto cambia, cambia anche la view!

Null Safety

Non tutte le collezioni accettano `null`.

- **ArrayList, LinkedList, HashSet:** Accettano `null`.
- **TreeSet, TreeMap:** Non accettano `null` (perché non saprebbero come ordinarlo rispetto agli altri elementi).
- **ArrayDeque, PriorityQueue:** Non accettano `null`.
- **Hashtable, ConcurrentHashMap:** Non accettano `null` né come chiave né come valore (scelta di design per la concorrenza).

Fail-Fast vs Fail-Safe Iterators

Questo è il concetto più tecnico relativo agli iteratori.

Deep Dive: Fail-Fast e la ConcurrentModificationException

Le collezioni standard (ArrayList, HashMap) hanno iteratori **Fail-Fast**. Internamente mantengono un contatore chiamato `modCount` (modification count).

1. Quando crei l'iteratore, esso si salva il valore attuale di `modCount`.
2. Ogni volta che chiami `next()`, l'iteratore verifica: *"Il modCount attuale è uguale a quello che ho salvato?"*
3. Se qualcun altro (o lo stesso thread in modo sbagliato) ha aggiunto/rimosso un elemento direttamente dalla collezione, il `modCount` cambia.
4. L'iteratore se ne accorge e lancia immediatamente `ConcurrentModificationException`.

Eccezione alla regola: L'unico modo sicuro per rimuovere un elemento mentre si itera è usare il metodo `iterator.remove()`, che aggiorna sincronizzando i contatori.

15.1.3 Reference: Tabella della Complessità (Big O)

Questa tabella riassume i costi temporali delle operazioni principali. Tienila come riferimento per scegliere la struttura dati adatta.

Implementazione	Accesso	Inserimento	Rimozione	Struttura Sottostante
ArrayList	$O(1)$	$O(1)^*$	$O(n)$	Array ridimensionabile
LinkedList	$O(n)$	$O(1)$	$O(1)$	Lista doppiamente concatenata
ArrayDeque	$O(1)$	$O(1)$	$O(1)$	Array circolare (No Nulls)
PriorityQueue	$O(1)$ (peek)	$O(\log n)$	$O(\log n)$	Binary Heap (Array)
HashMap	$O(1)$	$O(1)$	$O(1)$	Array di Bucket + LinkedList/R.B. Tree
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$	Red-Black Tree

* $O(1)$ *ammortizzato. Nel caso peggiore (quando l'array deve essere ridimensionato), l'inserimento costa* $O(n)$.

15.2 Le Liste (List Interface): Sequenzialità e Memoria

La `List` è la struttura più comune: una collezione ordinata che permette duplicati e controllo posizionale preciso. Tuttavia, la scelta tra le implementazioni non è stilistica, ma architetturale.

15.2.1 ArrayList: L'implementazione di default

L'`ArrayList` deve essere la tua scelta nel 90% dei casi. È un wrapper attorno a un array primitivo:

```
transient Object[] elementData; // non-private to simplify nested class access
private int size;
```

Algoritmo di Ridimensionamento (Resizing)

Poiché gli array in Java hanno dimensione fissa, l'ArrayList simula la crescita dinamica. Quando l'array è pieno e si invoca `add()`:

1. Viene calcolata una nuova capacità. La formula nel JDK (metodo `grow`) è:

$$\text{newCapacity} = \text{oldCapacity} + (\text{oldCapacity} \gg 1)$$

   L'operatore `» 1` è uno shift bitwise a destra (divide per 2). Quindi l'array cresce del **50%** circa.
2. Viene creato un nuovo array.
3. I dati vengono copiati massivamente usando il metodo nativo `System.arraycopy()` (molto veloce perché lavora a basso livello sulla memoria).

Deep Dive: CPU Cache Locality: Il vantaggio segreto

Perché `ArrayList` è spesso più veloce di `LinkedList` anche negli inserimenti, contro la teoria classica? La risposta è nell'hardware: **CPU Cache Lines**.

Gli elementi di un array sono contigui in memoria fisica. Quando la CPU carica un elemento nella cache L1/L2, carica anche i suoi vicini. Scorrere un ArrayList è un'operazione fulminea. Una `LinkedList`, invece, ha nodi sparsi ovunque nello Heap. Questo causa continui *Cache Miss*, costringendo la CPU a costose letture dalla RAM principale.

15.2.2 LinkedList: L'alternativa collegata

Implementa sia `List` che `Deque`. È una lista **doppiamente concatenata**. Ogni elemento è racchiuso in un oggetto `Node`:

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
}
```

Il costo nascosto della memoria

Questo è un punto cruciale per l'analisi delle performance.

Deep Dive: Analisi Memory Overhead

Immagina di voler salvare un semplice Integer (es. 4 byte).

- **ArrayList:** Occupa lo spazio per il riferimento all'Integer nell'array + l'oggetto Integer stesso. (Poco overhead).
- **LinkedList:** Per ogni elemento, deve istanziare un oggetto `Node`. Su una JVM a 64 bit standard:
  - Node Header: 12 byte (approx)
  - Reference to item: 4/8 byte
  - Reference to next: 4/8 byte
  - Reference to prev: 4/8 byte

Risultato: Una LinkedList consuma **4 o 5 volte più memoria** di un ArrayList per salvare gli stessi dati.

Colloquio: ArrayList vs LinkedList

Domanda: Quando useresti veramente una LinkedList?

Risposta: Solo in scenari molto specifici:

1. Quando devo aggiungere/rimuovere elementi **in testa** alla lista frequentemente (stack/queue behavior), anche se per questo preferisco `ArrayDeque`.
2. Quando uso pesantemente gli `Iterator` per inserire/rimuovere elementi nel mezzo della lista durante l'iterazione ($O(1)$ una volta che l'iteratore è posizionato).
3. Quando non ho vincoli di memoria ma devo evitare assolutamente il "singhiozzo" di latenza causato dal resizing dell'ArrayList (copia dell'array).

15.2.3 Vector e Stack: Il passato (Legacy)

Potresti trovarli in codice vecchio, ma non dovresti mai usarli in codice nuovo.

- **Vector:** Simile a ArrayList, ma **tutti** i suoi metodi sono `synchronized`. Questo lo rende lento in contesti single-thread e non sufficientemente granulare in contesti multi-thread moderni.
  - *Sostituto:* `ArrayList` (non thread-safe) o `CopyOnWriteArrayList` (thread-safe).
- **Stack:** Estende `Vector`. È un errore di design (una Pila non dovrebbe essere un Vettore e permettere accesso per indice).
  - *Sostituto:* `ArrayDeque` (più veloce, API più pulita).

15.3 Le Mappe (Map Interface): Hashing e Internals

Le Mappe sono strutture dati che associano chiavi uniche a valori. Sebbene l'interfaccia sia semplice (`put`, `get`), la complessità ingegneristica che si nasconde dietro è notevole.

15.3.1 HashMap: The Deep Dive

La `HashMap` è basata sul principio dell'**Hashing**. Non itera sugli elementi per trovarli, ma calcola la loro posizione.

Struttura Interna: Bucket e Nodi

Internamente, una HashMap mantiene un array:

```
1 transient Node<K,V>[] table;
```

Ogni cella di questo array è chiamata **bucket**. Un bucket non contiene un solo elemento, ma è la testa di una struttura dati (una lista concatenata o un albero) che gestisce gli elementi che "cadono" in quel preciso indice.

Dalla Chiave all'Indice

Quando esegui `map.put("Key", "Value")`, avvengono tre passaggi fondamentali:

1. **Hashing:** Viene chiamato `key.hashCode()`. Questo restituisce un intero a 32 bit (che può essere negativo).
2. **Smearing (Perturbazione):** Per proteggersi da funzioni di hash scritte male, Java applica uno XOR bitwise: `(h = key.hashCode()) ^ (h >> 16)`. Questo mescola i bit alti con quelli bassi.
3. **Indexing:** L'hash deve essere mappato su un indice valido dell'array (da 0 a `n-1`). Java non usa l'operatore modulo (%), ma un'operazione bitwise molto più veloce:

$$\text{index} = (n - 1) \ \& \ \text{hash}$$

Nota: Questo funziona solo perché la capacità della tabella (n) è forzata a essere sempre una **potenza di 2**.

Deep Dive: Java 8 Revolution: Treeify (Da Lista ad Albero)

Cosa succede se molte chiavi finiscono nello stesso bucket (Collisione)?

- **Fino a Java 7:** Gli elementi venivano accodati in una LinkedList. Se un attaccante inviava chiavi studiate per collidere (HashDoS attack), la ricerca degradava a $O(n)$, bloccando la CPU.
- **Da Java 8 in poi:** Esiste una soglia `TREEIFY_THRESHOLD = 8`. Se un bucket contiene più di 8 nodi, la LinkedList viene convertita dinamicamente in un **Red-Black Tree**.

Impatto: Il caso peggiore passa da $O(n)$ a $O(\log n)$.

Colloquio: Il Contratto hashCode ed equals

Domanda: Cosa succede se due oggetti restituiscono lo stesso `hashCode` ma `equals` ritorna `false`? E viceversa?

Risposta:

- **Stesso hash, equals false:** È una **collisione** legittima. Gli oggetti finiscono

nello stesso bucket e vengono gestiti (lista/albero). La mappa funziona, ma rallenta leggermente.
- **Diverso hash, equals true: Disastro.** Hai violato il contratto. Se inserisci l'oggetto, esso finisce nel bucket A. Quando provi a cercarlo, anche se usi un oggetto "uguale", il calcolo dell'hash potrebbe puntare al bucket B. Risultato: La mappa restituisce `null` anche se l'oggetto esiste.

Performance Tuning: Load Factor

Due parametri governano le performance:
- **Initial Capacity:** Dimensione iniziale dell'array (default 16).
- **Load Factor:** Soglia di riempimento (default 0.75).

Quando `size > capacity * loadFactor`, avviene il **Re-hashing**: la dimensione dell'array raddoppia e *tutti* gli elementi vengono ricalcolati e riposizionati. È un'operazione costosa che va evitata inizializzando la mappa con la dimensione corretta se nota a priori.

15.3.2 LinkedHashMap: Ordine e Cache

Estende `HashMap`. Oltre alla struttura a bucket standard, mantiene una **Doubly Linked List** che attraversa *tutte* le entry.
- **Insertion Order (Default):** Itera gli elementi nell'ordine in cui sono stati inseriti.
- **Access Order:** Se configurata nel costruttore (`accessOrder = true`), ogni volta che un elemento viene letto (`get`) viene spostato in fondo alla lista.

Use Case Senior: Implementare una **LRU Cache** (Least Recently Used) è banale estendendo LinkedHashMap e sovrascrivendo il metodo `removeEldestEntry()`.

15.3.3 TreeMap: Navigable e Ordinata

Implementa `NavigableMap`. Mantiene le chiavi **sempre ordinate** (Natural Order o Comparator).
- **Struttura:** Red-Black Tree (albero binario bilanciato).
- **Performance:** $O(\log n)$ per `get`, `put`, `remove`. Più lenta di HashMap.
- **Funzionalità uniche:** `firstKey()`, `lastKey()`, `subMap(from, to)`.

15.3.4 Mappe Specializzate

EnumMap

Se la chiave è un `Enum`, usa sempre questa. Non usa hashing. Internamente usa un semplice **Array** indicizzato dall'`ordinal()` dell'enum. Estremamente veloce e compatta in memoria (nessun oggetto Entry wrapper, nessuna collisione possibile).

WeakHashMap

Le chiavi sono memorizzate come **WeakReference**. Se una chiave non è più referenziata da nessun'altra parte nell'applicazione, il Garbage Collector la eliminerà alla prima occasione e la `WeakHashMap` rimuoverà automaticamente l'entry corrispondente. **Use Case:** Cache di metadati associati a oggetti temporanei.

IdentityHashMap

Viola volutamente le specifiche di Map. Per confrontare le chiavi usa l'uguaglianza referenziale (`k1 == k2`) invece dell'uguaglianza logica (`k1.equals(k2)`). Utile in framework di serializzazione o per gestire topologie di oggetti (grafi) dove istanze diverse con lo stesso contenuto devono essere distinte.

15.4 I Set (Insiemi): Unicità

L'interfaccia `Set` modella l'astrazione matematica di insieme: una collezione che non può contenere duplicati.

15.4.1 La verità architetturale: I Set sono Mappe

In Java, non esiste un motore di gestione dei Set scritto da zero. Tutte le implementazioni standard sono **wrapper** attorno alle rispettive implementazioni di `Map`.

Deep Dive: HashSet Internals: Il trucco del Valore Dummy

Quando istanzi un `HashSet`, internamente stai creando una `HashMap`.

```
// Dal codice sorgente del JDK
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();

public boolean add(E e) {
    return map.put(e, PRESENT) == null;
}
```

- **Chiave:** L'elemento che inserisci nel Set.
- **Valore:** Un oggetto statico condiviso (`PRESENT`).

Implicazione Memory: Un `HashSet` consuma la stessa memoria di una `HashMap`. C'è un leggero spreco poiché ogni entry ha un riferimento a questo oggetto dummy, ma permette il riutilizzo totale del codice della mappa.

15.4.2 Le tre implementazioni principali

1. **HashSet:** Basato su `HashMap`. Non garantisce alcun ordine. È il più veloce ($O(1)$).
2. **LinkedHashSet:** Basato su `LinkedHashMap`. Mantiene l'**ordine di inserimento**. Utile se devi rimuovere duplicati da una lista mantenendo la sequenza originale.
3. **TreeSet:** Basato su `TreeMap`. Mantiene gli elementi **ordinati** (Natural o Comparator). Implementa `NavigableSet`. Costo operazioni: $O(\log n)$.

15.4.3 Operazioni Insiemistiche (Algebra dei Set)

Spesso ignorate, queste operazioni "bulk" sono potentissime e ottimizzate:

- **Unione:** `setA.addAll(setB)` $\rightarrow A \cup B$
- **Intersezione:** `setA.retainAll(setB)` $\rightarrow A \cap B$ (Mantiene in A solo ciò che esiste anche in B).
- **Differenza Asimmetrica:** `setA.removeAll(setB)` $\rightarrow A - B$ (Rimuove da A tutto ciò che è in B).

15.5 Code e Code Prioritarie (Queue & Deque)

Le code sono fondamentali per disaccoppiare la produzione di dati dalla loro elaborazione.

15.5.1 PriorityQueue: Il Binary Heap

La `PriorityQueue` non segue il FIFO (First-In-First-Out). L'elemento in testa è sempre il "minimo" (secondo il `Comparable` o `Comparator`).

Attenzione: Se iteri su una PriorityQueue con un for-each, **non** otterrai gli elementi in ordine ordinato! L'ordine è garantito solo estraendo gli elementi (`poll()`).

Struttura Interna: Heap su Array

Non usa puntatori o alberi di oggetti. Usa un singolo array che simula un albero binario completo. Dato un nodo all'indice k:

- **Figlio Sinistro:** $2k + 1$
- **Figlio Destro:** $2k + 2$
- **Genitore:** $(k - 1)/2$

Colloquio: Complessità PriorityQueue

Domanda: Perché l'inserimento non è $O(1)$? **Risposta:** Quando inserisci un elemento (`offer`), esso viene messo alla fine dell'array. Poi deve "risalire" (Bubbling Up) scambiandosi col genitore finché la regola dell'Heap non è ripristinata. Questo richiede di percorrere l'altezza dell'albero: $O(\log n)$.

- **Peek (Testa):** $O(1)$
- **Offer (Inserimento):** $O(\log n)$
- **Poll (Rimozione testa):** $O(\log n)$
- **Contains (Ricerca):** $O(n)$ (Deve scansionare l'array, non è ottimizzato per la ricerca!)

15.5.2 ArrayDeque: La Pila Moderna

L'interfaccia `Deque` (Double Ended Queue) permette inserimenti e rimozioni da entrambi i lati. `ArrayDeque` è l'implementazione da usare sempre al posto della vecchia classe `Stack`.

Circular Buffer (Ring Buffer)

Internamente usa un array trattato come se fosse circolare. Mantiene due puntatori interi: `head` e `tail`. Quando la `tail` raggiunge la fine dell'array, riparte dall'indice 0 (se c'è spazio).

Vantaggi rispetto a LinkedList:

- **Cache Locality:** Essendo un array, è amico della CPU cache.
- **No Garbage:** Non crea nodi wrapper per ogni inserimento.

Limitazione: Non accetta valori `null`.

15.6 Ordinamento e Comparazione

Le collezioni come `TreeSet` o gli algoritmi di `Collections.sort()` hanno bisogno di un criterio per stabilire se l'oggetto A viene prima dell'oggetto B. Java fornisce due meccanismi distinti.

15.6.1 Comparable: L'Ordinamento Naturale

L'interfaccia `Comparable<T>` viene implementata **dentro** la classe di dominio. Definisce "cosa sono io rispetto a un altro".

```
public class Utente implements Comparable<Utente> {
    private Integer id;
    private String nome;

    @Override
    public int compareTo(Utente altro) {
        // Ritorna negativo se this < altro
        // Zero se uguali
        // Positivo se this > altro
        return this.id.compareTo(altro.id);
    }
}
```

Implementazione Comparable corretta

Colloquio: Il bug della sottrazione

Domanda: Perché è rischioso implementare `compareTo` facendo semplicemente `return this.valore - altro.valore`?

Risposta: Per il rischio di **Integer Overflow**. Se `this.valore` è un numero positivo molto grande e `altro.valore` è un numero negativo molto grande (es. -2 miliardi), la sottrazione matematicamente darebbe un risultato positivo enorme, ma in Java i bit "girano" (overflow) e il risultato diventa negativo. L'ordinamento risulterebbe errato. **Soluzione:** Usare sempre i metodi statici helper: `Integer.compare(a, b)`.

15.6.2 Comparator: L'Ordinamento Esterno

Se non puoi modificare la classe originale (es. classe di libreria) o se vuoi ordinare in modi diversi (es. per nome, poi per data), usi `Comparator<T>`.

Da Java 8, la sintassi è estremamente pulita grazie alle lambda e ai method reference:

```
// Ordinamento a catena (Fluent API)
Comparator<Utente> byNameThenId = Comparator
    .comparing(Utente::getNome)
    .thenComparing(Utente::getId)
    .reversed(); // Inverte l'ordine totale

utenti.sort(byNameThenId);
```

15.7 Modern Java: Immutabilità (Java 9 - 21)

Fino a Java 8, creare liste immutabili era verboso e spesso inefficiente. Java 9 ha introdotto i **Collection Factory Methods** (`List.of`, `Set.of`, `Map.of`).

15.7.1 Unmodifiable View vs True Immutability

È fondamentale capire la differenza tra "non modificabile" e "immutabile".

1. **Unmodifiable View (Java 8):** `Collections.unmodifiableList(originalList)` crea solo un guscio protettivo. Se qualcuno mantiene il riferimento a `originalList` e aggiunge un elemento, la "view" vedrà quel nuovo elemento! Non è una vera immutabilità, è solo una finestra bloccata.
2. **True Immutability (Java 9+):** `List.of("A", "B")` crea una struttura dati completamente distaccata e autonoma. Non c'è nessuna lista "sotto" che può cambiare.

15.7.2 Internals: Ottimizzazione della Memoria

Perché `List.of` è meglio di `new ArrayList` seguito da un blocco?

Deep Dive: Space Efficiency in List.of

Le implementazioni restituite da `List.of` sono altamente ottimizzate per risparmiare memoria.

- **Array Fisso:** Non hanno bisogno dei campi `capacity` o di logiche di ridimensionamento.
- **Classi Specializzate:** Per liste piccolissime (1 o 2 elementi), Java 9+ non crea nemmeno un array interno! Usa classi dedicate con campi fissi (es. `List12` con campi `e0`, `e1`) per evitare l'overhead dell'oggetto array.
- **No Nulls:** Le collezioni di Java 9 non accettano `null`. Questo semplifica i controlli interni e previene errori logici.

Colloquio: Arrays.asList vs List.of

Domanda: Che differenza c'è tra `Arrays.asList(1, 2)` e `List.of(1, 2)`?

Risposta:

- `Arrays.asList`: Restituisce una lista a **dimensione fissa**, ma i cui elementi sono **mutabili**. Posso fare `list.set(0, 99)`! Inoltre, accetta `null`. È "backed" dall'array originale.
- `List.of`: Restituisce una lista **completamente immutabile**. Non posso aggiungere, rimuovere, né settare elementi. Lancia `UnsupportedOperationException` se ci provi. Non accetta `null`.

15.8 Concurrency: Thread-Safe Collections (Livello Senior)

Quando più thread accedono e modificano la stessa collezione simultaneamente, le implementazioni standard (`ArrayList`, `HashMap`) falliscono. Possono lanciare `ConcurrentModificationException` o, peggio, corrompere silenziosamente i dati (race condition) lasciando la struttura in uno stato inconsistente (es. un loop infinito dentro una LinkedList).

15.8.1 L'Approccio Legacy: Synchronized Wrappers

La prima soluzione offerta da Java è stata "avvolgere" la collezione in un guscio sincronizzato.

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
Map<K,V> syncMap = Collections.synchronizedMap(new HashMap<>());
```

Il Problema Architetturale: Questi wrapper usano un **singolo lock globale** (mutex) su tutto l'oggetto. Se il Thread A sta scrivendo, il Thread B non può nemmeno leggere. In

sistemi ad alto traffico, questo crea un collo di bottiglia (bottleneck) devastante. È come avere un'autostrada a una sola corsia.

15.8.2 ConcurrentHashMap: Il Capolavoro di Ingegneria

Questa è probabilmente la classe più sofisticata del JDK.

Read Operations (Letture)

Le letture (`get`) sono quasi sempre **Lock-Free**. Non acquisiscono lock. Leggono i valori direttamente dalla memoria (sfruttando la visibilità garantita dai campi `volatile`). Sono veloci quanto una HashMap non sincronizzata.

Write Operations (Scritture) - Java 8+

Qui avviene la magia. Invece di bloccare tutta la mappa, si usa un approccio ibrido:

1. **Bucket Vuoto (CAS):** Se il thread vuole scrivere in un bucket ancora vuoto, usa l'istruzione CPU **CAS (Compare-And-Swap)**. *"Se la cella è null, scrivi il mio nodo. Se nel frattempo è cambiata, avvisami e riprovo."* È un'operazione atomica hardware, velocissima e senza lock.
2. **Collisione (Synchronized Block):** Solo se il bucket è già occupato, il thread acquisisce un lock `synchronized`, ma **SOLO sul primo nodo** di quel specifico bucket. Mentre un thread scrive nel bucket 5, un altro può scrivere liberamente nel bucket 10.

Colloquio: ConcurrentHashMap vs Hashtable

Domanda: Perché `Hashtable` è deprecata in favore di `ConcurrentHashMap`?

Risposta: `Hashtable` usa un lock su tutta la mappa per ogni operazione (come i synchronized wrappers). `ConcurrentHashMap` blocca solo porzioni microscopiche (bucket stripping) o usa CAS. La scalabilità è incomparabile. Inoltre, `Hashtable` non accetta `null`, mentre `ConcurrentHashMap` non accetta `null` (scelta di design per evitare ambiguità in concorrenza: se `get` torna null, non sapresti se la chiave manca o se il valore è null).

15.8.3 CopyOnWriteArrayList

Una lista thread-safe molto particolare, pensata per scenari **Read-Heavy** (tante letture, pochissime scritture).

Funzionamento:

- **Lettura:** Nessun lock. I thread leggono l'array corrente.
- **Scrittura:** Non modifica l'array esistente. Crea una **nuova copia** dell'intero array, aggiunge l'elemento, e sposta il riferimento alla nuova copia.

Use Case: Lista di Event Listeners (i listener vengono aggiunti raramente all'avvio, ma notificati spessissimo). Non usare mai per cache o dati che cambiano spesso!

15.8.4 BlockingQueue: Il cuore dei sistemi asincroni

Queste code gestiscono nativamente l'attesa tra thread Produttori e Consumatori.

- **put(e):** Inserisce. Se la coda è piena, **blocca** il thread finché non si libera spazio.
- **take():** Preleva. Se la coda è vuota, **blocca** il thread finché non arriva un dato.

Implementazioni a confronto

Deep Dive: ArrayBlockingQueue vs LinkedBlockingQueue

- **ArrayBlockingQueue:** Basata su array a dimensione fissa. Usa un **singolo lock** per proteggere sia la testa che la coda. Produttori e consumatori competono per lo stesso lock. Minore throughput.
- **LinkedBlockingQueue:** Basata su nodi. Usa **due lock distinti** (`putLock` e `takeLock`). Un thread può inserire mentre un altro preleva simultaneamente. Migliore scalabilità.

SynchronousQueue

Una coda strana: ha **capacità zero**. Non mantiene elementi. Ogni operazione di inserimento (`put`) deve attendere che un altro thread faccia una `take` contestuale. È un meccanismo di "hand-off" (passaggio di mano) diretto. È la coda di default usata da `Executors.newCachedThreadPool()` per passare task ai thread senza accumularli.

Capitolo 16

Funzionale e Moderno (Java 8 - 21)

Java 8 ha introdotto il più grande cambiamento sintattico nella storia del linguaggio, spostandolo verso un paradigma ibrido (OOP + Funzionale). Le versioni successive (fino alla 21) hanno lavorato per ridurre la verbosità ("Boilerplate").

16.1 Programmazione Funzionale: Lambda e Streams

Java 8 ha introdotto il più grande cambiamento paradigmatico nella storia del linguaggio, portandolo da un approccio puramente imperativo a uno ibrido funzionale. Questo capitolo analizza come la JVM gestisce queste astrazioni a basso livello.

16.2 Lambda Expressions e Functional Interfaces

Una **Lambda Expression** non è altro che una sintassi concisa per implementare una **Functional Interface**.

16.2.1 Definizione: SAM Type

Una Functional Interface è un'interfaccia che rispetta la regola **SAM (Single Abstract Method)**: possiede esattamente un metodo astratto.

```
@FunctionalInterface // Annotation opzionale ma raccomandata
public interface Calcolatore {
    int elabora(int a, int b); // L'unico metodo astratto

    // I metodi default o static non contano
    default void reset() { ... }
}
```

La Lambda `(a, b) → a + b` fornisce l'implementazione del metodo `elabora` al volo, eliminando il boilerplate delle Classi Anonime.

Deep Dive: Sotto il cofano: invokeDynamic vs Classi Anonime

È un errore comune pensare che le Lambda siano solo "zucchero sintattico" che il compilatore trasforma in Classi Anonime (`new Interface() { ... }`).

Differenza Architetturale:

1. **Classi Anonime:** Il compilatore genera un file `.class` fisico separato su disco (es. `Main$1.class`) per ogni classe anonima. Questo aumenta la dimensione del JAR e

il carico sul ClassLoader.

2. **Lambda Expressions:** Non generano classi al momento della compilazione. Il compilatore emette un'istruzione bytecode chiamata `invokedynamic` (introdotta in Java 7).

La creazione dell'oggetto reale è delegata alla JVM a **Runtime**. Questo permette alla JVM di ottimizzare la creazione (es. riutilizzando l'istanza se la lambda non cattura variabili) e riduce l'occupazione di memoria (Metaspace).

16.2.2 Cheat Sheet: Le Interfacce Funzionali Standard

Non serve creare ogni volta la tua interfaccia `Calcolatore`. Il package `java.util.function` copre il 90% dei casi d'uso comuni, specialmente negli Stream.

Interfaccia	Input	Output	Metodo Astratto
`Predicate<T>`	`T`	`boolean`	`boolean test(T t)`
Uso:	`stream.filter(x → x > 10)`		
`Function<T, R>`	`T`	`R`	`R apply(T t)`
Uso:	`stream.map(user → user.getName())`		
`Consumer<T>`	`T`	`void`	`void accept(T t)`
Uso:	`stream.forEach(System.out::println)`		
`Supplier<R>`	`None`	`R`	`R get()`
Uso:	`Optional.orElseGet(() → new User())`		
`UnaryOperator<T>`	`T`	`T`	(estende Function)
Uso:	`list.replaceAll(s → s.toUpperCase())`		

Tabella 16.1: Interfacce Funzionali Standard di Java 8

16.3 Method References: L'operatore ::

Le Method References sono una sintassi abbreviata per le Lambda Expression che chiamano un metodo esistente. Rendono il codice estremamente leggibile.

Tipo	Lambda	Method Reference
Static Method	`x → Math.abs(x)`	`Math::abs`
Instance Method (oggetto specifico)	`str → System.out.println(str)`	`System.out::println`
Instance Method (oggetto arbitrario)	`(s1, s2) → s1.compareTo(s2)`	`String::compareTo`
Constructor	`() → new ArrayList<>()`	`ArrayList::new`

Tabella 16.2: Tipi di Method References

Colloquio: Quando usare `String::compareTo` vs `str::compareTo`?

È una domanda sottile.

- **`String::compareTo`** (Class::instanceMethod): Il primo parametro della lambda diventa il target del metodo (es. `s1.compareTo(s2)`).
- **`myString::compareTo`** (instance::instanceMethod): Il metodo viene chiamato su un oggetto specifico già esistente (`myString`), ignorando il parametro della lambda come target.

16.4 Variable Capture: "Effectively Final"

Le Lambda possono accedere alle variabili dello scope esterno (Closure), ma con una restrizione fondamentale: le variabili devono essere **Effectively Final**.

```
int fattore = 10;
// fattore = 5; // Se decommenti questa riga, la lambda sotto non compila!

Stream.of(1, 2, 3).map(n -> n * fattore).forEach(System.out::println);
```

Perché? Le variabili locali vivono nello Stack, mentre la Lambda (che è un oggetto) vive nello Heap e potrebbe essere eseguita da un altro thread molto dopo che il metodo originale è terminato (e lo stack distrutto). Java passa alla lambda una **copia** del valore. Se il valore cambiasse, la copia sarebbe inconsistente.

16.5 Stream API: Elaborazione Dichiarativa

Introdotte in Java 8, le Stream rappresentano un cambio di paradigma da *imperativo* (come fare, es. cicli for) a *dichiarativo* (cosa voglio ottenere). Una `Stream` non è una struttura dati, ma una **pipeline** che trasporta dati da una sorgente, applica trasformazioni e produce un risultato, senza mai modificare la sorgente originale (immutabilità).

Il ciclo di vita è rigoroso:

Sorgente → Operazioni Intermedie (Lazy) → Operazione Terminale (Eager)

16.5.1 Legenda delle Operazioni

È fondamentale distinguere le operazioni. Una stream senza operazione terminale è come un rubinetto chiuso: l'acqua (i dati) non scorre.

Operazioni Intermedie (Lazy)

Restituiscono sempre una nuova `Stream<T>`. Non eseguono nulla immediatamente, ma "configurano" la pipeline.

- `filter(Predicate)`: Mantiene gli elementi che soddisfano la condizione.
- `map(Function)`: Trasforma ogni elemento (1:1).
- `flatMap(Function)`: Trasforma ogni elemento in una Stream e appiattisce il risultato (1:N).
- `sorted(Comparator)`: Ordina gli elementi (Stateful: richiede di vedere tutti i dati).
- `distinct()`: Rimuove i duplicati (usa `equals()`).
- `limit(n) / skip(n)`: Taglia o salta elementi (Short-circuiting).
- `peek(Consumer)`: Esegue un'azione senza modificare lo stream (utile per debug).

Operazioni Terminali (Eager)

Avviano l'elaborazione, producono un risultato (o void) e **chiudono** la stream. Dopo questo punto la stream non è più riutilizzabile.

- `collect(Collector)`: Accumula i risultati in una List, Map, Set, etc.
- `forEach(Consumer)`: Itera su ogni elemento (ritorna `void`).
- `reduce(BinaryOperator)`: Combina gli elementi in un unico valore (es. somma).
- `count()`: Conta gli elementi.
- `anyMatch / allMatch / noneMatch`: Restituiscono booleani (Short-circuiting).

- `findFirst / findAny`: Restituiscono un `Optional<T>`.

16.5.2 Lazy Evaluation e Loop Fusion

La JVM è estremamente intelligente nell'ottimizzare le stream. Grazie alla *Lazy Evaluation*, le operazioni non vengono eseguite una per volta sull'intera collezione.

Deep Dive: Cos'è il Loop Fusion?

Invece di iterare la lista 3 volte per 3 operazioni diverse (filtro, mappa, limite), la JVM fonde le istruzioni in un **singolo passaggio** sui dati. Inoltre, grazie allo *Short-Circuiting*, si ferma appena ha ottenuto il risultato necessario.

```
List<String> nomi = Arrays.asList("Anna", "Bob", "Alice", "Carlo");

// Qui stiamo solo DEFINENDO la pipeline. Nessun dato viene toccato.
Stream<String> stream = nomi.stream()
    .filter(s -> {
        System.out.println("Filter: " + s);
        return s.startsWith("A");
    })
    .map(s -> {
        System.out.println("Map: " + s);
        return s.toUpperCase();
    })
    .limit(1); // Mi serve solo il primo risultato!

System.out.println("--- Inizio Esecuzione ---");
// L'esecuzione parte SOLO ora (collect è terminale)
List<String> risultato = stream.collect(Collectors.toList());
System.out.println("Risultato: " + risultato);

/* OUTPUT CONSOLE:
--- Inizio Esecuzione ---
Filter: Anna       <-- Trovato match!
Map: Anna          <-- Trasformato subito
Risultato: [ANNA]  <-- Stop! Bob, Alice e Carlo non sono stati nemmeno letti.
*/
```

Listing 16.1: Dimostrazione di Lazy Evaluation e Short-Circuiting

16.5.3 Il pericolo dei Parallel Stream

Il metodo `.parallelStream()` suddivide lo stream in più chunk ed elabora ogni pezzo su un thread diverso, ricomponendo poi i risultati. Sembra una bacchetta magica per le performance, ma nasconde un grave pericolo architetturale.

Colloquio: Perché evitare Parallel Stream per operazioni I/O?

È una domanda da Senior. La risposta sta nel **Thread Pool**. Tutti i parallel stream all'interno di una JVM (a meno di configurazioni complesse) condividono lo stesso thread pool globale: il **Common ForkJoinPool**. La dimensione di questo pool è fissa e limitata:

$$\text{Pool Size} = \text{Numero Core CPU} - 1$$

Scenario Disastroso: Immagina di usare `parallelStream()` per fare chiamate HTTP o query al Database (I/O Blocking). I thread del ForkJoinPool si bloccheranno in attesa della risposta del server/DB. Bastano poche richieste concorrenti per **saturare completamente** il pool globale.

Conseguenza: L'intera applicazione si blocca o rallenta drasticamente ("Starvation"), perché nessun altro parallel stream (anche in parti completamente diverse del codice) può trovare un thread libero per essere eseguito.

Best Practice: Usa i parallel stream **SOLO** per calcoli puri (CPU-intensive) su dataset enormi già presenti in memoria, dove non c'è attesa di I/O.

16.6 Optional: Evitare NullPointerException

`Optional<T>` è un contenitore che può contenere un valore non nullo oppure essere vuoto. Serve a rendere esplicito nel ritorno di un metodo che "il valore potrebbe non esserci".

Anti-Pattern da evitare:

```
Optional<String> opt = trovaUtente();
if (opt.isPresent()) {  // Non farlo! E' uguale a if (x != null)
    System.out.println(opt.get());
}
```

Approccio Funzionale (Best Practice):

```
trovaUtente()
    .map(String::toUpperCase)
    .ifPresent(System.out::println);

// Oppure lanciare eccezione se manca
String user = trovaUtente().orElseThrow(() -> new UserNotFoundException());
```

Nota Tecnica: `Optional` non è `Serializable`. Non usarlo mai come campo di una classe (Entity o DTO), ma solo come tipo di ritorno dei metodi.

16.7 Reflection API e Annotations: Come funzionano i Framework

Questo è il prerequisito per capire Spring.

- **Reflection:** Permette a un programma Java di ispezionare e modificare il proprio comportamento a runtime (accedere a metodi privati, istanziare classi dato il nome stringa). È potente ma lenta (disabilita le ottimizzazioni JIT).
- **Annotations (`@Override`, `@Entity`):** Sono metadati. Di per sé **non fanno nulla**.

Colloquio: Come fa Spring a sapere che una classe è un Bean?

Spring usa la Reflection all'avvio:

1. Scansiona tutte le classi nel classpath (È una lista di percorsi, non un contenitore di classi). Quindi va a guardare, una per una, tutte le classi che si trovano in tutte le cartelle e in tutti i .jar elencati nel classpath.
2. Controlla se la classe ha l'annotazione `@Component` (o derivate).
3. Se sì, usa `Class.newInstance()` (o costruttori) per crearne un'istanza e gestirla.

Senza Reflection, framework come Spring, Hibernate o JUnit non potrebbero esistere.

16.8 Novità recenti (Java 14 - 21)

Per dimostrare di essere aggiornati, citare queste feature è essenziale.

16.8.1 1. Records (Java 16)

Classi immutabili "data-carrier". Sostituiscono i DTO pieni di boilerplate.

```
public record Point(int x, int y) {}
// Genera automaticamente: costruttore, equals, hashCode, toString e getters.
```

16.8.2 2. Pattern Matching per instanceof (Java 16)

Basta cast manuali.

```
if (obj instanceof String s) { // Crea variabile 's' automaticamente castata
    System.out.println(s.length());
}
```

16.8.3 3. Sealed Classes (Java 17)

Permettono di controllare chi può estendere una classe (gerarchie chiuse). Ottimo per il Domain Modeling sicuro.

```
public abstract sealed class Shape permits Circle, Square { ... }
```

Capitolo 17

Multithreading e Concorrenza

Scrivere codice che esegue più operazioni simultaneamente è essenziale per sfruttare le moderne CPU multi-core e mantenere le applicazioni reattive. Tuttavia, la concorrenza introduce classi di bug complesse (Race Conditions, Deadlocks) e richiede una comprensione profonda di come la JVM interagisce con il Sistema Operativo sottostante.

17.1 Processi vs Threads e Ciclo di Vita

Per capire veramente come funziona un Thread in Java, dobbiamo scendere un gradino sotto il linguaggio e osservare l'architettura del Sistema Operativo.

17.1.1 Architettura: User Space, Kernel e System Calls

La memoria di un computer gestito da un Sistema Operativo moderno (Linux, Windows, macOS) è divisa in due zone di sicurezza rigorosamente separate:

1. **User Space (Spazio Utente):** È la zona "non privilegiata" dove girano le applicazioni standard, inclusa la JVM. Qui il codice ha accesso alla memoria allocata per il programma (Heap Java), ma **non ha accesso diretto all'hardware** (Disco, Scheda di Rete, Thread della CPU).
2. **Kernel Space (Spazio Kernel):** È il "cuore" del sistema operativo. Ha privilegi totali e controlla l'hardware. Gestisce lo scheduling dei processi e l'I/O.

Deep Dive: Cos'è una System Call?

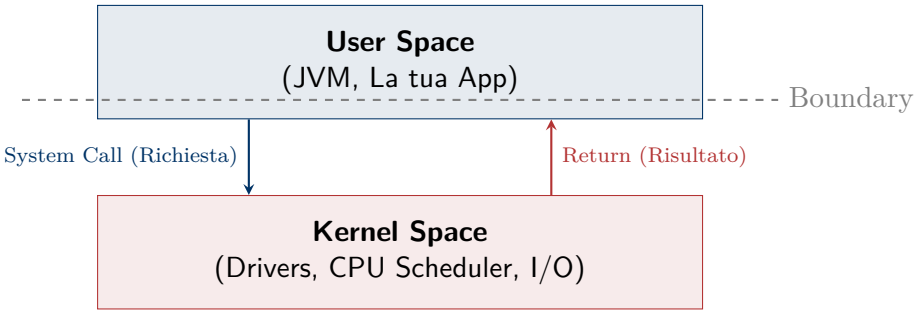
Quando un thread Java vuole fare un'operazione "reale" (es. leggere un file, aprire una connessione Socket, o creare un nuovo thread), non può farlo da solo perché gira in User Space. Deve chiedere il permesso al Kernel.

Questa richiesta si chiama **System Call (SysCall)**.

Il flusso è costoso:

1. Il programma in User Space invoca una funzione di libreria.
2. Viene scatenata un'interruzione software.
3. La CPU esegue un **Context Switch**: salva lo stato del programma corrente, cambia modalità in *Kernel Mode* ed esegue l'operazione.
4. Il risultato viene copiato dallo spazio Kernel allo spazio User.
5. La CPU torna in *User Mode* e riprende il programma.

Questo costo (overhead) è il motivo per cui creare migliaia di thread nativi è sconsigliato.



17.1.2 Processo vs Thread

Spesso confusi, hanno differenze strutturali enormi:

- **Processo:** È un'istanza di un programma in esecuzione (es. l'intero processo `java.exe`).
  - Ha un proprio spazio di indirizzamento memoria **isolato**. Se un processo crasha, non influenza gli altri.
  - La comunicazione tra processi (IPC) è lenta e complessa.
- **Thread:** È l'unità più piccola di esecuzione all'interno di un processo.
  - I thread di uno stesso processo **condividono la memoria** (lo Heap).
  - Hanno però un proprio **Stack** privato (per le variabili locali e la catena delle chiamate).
  - Sono "leggeri" rispetto ai processi, ma "pesanti" in termini assoluti (circa 1-2MB di RAM riservata per lo stack di ogni thread).

17.1.3 User Threads vs Daemon Threads

In Java, i thread non sono tutti uguali. Esiste una proprietà chiamata "Daemon" che cambia il modo in cui la JVM decide quando spegnersi.

- **User Thread (Default):** Sono i thread "lavoratori" o "protagonisti". La JVM **resta in vita** finché esiste almeno un User Thread attivo (anche se il metodo `main` è terminato).
- **Daemon Thread:** Sono thread di "servizio" o "background" (es. il Garbage Collector, o un thread che invia statistiche di monitoraggio). La loro esistenza dipende dagli User Thread. **Regola:** Se tutti gli User Thread terminano, la JVM uccide istantaneamente tutti i Daemon Thread rimasti e si chiude.

```
Thread backgroundTask = new Thread(() -> {
    while(true) {
        System.out.println("Pulizia cache in background...");
        try { Thread.sleep(1000); } catch (InterruptedException e) {}
    }
});

// FONDAMENTALE: Va settato PRIMA di chiamare start()
// Se lo fai dopo, lancia IllegalThreadStateException.
backgroundTask.setDaemon(true);

backgroundTask.start();
System.out.println("Main terminato");
// Risultato: La JVM si chiude subito dopo questa riga.
// Il thread 'backgroundTask' viene ucciso immediatamente,
// probabilmente prima di stampare "Pulizia...".
```

Listing 17.1: Creazione di un Daemon Thread

17.1.4 I Metodi Fondamentali (API Lifecycle)

Oltre a `start()`, la classe `Thread` espone metodi critici per orchestrare l'esecuzione e la cooperazione tra thread. È vitale distinguere tra metodi **statici** (che agiscono sempre sul thread *corrente*) e metodi **di istanza** (che agiscono sul thread rappresentato dall'oggetto).

1. `static void sleep(long millis)`: Mette in pausa il thread **corrente** per un tempo determinato. Lo stato del thread passa da *RUNNABLE* a *TIMED_WAITING*. **Nota:** Non rilascia i lock o i monitor che il thread possiede!
2. `void join()`: Permette a un thread di "aspettare" la fine di un altro. Se il Thread A chiama `B.join()`, il Thread A si blocca (entra in *WAITING*) finché il Thread B non termina la sua esecuzione. È essenziale per coordinare risultati.
3. `static void yield()`: È un suggerimento allo Scheduler del Sistema Operativo. Il thread corrente dice: *"Ho finito la mia fase critica, se c'è qualcun altro che vuole usare la CPU, mi faccio da parte"*. **Attenzione:** Lo scheduler può ignorare questo suggerimento. Non garantisce nulla.
4. `void interrupt()`: Invia un segnale di interruzione al thread target. Non ferma il thread brutalmente (come il deprecato `stop()`), ma setta solo un flag booleano interno. Sta al codice del thread controllare questo flag e terminare gentilmente.

Colloquio: Start() vs Run()

Domanda: "Qual è la differenza tra chiamare `t.start()` e `t.run()`?"

Risposta: È la differenza tra "Concorrenza" e "Chiamata di metodo sequenziale".

- `t.run()`: Esegue il corpo del metodo `run` nel **thread corrente** (es. main). Non viene creato nessun nuovo stack, non c'è parallelismo. È una semplice chiamata di funzione.
- `t.start()`: Fa una System Call al sistema operativo per allocare un nuovo Thread nativo, crea un nuovo Stack, e istruisce lo scheduler OS di eseguire il metodo `run` in quel nuovo contesto.

Colloquio: Differenza tra sleep() e wait()

Questa è una delle domande più frequenti in assoluto. Entrambi mettono in pausa il thread, ma con una differenza enorme nella gestione della sincronizzazione.

- `Thread.sleep()`:
  - È un metodo statico di `Thread`.
  - **Mantiene i Lock:** Se il thread ha acquisito un lock (è dentro un `synchronized`), va a dormire *tenendosi le chiavi in tasca*. Nessun altro può entrare in quella sezione critica.
  - Usato per: attese temporali.
- `Object.wait()`:
  - È un metodo di `Object` (ogni oggetto ce l'ha).
  - **Rilascia il Lock:** Il thread rilascia il monitor e va in attesa. Altri thread possono entrare nel blocco `synchronized`.
  - Usato per: comunicazione inter-thread (notify).

Deep Dive: Interrupt: I tre metodi della confusione

Gestire l'interruzione è complicato perché ci sono tre metodi con nomi simili:

1. `interrupt()` (Istanza): "Bussa alla porta". Setta il flag *interrupted status* a `true`.

2. `isInterrupted()` (Istanza): Controlla il flag senza modificarlo. Restituisce `true/false`.
3. `Thread.interrupted()` (Statico): Controlla il flag del thread corrente e **lo resetta a false!** *Perché resettarlo?* È utile se vuoi "consumare" l'interruzione e continuare a lavorare pulito, ma è fonte di molti bug se usato per sbaglio al posto di `isInterrupted()`.

17.2 Problemi di Concorrenza: Race Condition e Deadlock

La programmazione concorrente introduce una classe di bug subdoli legati alla gestione dello *stato condiviso mutabile*. Se due thread operano sugli stessi dati senza coordinamento, il risultato diventa dipendente dalla tempistica di esecuzione (interleaving).

17.2.1 Race Condition (Condizione di Corsa)

Una Race Condition si verifica quando la correttezza di un calcolo dipende dalla sequenza temporale relativa o dall'ordine di esecuzione di più thread. Il caso più classico è l'operazione **Read-Modify-Write** non atomica.

```
public class UnsafeCounter {
    private int count = 0;

    // QUESTO METODO NON E' THREAD-SAFE!
    public void increment() {
        count++;
    }

    public int getCount() { return count; }
}
```

Listing 17.2: Esempio di Race Condition (Il Contatore)

Se lanciamo due thread che chiamano `increment()` 1000 volte ciascuno, ci aspettiamo che il risultato finale sia 2000. Spesso, invece, otteniamo un numero inferiore (es. 1998, 1850). Perché?

Deep Dive: Anatomia di count++

Agli occhi del programmatore Java, `count++` sembra un'unica istruzione. Agli occhi della CPU (e del Bytecode), sono **tre** istruzioni distinte:

1. **READ:** Leggi il valore corrente di `count` dalla memoria in un registro locale (es. valore 10).
2. **MODIFY:** Incrementa il valore nel registro (10 + 1 = 11).
3. **WRITE:** Scrivi il nuovo valore (11) nella memoria principale.

Lo scenario del disastro:

- **Thread A** legge 10.
- *Context Switch! Il sistema operativo mette in pausa A e avvia B.*
- **Thread B** legge 10 (perché A non ha ancora scritto 11).
- **Thread B** calcola 11 e scrive 11.
- *Context Switch! Torna A.*
- **Thread A** (che aveva già calcolato 11 nel suo registro) scrive 11.

Risultato: Due incrementi effettuati, ma il contatore è salito solo di 1. L'aggiornamento

di B è stato sovrascritto ("Lost Update").

17.2.2 Deadlock (Stallo)

Il Deadlock è una situazione in cui due o più thread sono bloccati per sempre, in attesa l'uno dell'altro. Nessuno può progredire. Accade spesso quando si usano più lock annidati.

```
public class DeadlockDemo {
    private final Object lockA = new Object();
    private final Object lockB = new Object();

    public void thread1() {
        synchronized(lockA) {
            System.out.println("T1: Preso Lock A");
            try { Thread.sleep(100); } catch (Exception e){}

            synchronized(lockB) { // T1 vuole B, ma ce l'ha T2
                System.out.println("T1: Preso Lock B");
            }
        }
    }

    public void thread2() {
        synchronized(lockB) {
            System.out.println("T2: Preso Lock B");
            try { Thread.sleep(100); } catch (Exception e){}

            synchronized(lockA) { // T2 vuole A, ma ce l'ha T1
                System.out.println("T2: Preso Lock A");
            }
        }
    }
}

public class MainDeadlock {
    public static void main(String[] args) {
        DeadlockDemo demo = new DeadlockDemo();

        // **Creazione dei Thread**
        Thread t1 = new Thread(() -> demo.thread1(), "Thread-1");
        Thread t2 = new Thread(() -> demo.thread2(), "Thread-2");

        // **Avvio dei Thread**
        t1.start();
        t2.start();

        System.out.println("Avviati Thread-1 e Thread-2");
    }
}
```

Listing 17.3: Esempio classico di Deadlock

In questo scenario:

- T1 possiede A e aspetta B.
- T2 possiede B e aspetta A.

- È un abbraccio mortale (Circular Wait).

Colloquio: Come prevenire e diagnosticare un Deadlock?

Domanda: "Come fai a evitare i deadlock nel tuo codice?"

Risposta:

1. **Lock Ordering (Ordine Globale):** È la soluzione matematica. Se tutti i thread acquisiscono i lock nello stesso ordine (es. Sempre prima A, poi B), il deadlock circolare diventa impossibile.
2. **Timeouts:** Usare `ReentrantLock.tryLock(timeout)` invece di `synchronized`. Se non riesco a prendere il lock entro 1 secondo, rinuncio e torno indietro (evitando di bloccarmi per sempre).
3. **Diagnosi:** Se un'applicazione Java si blocca in produzione, uso il comando `jstack <pid>`. Il dump mostrerà chiaramente: *"Found one Java-level deadlock"*.

17.3 Sincronizzazione e Lock Base

Per risolvere le Race Condition, dobbiamo introdurre dei meccanismi di controllo per gestire l'accesso allo stato condiviso. Java offre strumenti nativi per garantire due proprietà fondamentali:

1. **Atomicità (Mutual Exclusion):** Solo un thread alla volta può eseguire un blocco di codice.
2. **Visibilità (Memory Consistency):** Le modifiche fatte da un thread devono essere immediatamente visibili agli altri thread (bypassando le cache della CPU).

17.3.1 La keyword `synchronized`

È il meccanismo di locking più antico e semplice di Java. Si basa sul concetto di **Monitor Lock** (o Intrinsic Lock): in Java, *ogni oggetto* può fungere da lucchetto.

- Quando un thread entra in un blocco `synchronized`, acquisisce il lock dell'oggetto.
- Se un altro thread prova a entrare, trova il lock occupato e viene messo in stato **BLOCKED** (sospeso dal sistema operativo) finché il primo non esce.
- Garantisce sia **Atomicità** che **Visibilità**.

```
public class SafeCounter {
    private int count = 0;
    // Lock implcito: 'this' (l'istanza corrente)
    public synchronized void increment() {
        count++;
    }
    // Anche le letture devono essere sincronizzate per garantire la visibilità!
    public synchronized int getCount() {
        return count;
    }
}
```

Listing 17.4: Rendere il contatore Thread-Safe

17.3.2 La keyword `volatile`

Spesso fraintesa, `volatile` è una forma di sincronizzazione "leggera". **Garantisce solo la Visibilità, NON l'Atomicità.**

Deep Dive: Il problema della Cache CPU

Le CPU moderne hanno cache (L1, L2, L3) velocissime. Se il Thread A modifica una variabile `flag = true`, potrebbe scrivere questo valore solo nella sua Cache L1 locale, senza aggiornare subito la RAM principale. Il Thread B (che gira su un altro Core) continua a leggere `flag = false` dalla sua cache.

Dichiarando `private volatile boolean flag`, obblighi la JVM a:

1. Scrivere immediatamente le modifiche in **Main Memory (RAM)**.
2. Invalidare le cache degli altri core per quella variabile.

Colloquio: Volatile e i++

Domanda: "Se dichiaro `volatile int count = 0`, l'operazione `count++` diventa thread-safe?"

Risposta: NO. `volatile` garantisce che tutti vedano l'ultimo valore scritto, ma non protegge la sequenza "Leggi-Modifica-Scrivi". Due thread potrebbero ancora leggere lo stesso valore contemporaneamente e sovrascriversi. Usa `volatile` solo per flag di stato (es. `isRunning`) o variabili scritte da un solo thread e lette da molti.

17.3.3 Variabili Atomiche e CAS (Compare-And-Swap)

Usare `synchronized` è costoso perché blocca i thread (Context Switch). Per operazioni semplici (contatori, accumumulatori), Java fornisce il pacchetto `java.util.concurrent.atomic`. Queste classi non usano lock, ma si basano su una strategia **Optimistic** supportata dall'hardware.

```
1 private AtomicInteger count = new AtomicInteger(0);
2
3 public void increment() {
4     // Thread-safe, veloce e non-bloccante
5     count.incrementAndGet();
6 }
```

Listing 17.5: AtomicInteger

Deep Dive: Deep Dive: Come funziona il CAS?

Il **Compare-And-Swap (CAS)** è un'istruzione di basso livello della CPU. Accetta tre parametri:

1. **M:** Indirizzo di memoria.
2. **E:** Valore Atteso (Expected).
3. **N:** Nuovo Valore (New).

L'algoritmo atomico eseguito dalla CPU è: *"Guarda l'indirizzo M. Se contiene E, scrivi N e ritorna true. Altrimenti (qualcun altro ha cambiato il valore nel frattempo), non fare nulla e ritorna false."*

Java usa questo in un ciclo (spinlock) finché non ha successo:

```
int current, next;
do {
    current = get();      // Leggo il valore attuale (es. 10)
    next = current + 1;   // Calcolo il prossimo (11)
    // Provo a scrivere atomically. Se nel frattempo è diventato 12,
    // il CAS fallisce e il ciclo si ripete rileggendo 12.
} while (!compareAndSet(current, next));
```

17.4 Strumenti di Sincronizzazione Avanzata

Sebbene la keyword `synchronized` sia semplice da usare, è rigida: il blocco è implicito, non si può interrompere un thread in attesa del lock e non si può tentare di acquisire il lock con un timeout. Il pacchetto `java.util.concurrent.locks` offre primitive più potenti e flessibili.

17.4.1 ReentrantLock: Il Lock Esplicito

Il `ReentrantLock` è un'implementazione dell'interfaccia `Lock` che offre le stesse garanzie di atomicità e visibilità di `synchronized`, ma con controllo manuale. Si chiama "Reentrant" perché lo stesso thread può riacquisire il lock più volte (ricorsivamente) senza bloccarsi su se stesso.

Colloquio: Synchronized vs ReentrantLock

Domanda: "Perché dovrei complicarmi la vita con `ReentrantLock` invece di usare `synchronized`?"

Risposta: Ci sono tre casi d'uso dove `synchronized` fallisce:

1. **Timeout (Prevenzione Deadlock):** Con `synchronized`, se il lock è occupato, aspetti in eterno. Con `lock.tryLock(1, TimeUnit.SECONDS)`, se non ottieni il lock entro un secondo, puoi rinunciare e fare altro.
2. **Fairness (Equità):** `synchronized` è "ingiusto": se 100 thread aspettano, ne sceglie uno a caso (rischio Starvation). `ReentrantLock(true)` gestisce una coda FIFO: chi arriva prima, entra prima.
3. **Interruptibility:** Un thread bloccato su `synchronized` non può essere interrotto. Con `lock.lockInterruptibly()`, puoi svegliarlo.

```
private final ReentrantLock lock = new ReentrantLock();

public void safeMethod() {
    lock.lock(); // 1. Acquisisco il lock
    try {
        // Sezione Critica: Eseguo operazioni rischiose
        doCriticalWork();
    } finally {
        // 2. FONDAMENTALE: Rilascio nel finally
        // Se il codice sopra lancia un'eccezione, il lock DEVE essere rilasciato
        // altrimenti avremo un Deadlock globale.
        lock.unlock();
    }
}
```

Listing 17.6: Pattern Try-Finally (Obbligatorio)

17.4.2 Semaphore: Gestione dei Permessi (Throttling)

Mentre un Lock è un meccanismo di **Mutua Esclusione** (accesso 1 a 1), un `Semaphore` gestisce un set di **permessi** (Permits). È utile per limitare il carico (Throttling) su una risorsa che può gestire più richieste contemporanee, ma non infinite.

Analogia: Immagina un parcheggio con 5 posti.

- Se arrivano 5 auto, entrano tutte (il contatore scende a 0).
- La 6ř auto si ferma alla sbarra (`acquire()`) e aspetta.
- Quando un'auto esce (`release()`), la sbarra si alza per chi aspetta.

```java
public class ConnectionPool {
    // Solo 5 thread possono accedere simultaneamente
    private final Semaphore semaphore = new Semaphore(5);

    public void accessDatabase() {
        try {
            // Prende un permesso. Se count == 0, si blocca qui.
            semaphore.acquire();

            System.out.println("Connessione acquisita: " + Thread.currentThread().
    ↪ getName());
            // Simulazione lavoro...
            Thread.sleep(100);

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            // Restituisce il permesso, svegliando eventuali thread in coda
            semaphore.release();
            System.out.println("Connessione rilasciata");
        }
    }
}
```

Listing 17.7: Semaphore per limitare connessioni DB

Deep Dive: Lock vs Semaphore

Una differenza sottile ma importante:

- Il **Lock** ha un proprietario (Owner): solo il thread che ha fatto `lock()` può fare `unlock()`.
- Il **Semaphore** non ha proprietario: il thread A può fare `acquire()` e il thread B può fare `release()` (utile in pattern produttore-consumatore particolari o death-detection).

17.5 Pattern di Coordinamento e Code

Uno dei pattern più diffusi nei sistemi concorrenti è il **Producer-Consumer** (Produttore-Consumatore). In passato, questo si implementava usando i metodi di basso livello `Object.wait()` e `Object.notify()`, ma questo approccio è difficile da mantenere e prono a errori (es. *Spurious Wakeups* o segnali persi).

Java offre soluzioni di alto livello nel pacchetto `java.util.concurrent`, in particolare l'interfaccia **BlockingQueue**.

17.5.1 BlockingQueue: La Coda Thread-Safe

Una `BlockingQueue` è una struttura dati che gestisce automaticamente la sincronizzazione tra thread. Ha due proprietà chiave che eliminano la necessità di controlli manuali:

- Se la coda è **Piena**, il thread Produttore che chiama `put()` viene bloccato (messo in attesa) finché non si libera uno spazio.
- Se la coda è **Vuota**, il thread Consumatore che chiama `take()` viene bloccato finché non arriva un nuovo elemento.

```
public class QueueDemo {
    // Coda con capacità fissa di 10 elementi
    private BlockingQueue<String> queue = new ArrayBlockingQueue<>(10);

    public void start() {
        // PRODUTTORE
        new Thread(() -> {
            try {
                for (int i = 0; i < 100; i++) {
                    String msg = "Messaggio " + i;
                    // Si blocca qui se la coda è piena (nessun dato perso)
                    queue.put(msg);
                    System.out.println("Prodotto: " + msg);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();

        // CONSUMATORE
        new Thread(() -> {
            try {
                while (true) {
                    // Si blocca qui se la coda è vuota (nessun busy-waiting)
                    String msg = queue.take();
                    System.out.println("Consumato: " + msg);
                    process(msg);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();
    }

    private void process(String msg) { /* Simulazione lavoro */ }
}
```

Listing 17.8: Producer-Consumer con ArrayBlockingQueue

17.5.2 Poison Pill Pattern (La Pillola Avvelenata)

Un problema comune con i consumatori è: *"Come faccio a fermare il loop `while(true)` in modo pulito?"*. Uccidere il thread con `stop()` è deprecato. Interromperlo mentre lavora potrebbe lasciare dati corrotti.

La soluzione elegante è il **Poison Pill Pattern**: inseriamo nella coda un oggetto speciale che significa "Fine delle trasmissioni". Quando il consumatore lo pesca, sa che deve terminare.

```
public class GracefulShutdownDemo {
    // Oggetto sentinella (o una costante stringa specifica)
    private static final String POISON_PILL = "###STOP###";
```

```
    private BlockingQueue<String> queue = new LinkedBlockingQueue<>();

    public void producer() {
        try {
            queue.put("Job A");
            queue.put("Job B");
            // ... fine del lavoro ...
            queue.put(POISON_PILL); // Invia segnale di stop
        } catch (InterruptedException e) { ... }
    }

    public void consumer() {
        try {
            while (true) {
                String msg = queue.take();

                // Controllo se è la pillola
                if (msg.equals(POISON_PILL)) {
                    System.out.println("Ricevuto segnale di stop. Chiusura...");
                    break; // Esce dal loop while in modo pulito
                }

                process(msg);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Listing 17.9: Poison Pill Implementation

Deep Dive: Multiple Consumers e Poison Pill

Se hai **più consumatori** (es. 5 thread che leggono dalla stessa coda), inserire una sola Poison Pill non basta: la leggerà solo il primo consumatore fortunato, che si spegnerà, mentre gli altri 4 rimarranno bloccati in eterno su `take()`.

Soluzione: Il produttore deve inserire **N Poison Pills** (una per ogni consumatore), oppure ogni consumatore, prima di morire, deve rimettere la pillola nella coda per il prossimo.

17.6 Gestione delle Risorse: Thread Pools ed Executors

Abbiamo visto come creare un thread con `new Thread().start()`. Tuttavia, in un'applicazione Enterprise, creare thread manualmente è considerato una **Bad Practice**.

17.6.1 Il costo di `new Thread()`

Riprendendo il concetto di System Call visto all'inizio:

- La creazione di un thread richiede l'allocazione di memoria per lo stack (circa 1MB).
- Richiede una chiamata al Kernel.

- Se arrivano 10.000 richieste e creiamo 10.000 thread, il server andrà in **OutOfMemoryError** o la CPU passerà il 100% del tempo a fare Context Switch invece che lavorare.

17.6.2 Il concetto di "Task" vs "Thread"

Java 5 ha introdotto l'**Executor Framework** per disaccoppiare "cosa deve essere fatto" da "chi lo fa".

- **Task (Il Lavoro):** È l'unità logica di lavoro. Rappresentato da `Runnable` (se non ritorna nulla) o `Callable<T>` (se ritorna un risultato).
- **Thread (Il Lavoratore):** È la risorsa fisica che esegue il task.
- **Thread Pool (Il Gestore):** È un gruppo di thread riutilizzabili che prelevano i task da una coda.

L'idea è: invece di creare un nuovo lavoratore per ogni mattone da spostare, assumiamo 10 lavoratori fissi e mettiamo i mattoni in una coda.

17.6.3 L'interfaccia ExecutorService

Invece di lavorare con la classe `Thread`, interagiamo con `ExecutorService`.

```
// Creo un pool con ESATTAMENTE 4 thread (es. numero di Core CPU)
ExecutorService executor = Executors.newFixedThreadPool(4);

for (int i = 0; i < 1000; i++) {
    int taskId = i;
    // Sottometto 1000 task.
    // Non vengono creati 1000 thread!
    // I 4 thread si scambiano i task man mano che finiscono.
    executor.submit(() -> {
        System.out.println("Task " + taskId + " eseguito da " +
                           Thread.currentThread().getName());
    });
}

// Importante: chiudere il pool alla fine
executor.shutdown();
```

Listing 17.10: Uso di un Thread Pool Fisso

17.6.4 Tipologie di Pool (Executors Factory)

La classe `Executors` offre vari metodi factory. È vitale conoscerne le differenze e i pericoli.

1. **FixedThreadPool(n):** Ha un numero fisso di thread. Se tutti sono occupati, i nuovi task finiscono in una `LinkedBlockingQueue` di attesa. *Uso:* Ideale per carichi prevedibili e CPU-bound.
2. **CachedThreadPool():** Non ha un limite massimo di thread. Se arrivano task e tutti i thread sono occupati, ne crea di nuovi. Se un thread sta fermo per 60 secondi, viene distrutto. *Uso:* Per task brevi e sporadici.
3. **SingleThreadExecutor():** Ha un solo thread. Garantisce che i task vengano eseguiti **sequenzialmente** (uno dopo l'altro).
4. **ScheduledThreadPool(n):** Sostituisce il vecchio `java.util.Timer`. Permette di eseguire task con ritardo o periodicamente (es. ogni 10 secondi).

Colloquio: Perché NON usare Executors.newCachedThreadPool() in produzione?

Domanda: "In un server ad alto traffico, qual è il rischio di usare `CachedThreadPool` o `FixedThreadPool`?"

Risposta:

- **CachedThreadPool:** Se il traffico esplode, crea thread all'infinito fino a saturare la RAM (`OutOfMemoryError`). Non ha freni.
- **FixedThreadPool:** Anche se i thread sono fissi, la **coda di attesa** interna è illimitata (`Integer.MAX_VALUE`). Se i task arrivano più velocemente di quanto vengano smaltiti, la coda riempie la Heap Memory fino al crash.

Soluzione Senior: In produzione non si usano i metodi factory di `Executors`. Si istanzia manualmente `ThreadPoolExecutor` configurando una coda con capacità limitata (Bounded Queue) e una *Rejection Policy* (cosa fare quando la coda è piena: scartare? lanciare eccezione?).

Deep Dive: Configurazione Manuale Sicura

Ecco come si crea un pool "a prova di bomba" per la produzione:

```
ExecutorService safePool = new ThreadPoolExecutor(
    4,  // Core Pool Size (Thread minimi)
    10, // Max Pool Size (Thread massimi sotto carico)
    60L, TimeUnit.SECONDS, // Tempo di vita dei thread extra
    new ArrayBlockingQueue<>(500), // CODA LIMITATA a 500 task
    new ThreadPoolExecutor.CallerRunsPolicy() // Se piena, esegue il chiamante
            //(rallenta il flusso)
);
```

17.6.5 I Metodi dell'ExecutorService: Submit e Shutdown

Una volta ottenuto un'istanza di `ExecutorService`, dobbiamo interagire con essa. L'interfaccia offre due categorie principali di metodi: quelli per sottomettere i task e quelli per gestire il ciclo di vita del pool.

1. Sottomissione dei Task

Esistono due modi per passare lavoro al pool. Scegliere quello giusto è importante per la gestione delle eccezioni.

- `void execute(Runnable command)`: Metodo "Fire-and-Forget". Si passa un task che non restituisce nulla. *Difetto:* Se il task lancia un'eccezione non controllata (RuntimeException), l'eccezione viene stampata sulla console e il thread potrebbe morire (venendo rimpiazzato dal pool), ma il chiamante non ha modo di saperlo o gestirlo.
- `Future<T> submit(Callable<T> task)`: Il metodo preferito. Restituisce un oggetto `Future` che rappresenta il risultato pendente. *Vantaggio:* Se il task lancia un'eccezione, questa viene "catturata" e conservata dentro il Future. Verrà rilanciata (come `ExecutionException`) solo quando chiamerai `future.get()`.

Colloquio: execute() vs submit()

Domanda: "Qual è la differenza fondamentale nella gestione degli errori tra `execute` e `submit`?"

Risposta:
- Con `execute()`, le eccezioni "scappano" dallo stack del thread e finiscono nell'`UncaughtExceptionHandler`. Se non ne hai configurato uno, vedi solo lo stacktrace nei log di sistema.
- Con `submit()`, l'eccezione è "ingabbiata" nel Future. Il task sembra finire con successo, ma l'errore esplode solo quando il thread principale chiede il risultato.

2. Spegnimento (Shutdown)

Un `ExecutorService` non è un oggetto normale: contiene thread attivi. Se il tuo programma Java finisce il 'main' ma non spegni il pool, la JVM **non si chiuderà mai** (perché i thread del pool sono User Threads, non Daemon).

Ecco la "Trinità dello Shutdown":
- `void shutdown()`: Lo spegnimento "gentile". Il pool smette di accettare *nuovi* task, ma continua a lavorare per finire quelli già in coda. È asincrono (ritorna subito).
- `List<Runnable> shutdownNow()`: Lo spegnimento "brutale". Tenta di fermare i thread attivi (usando `interrupt()`) e restituisce una lista dei task che erano in coda e non sono mai partiti.
- `boolean awaitTermination(long timeout, TimeUnit unit)`: Metodo bloccante. Il thread corrente aspetta che il pool sia effettivamente spento (tutti i task finiti). Restituisce `true` se il pool si è spento, `false` se è scaduto il timeout.

```
ExecutorService pool = Executors.newFixedThreadPool(10);

// ... utilizzo del pool ...

// 1. Smetti di accettare nuovi ordini
pool.shutdown();

try {
    // 2. Aspetta un tempo ragionevole (es. 60s) che finiscano i lavori in corso
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {

        // 3. Tempo scaduto! Forza la chiusura
        pool.shutdownNow();

        // 4. Aspetta ancora un attimo che i thread rispondano all'interrupt
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Il Pool non vuole terminare!");
    }
} catch (InterruptedException ie) {
    // 5. (Opzionale) Se il thread che sta facendo lo shutdown viene interrotto
    pool.shutdownNow();
    Thread.currentThread().interrupt();
}
```

Listing 17.11: Il Pattern "Graceful Shutdown" (Da imparare a memoria)

17.7 Gestione Asincrona: Future vs CompletableFuture

Fino a Java 5, il multithreading si basava sul lancio di `Runnable`, che però avevano un grosso limite: il metodo `run()` è `void`. Non restituisce alcun valore. Se volevamo il risultato di un

calcolo parallelo, dovevamo inventarci meccanismi complessi con variabili condivise.

17.7.1 L'interfaccia Future (Java 5)

Java 5 ha introdotto `Callable<T>` (che restituisce un valore) e `Future<T>`, un "placeholder" per un risultato che arriverà in futuro.

```
ExecutorService executor = Executors.newFixedThreadPool(2);

// Sottomettiamo un task che richiede tempo (es. 2 secondi)
Future<Integer> future = executor.submit(() -> {
    Thread.sleep(2000);
    return 42;
});

System.out.println("Faccio altro nel frattempo...");

try {
    // IL PROBLEMA E' QUI:
    // get() è BLOCCANTE. Il main thread si ferma qui e aspetta.
    // Abbiamo perso il vantaggio dell'asincronia nel momento in cui ci serviva il
    ↪ dato.
    Integer result = future.get();
    System.out.println("Risultato: " + result);
} catch (Exception e) { e.printStackTrace(); }
```

Listing 17.12: Il vecchio approccio con Future

17.7.2 CompletableFuture (Java 8): La Rivoluzione

Il limite di `Future` è che non puoi dire: *"Quando hai finito, fai questo, poi questo, poi quest'altro"*. Devi per forza bloccarti con `get()` per orchestrare i passi successivi.

Java 8 introduce **CompletableFuture**, che implementa il pattern delle **Promise**. Permette di costruire una **pipeline** di operazioni che verranno eseguite in cascata, senza mai bloccare il thread principale.

```
// 1. Avvia un task asincrono (in un thread del ForkJoinPool)
CompletableFuture.supplyAsync(() -> fetchUserId("alessandro"))

    // 2. Quando l'ID arriva, usalo per scaricare gli ordini (Chain asincrona)
    .thenCompose(userId -> fetchOrders(userId))

    // 3. Quando gli ordini arrivano, calcola il totale (Trasformazione sincrona)
    .thenApply(orders -> calculateTotal(orders))

    // 4. Alla fine, stampa il risultato o gestisci l'errore
    .thenAccept(total -> System.out.println("Totale speso: " + total))

    .exceptionally(ex -> {
        System.err.println("Qualcosa è andato storto: " + ex.getMessage());
        return null;
    });

// Il main thread prosegue immediatamente, senza aspettare i risultati!
System.out.println("Non mi sono bloccato!");
```

Listing 17.13: Pipeline Asincrona con CompletableFuture

Deep Dive: thenApply vs thenCompose

È la domanda da "100 punti" sui CompletableFuture (simile a `map` vs `flatMap` negli Stream).

- **thenApply (Map):** Prende il valore precedente e lo trasforma. *Input: String* → *Output: Int.* Usa questo se la funzione di trasformazione è veloce e sincrona.
- **thenCompose (FlatMap):** Prende il valore precedente e restituisce un **nuovo CompletableFuture**. *Input: String* → *Output: CompletableFuture<Int>*. Usa questo se lo step successivo è a sua volta un'operazione asincrona (es. chiamare un altro microservizio), per evitare di avere `Future<Future<Int»` annidati.

Colloquio: Combinare più Future

Domanda: "Devo chiamare 3 servizi REST in parallelo e aspettare che abbiano finito tutti. Come faccio?"

Risposta: Si usa `CompletableFuture.allOf()`.

```
var f1 = CompletableFuture.supplyAsync(() -> service1.call());
var f2 = CompletableFuture.supplyAsync(() -> service2.call());
var f3 = CompletableFuture.supplyAsync(() -> service3.call());

// Crea un nuovo Future che si completa quando TUTTI e 3 sono finiti
CompletableFuture<Void> allDone = CompletableFuture.allOf(f1, f2, f3);

// Join finale (qui blocchiamo solo alla fine per raccogliere i dati)
allDone.join();
```

17.8 Java 21: Virtual Threads (Project Loom)

Nelle sezioni precedenti abbiamo stabilito due verità dolorose:

1. I Thread del Sistema Operativo sono costosi (1-2 MB di RAM, Context Switch lento).
2. Per sopravvivere, dobbiamo usare i Thread Pool per limitarne il numero.

Questo approccio ha funzionato per anni, ma ha un limite: il modello *Thread-per-Request* non scala. Se hai un pool di 200 thread e arrivano 201 richieste che fanno I/O (es. query lenta al DB), la 201esima richiesta deve aspettare in coda, anche se la CPU è scarica.

Con Java 21 (LTS), Project Loom introduce i **Virtual Threads**, cambiando per sempre le regole del gioco.

17.8.1 Platform Threads vs Virtual Threads

Per capire la rivoluzione, confrontiamo il vecchio e il nuovo:

- **Platform Thread (Il Vecchio):** È un wrapper 1:1 su un thread del Sistema Operativo.
  - Gestito dal Kernel.
  - Stack fisso e grande.
  - Costoso da creare e bloccare.
- **Virtual Thread (Il Nuovo):** È un'entità interamente gestita dalla JVM (User-mode thread).

- Non ha un legame 1:1 con l'OS.
- Stack dinamico che parte da pochi byte (nell'Heap).
- Possiamo crearne milioni senza finire la memoria.

17.8.2 Architettura M:N (Mounting e Unmounting)

Come fanno milioni di Virtual Threads a girare su, ad esempio, 8 Core fisici? La JVM introduce un livello di indirezione.

1. **Carrier Thread:** È il thread fisico (Platform Thread) che funge da "motore". Ce ne sono pochi (solitamente pari al numero di core CPU).
2. **Mounting:** Quando un Virtual Thread deve eseguire calcoli (CPU Bound), la JVM lo "monta" su un Carrier Thread temporaneo.
3. **Unmounting (La Magia):** Quando il Virtual Thread chiama un'operazione bloccante (es. `Thread.sleep()`, `socket.read()`, query JDBC), la JVM **non blocca** il Carrier Thread.
   - Salva lo stack del Virtual Thread nell'Heap ("congela" lo stato).
   - Stacca il Virtual Thread dal Carrier.
   - Il Carrier è ora libero di eseguire un *altro* Virtual Thread.
4. **Resuming:** Quando l'operazione di I/O finisce (l'OS notifica la JVM), il Virtual Thread viene riattivato e montato sul primo Carrier disponibile.

Deep Dive: System Calls "Truccate"

Nella Sezione 1 abbiamo detto che le System Call sono bloccanti. Come fa Loom a evitarlo?

Gli ingegneri di Java hanno riscritto quasi tutte le librerie standard (`java.io`, `java.net`, `java.util.concurrent`). Quando tu scrivi codice bloccante:

```
// Sembra bloccante...
var response = httpClient.send(request);
```

Sotto il cofano, la JVM non chiama la System Call bloccante `read()`. Chiama invece una modalità I/O asincrona del sistema operativo (come `epoll` su Linux o `kqueue` su Mac) e "parcheggia" il Virtual Thread, liberando immediatamente il Carrier.

Risultato: Scrivi codice sincrono, semplice e leggibile, ma ottieni le performance del codice asincrono non bloccante.

17.8.3 Addio Thread Pools?

Con i Virtual Threads, la gestione delle risorse cambia radicalmente.

Colloquio: Devo usare un Thread Pool per i Virtual Threads?

Domanda: "Ho migrato a Java 21. Come configuro il pool per i Virtual Threads?"

Risposta: Non lo fai. Il pooling serve a razionare risorse costose. I Virtual Threads sono economici ("cheap"). Non si riciclano i Virtual Threads; se ne crea uno nuovo per ogni task e lo si lascia morire alla fine.

Invece di `newFixedThreadPool`, si usa il nuovo esecutore: `Executors.newVirtualThreadPerTaskExecutor()`.

```
1 // Esempio pratico: simuliamo 100.000 richieste concorrenti
```

```
// Con i Platform Threads, questo manderebbe in crash la JVM (OOM).

try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 100_000).forEach(i -> {
        executor.submit(() -> {
            // Simula operazione I/O (es. chiamata HTTP) di 1 sec
            Thread.sleep(1000);
            return i;
        });
    });
}
// Tempo totale esecuzione: ~1 secondo (perché corrono tutti insieme!)
```

Listing 17.14: Il server da 1 milione di thread

17.8.4 Quando NON usare i Virtual Threads

I Virtual Threads non sono la panacea per tutto.

- **I/O Bound (Database, Network): Perfetti.** Qui i Virtual Threads brillano perché passano il tempo ad aspettare, lasciando libero il Carrier.
- **CPU Bound (Calcoli intensivi, Rendering video, Hashing): Inutili (o dannosi).** Se un thread deve macinare numeri per 10 minuti senza mai fermarsi, occuperà il Carrier Thread per 10 minuti. Lo scheduling dei Virtual Threads non fa "preemption" temporale. Per task CPU intensive, i cari vecchi Platform Threads (o `ForkJoinPool`) sono ancora la scelta migliore.

Capitolo 18

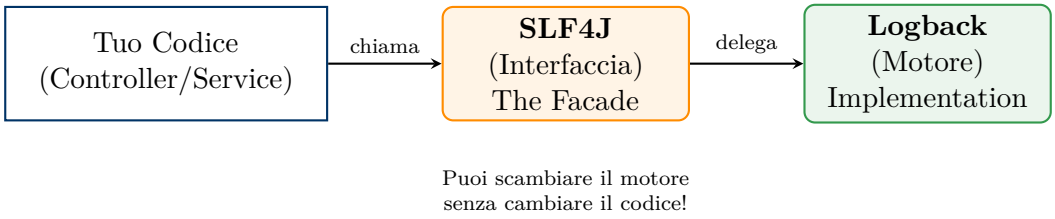
Logging e Observability

"I log sono la scatola nera del tuo aereo. Se l'applicazione si schianta, sono l'unica cosa che ti resta per capire perché."

Molti Junior usano `System.out.println()`. Un Senior sa che in produzione la console non esiste (o è un buco nero ingestibile). In Java, il logging è maturo e strutturato secondo un pattern architetturale preciso: il **Facade Pattern**.

18.1 L'Architettura: SLF4J e Logback

In Spring Boot non parliamo quasi mai direttamente con la libreria di logging. Usiamo un'interfaccia.



- **SLF4J (Simple Logging Facade for Java):** È l'interfaccia standard. Tu usi `Logger.info()`.
- **Logback:** È l'implementazione di default di Spring Boot. È il motore che scrive fisicamente su file o console.
- **Log4j2:** Un'altra implementazione potente (usata per alte performance async).

18.2 I Livelli di Log (Hierarchy)

Non tutti i messaggi sono uguali. Configurare i livelli correttamente salva lo spazio su disco e la sanità mentale.

Livello	Significato	Ambiente
ERROR	Qualcosa si è rotto e l'operazione è fallita. Richiede intervento.	PROD
WARN	Qualcosa non va (es. disco quasi pieno, retry connessione), ma l'app funziona ancora.	PROD
INFO	Eventi di business significativi (es. "Utente X creato", "Ordine Y pagato").	PROD
DEBUG	Dettagli tecnici per sviluppatori (es. "Query SQL eseguita", "Parametri input").	DEV/TEST
TRACE	Dettaglio atomico (es. inizio/fine metodo, valori variabili intermedie). Verbosissimo.	DEV

18.3 Anatomia di un Logger: Gerarchia e Istanziazione

Quando scrivi la classica riga di inizializzazione del logger, stai definendo la posizione della tua classe all'interno di una struttura ad albero.

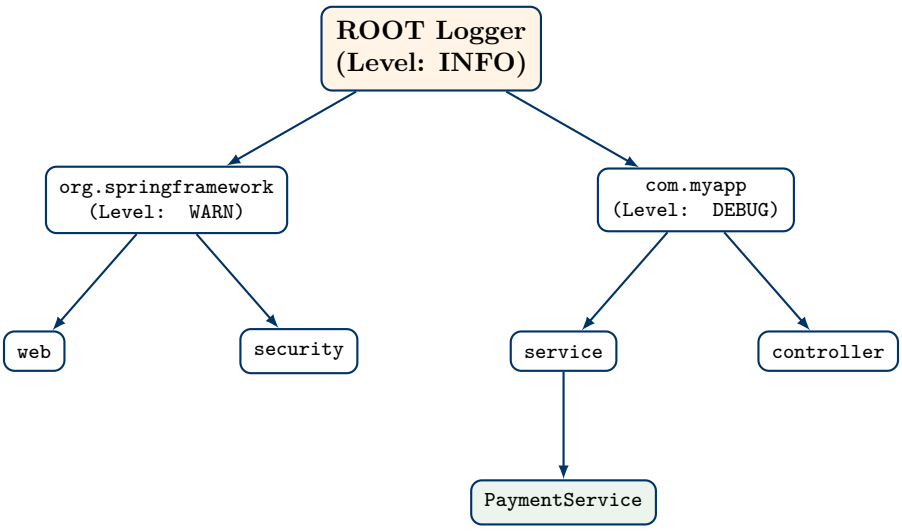
```
public class PaymentService {
    // Perché static? Perché final?
    private static final Logger log = LoggerFactory.getLogger(PaymentService.class);
}
```

Analizziamo questa riga parola per parola:

- **private:** Il logger serve solo a questa classe.
- **static:** Esiste una sola istanza del logger per *tutti* gli oggetti `PaymentService`. Risparmia memoria (non ricrei il logger per ogni transazione).
- **final:** Il riferimento non cambia mai.
- **getLogger(PaymentService.class):** Qui avviene la magia. Stai dicendo alla Factory: *"Dammi il logger associato al namespace* `com.azienda.app.service.PaymentService`*"*.

18.3.1 La Gerarchia ad Albero (Logger Tree)

I logger non sono entità isolate. Sono organizzati gerarchicamente in base al package java (il nome della classe). Esiste un logger speciale, chiamato **ROOT**, che è il padre di tutti.



18.3.2 Ereditarietà della Configurazione

Questa struttura ad albero spiega come funziona il file `application.properties`. Quando configuri un livello di log, questo si propaga a cascata ("bubbling down") a tutti i figli, a meno che un figlio non lo sovrascriva.

```
# 1. Configurazione Globale (ROOT)
logging.level.root=INFO

# 2. Configurazione Specifica (Sovrascrittura)
# Voglio vedere tutto quello che succede nel mio codice...
logging.level.com.myapp=DEBUG

# 3. ...ma le librerie di Spring devono stare zitte (solo errori gravi)
logging.level.org.springframework=WARN
```

application.properties

Deep Dive: Lombok @Slf4j: Cosa genera?

Quando usi l'annotazione `@Slf4j`, Lombok genera esattamente la riga statica che abbiamo visto sopra, usando il nome della classe in cui ti trovi.

```
// Lombok genera questo a compile time:
private static final org.slf4j.Logger log =
    org.slf4j.LoggerFactory.getLogger(NomeDellaTuaClasse.class);
```

Inoltre, se rinomini la classe (Refactoring), Lombok aggiorna automaticamente il logger, mentre se lo scrivi a mano spesso ti dimentichi di cambiare il `.class` nel `getLogger`, creando confusione nei log.

18.4 Come scrivere i Log: Performance e Placeholder

18.4.1 L'errore della Concatenazione

```
// BAD: String Concatenation
// Anche se il livello è impostato a ERROR, Java DEVE comunque
// concatenare queste stringhe (costoso) PRIMA di entrare nel metodo debug.
log.debug("L'utente " + user.getName() + " ha fatto login da " + ip);
```

18.4.2 La Soluzione: I Placeholder {}

SLF4J supporta i placeholder. La concatenazione avviene **solo se** il livello di log è abilitato.

```
// GOOD: Placeholder
// Se il livello è INFO, questa riga costa quasi zero.
log.debug("L'utente {} ha fatto login da {}", user.getName(), ip);
```

Deep Dive: Lombok @Slf4j

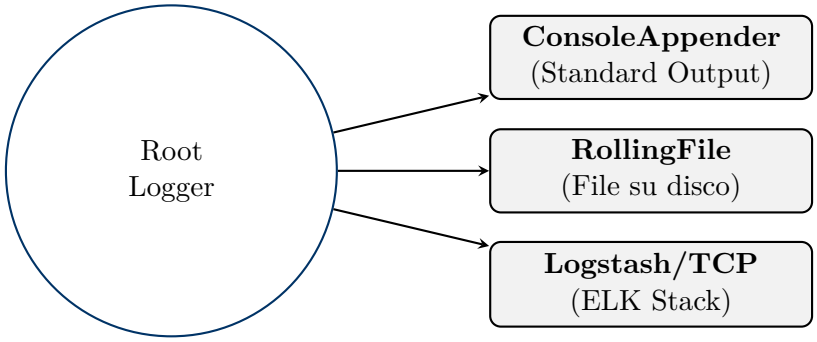
Non istanziare mai il logger a mano in ogni classe. Usa Lombok.

```
@Service
@Slf4j // Genera: private static final Logger log = LoggerFactory.getLogger(...)
```

```
public class UserService {
    public void doSomething() {
        log.info("Sto facendo qualcosa...");
    }
}
```

18.5 Appenders: Dove vanno i log?

Logback usa il concetto di **Appender** per decidere la destinazione. Un Logger può avere più Appender contemporaneamente.



18.5.1 Rolling Policy (Rotazione)

Non scrivere mai su un file unico infinito (`app.log`). Usa una **RollingPolicy**.

- **TimeBased:** Crea un file nuovo ogni giorno (`app-2023-10-01.log`).
- **SizeBased:** Crea un file nuovo ogni 10MB.

18.6 MDC: Tracciare le richieste nel caos

In un'app web multithread, i log di 10 utenti si mischiano. Come fai a filtrare solo i log relativi alla richiesta dell'utente Mario?

L'**MDC (Mapped Diagnostic Context)** è una mappa Thread-Local (simile a una `Map<String, String>`) che "viaggia" insieme al thread corrente.

```
// All'inizio della richiesta (es. in un Filter)
MDC.put("userId", "user-123");
MDC.put("transactionId", UUID.randomUUID().toString());

// ... nel Service (senza passare parametri!) ...
log.info("Pagamento avviato");
// Output automatico:
// [INFO] [userId:user-123] [tx:abc-999] Pagamento avviato

// Alla fine (nel finally)
MDC.clear(); // Fondamentale pulire il thread!
```

Uso di MDC in un Interceptor

18.7 Best Practices di Sicurezza

Colloquio: Cosa NON loggare mai?

Domanda: Quali dati sono vietati nei log (GDPR/Security)?

Risposta:

- **Password:** Mai, nemmeno hashate.
- **Dati Carte di Credito (PAN):** Assolutamente vietato (PCI-DSS).
- **Token di Sessione/JWT:** Se li logghi, chi legge i log può rubare l'identità.
- **Dati PII (Personally Identifiable Information):** Codici fiscali, indirizzi, se non strettamente necessario e protetto.

Parte IV

Ingegneria del Software e Design

Capitolo 19

Principi SOLID e Clean Code

I principi SOLID (acronimo introdotto da Robert C. Martin, "Uncle Bob") non sono regole rigide, ma linee guida diagnostiche per gestire il "debito tecnico". Un codice che viola i SOLID tende a diventare rigido (difficile da cambiare) e fragile (si rompe in punti inaspettati).

19.1 S - Single Responsibility Principle (SRP)

Colloquio: Definizione Reale di SRP

Molti dicono: "Una classe deve fare una sola cosa". È vago. La definizione corretta è: **"Una classe deve avere una sola ragione per cambiare"**. Se devi modificare la stessa classe sia perché è cambiato il database, sia perché è cambiata la formula di calcolo delle tasse, stai violando l'SRP.

Violazione (Bad Design):

```
public class Fattura {
    public float calcolaTotale() { ... } // Logica di Business
    public void stampaPDF() { ... }      // Logica di Presentazione
    public void salvaSuDB() { ... }      // Logica di Persistenza
}
```

Questa classe è un "God Object". Se cambia la libreria PDF, tocchi la classe. Se cambia SQL in Mongo, tocchi la stessa classe.

Soluzione (Refactoring): Separare le responsabilità in classi dedicate: `FatturaCalculator`, `FatturaPrinter`, `FatturaRepository`.

19.2 O - Open/Closed Principle (OCP)

"Le entità software devono essere **aperte all'estensione**, ma **chiuse alla modifica**". Dovresti poter aggiungere nuove funzionalità senza toccare il codice sorgente esistente (e già testato).

Violazione (Bad Design): Usare `switch` o catene di `if-else` per determinare il comportamento.

```
public double calcolaSconto(String tipoUtente) {
    if (tipoUtente.equals("Standard")) return 0.10;
    else if (tipoUtente.equals("VIP")) return 0.20;
    // Se aggiungo "SuperVIP", devo modificare questo file!
    return 0;
```

```
}
```

Soluzione (Refactoring): Usare il Polimorfismo (Interfacce).

```
interface ScontoStrategy { double applica(); }

class ScontoVIP implements ScontoStrategy { ... }
class ScontoStandard implements ScontoStrategy { ... }

// Il codice che usa la strategia non cambia se ne aggiungi una nuova
public double calcola(ScontoStrategy strategia) {
    return strategia.applica();
}
```

19.3 L - Liskov Substitution Principle (LSP)

"Gli oggetti di una superclasse devono poter essere sostituiti con oggetti delle sottoclassi senza rompere l'applicazione".

In pratica: Una sottoclasse non deve mai restringere il comportamento del padre o lanciare eccezioni impreviste per metodi che il padre supporta.

Deep Dive: Il classico esempio del Quadrato/Rettangolo

Matematicamente un Quadrato è un Rettangolo. In OOP, questa eredità è spesso sbagliata. Se hai un metodo `resize(Rettangolo r)` che raddoppia la larghezza, e gli passi un `Quadrato`, potresti rompere la logica perché il Quadrato forza altezza = larghezza. **Sintomo di violazione:** Se nel codice devi fare `if (obj instanceof Sottoclasse)`, probabilmente stai violando LSP.

19.4 I - Interface Segregation Principle (ISP)

"I client non devono essere forzati a dipendere da interfacce che non usano". Meglio avere tante interfacce piccole e specifiche (*Role Interfaces*) piuttosto che un'unica interfaccia gigante (*Header Interface*).

Violazione:

```
interface Worker {
    void lavora();
    void mangia();
}

class Robot implements Worker {
    public void lavora() { ... }
    public void mangia() {
        // I robot non mangiano! Violazione.
        throw new UnsupportedOperationException();
    }
}
```

Soluzione: Dividere in `Workable` e `Eatable`. Il Robot implementa solo `Workable`.

19.5 D - Dependency Inversion Principle (DIP)

Questo è il principio che abilita framework come Spring.

1. I moduli di alto livello (Business Logic) non devono dipendere da moduli di basso livello (Dettagli tecnici, es. MySQL, FileSystem). Entrambi devono dipendere da **Astrazioni** (Interfacce).
2. Le astrazioni non devono dipendere dai dettagli. I dettagli devono dipendere dalle astrazioni.

Violazione (Tight Coupling):

```
class ServizioVendite {
    // Dipendenza diretta dalla classe concreta MySQL
    private MySQLDatabase db = new MySQLDatabase();

    void vendi() { db.save(); }
}
```

Soluzione (Dependency Injection):

```
class ServizioVendite {
    private Database db; // Dipende dall'interfaccia (Astrazione)

    // Qualcuno da fuori iniettera' l'implementazione concreta
    public ServizioVendite(Database db) {
        this.db = db;
    }
}
```

19.6 Bonus: DRY, KISS e YAGNI

- **DRY (Don't Repeat Yourself):** Ogni pezzo di conoscenza/logica deve avere una rappresentazione unica. Se copi-incolli codice, stai creando debito tecnico.
- **KISS (Keep It Simple, Stupid):** La soluzione più semplice è quasi sempre la migliore. Evita l'over-engineering.
- **YAGNI (You Ain't Gonna Need It):** Non scrivere codice per funzionalità che "potrebbero servire in futuro". Scrivi solo ciò che serve oggi. (Principio cardine dell'Extreme Programming).

Capitolo 20

Il Pattern Singleton (Creational)

Iniziamo dal pattern più famoso, più utilizzato e paradossalmente più frainteso dell'ingegneria del software. Il **Singleton** ha uno scopo semplice: garantire che una classe abbia **una e una sola istanza** e fornire un punto di accesso globale ad essa.

Nel mondo moderno, l'abuso del Singleton è considerato un anti-pattern, ma capirlo è fondamentale per comprendere come funzionano i framework come Spring.

20.1 Il Problema

In molte applicazioni, ci sono componenti che devono essere unici per natura:

- Un **Connection Pool** verso il Database (non vuoi aprirne 50 diversi).
- Un **Configuration Manager** che legge le proprietà all'avvio.
- Un **Logger** che scrive su un file condiviso.

Se istanziassimo questi oggetti ogni volta (`new Database()`), avremmo spreco di memoria, corruzione dei dati o errori di connessione.

20.2 Implementazione "Senior": Thread Safety

Un Junior implementa il Singleton in modo "Lazy" (crealo solo se serve), ma dimentica che le applicazioni web sono **Multi-Thread**.

```
public class DatabaseConnection {
    private static DatabaseConnection instance;

    // 1. Costruttore privato: Nessuno può fare 'new' da fuori
    private DatabaseConnection() {}

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            // RACE CONDITION:
            // Se due thread arrivano qui contemporaneamente,
            // entrambi creano una nuova istanza!
            instance = new DatabaseConnection();
        }
        return instance;
    }
}
```

Approccio Naive (Non Thread-Safe)

20.2.1 La Soluzione: Double-Checked Locking

Per renderlo sicuro, potremmo mettere `synchronized` sul metodo, ma rallenteremmo tutte le letture future. L'approccio professionale è il **Double-Checked Locking** con la keyword `volatile`.

```
public class DatabaseConnection {

    // 'volatile' garantisce che le modifiche siano visibili
    // immediatamente a tutti i thread (niente cache CPU locale)
    private static volatile DatabaseConnection instance;

    private DatabaseConnection() {}

    public static DatabaseConnection getInstance() {
        // Primo check (senza lock, veloce)
        if (instance == null) {

            // Acquisisco il lock solo se necessario
            synchronized (DatabaseConnection.class) {

                // Secondo check (dentro il lock)
                // Necessario perché un altro thread potrebbe aver creato
                // l'istanza mentre eravamo in attesa del lock.
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }
}
```

Thread-Safe Performance Singleton

20.3 La Soluzione "Effective Java" (Enum)

Joshua Bloch (autore di Java Core) suggerisce che il modo migliore per implementare un Singleton in Java è usare un **Enum**.

```
public enum ConfigurationManager {
    INSTANCE; // L'unica istanza possibile

    private Properties props;

    // Costruttore (eseguito una volta sola dalla JVM)
    ConfigurationManager() {
        props = new Properties();
        // load from file...
    }

    public String getProperty(String key) {
        return props.getProperty(key);
    }
}
```

```
16
17 // Utilizzo
18 String dbUrl = ConfigurationManager.INSTANCE.getProperty("db.url");
```

Deep Dive: Perché l'Enum è superiore?

L'approccio Enum risolve gratis due problemi complessi:

1. **Reflection Attack:** Con il Singleton classico, un hacker potrebbe usare `setAccessible(true)` sul costruttore privato e creare una seconda istanza. Con gli Enum, la JVM impedisce la creazione manuale.
2. **Serialization:** Se serializzi e deserializzi un Singleton classico, ottieni una nuova istanza (copia). Con gli Enum, Java garantisce che rimanga la stessa istanza.

20.4 Spring Singleton vs GoF Singleton

Questa è la domanda che separa chi conosce i Pattern da chi conosce Spring.

Colloquio: Spring Bean Singleton

Domanda: I Bean di Spring sono Singleton di default. Sono la stessa cosa del Singleton Pattern?

Risposta: No, concettualmente è diverso.

- **GoF Singleton (Design Pattern):** Garantisce una istanza per **ClassLoader** (in pratica, una per JVM). È hardcoded nella classe (static).
- **Spring Singleton (Scope):** Garantisce una istanza per **Container (ApplicationContext)**.

Conseguenza: Se avvio due ApplicationContext nella stessa JVM, avrò due istanze del Bean "Singleton". In Spring, il Singleton è una regola di gestione del ciclo di vita, non una caratteristica strutturale della classe.

20.5 Perché è considerato un Anti-Pattern?

Oggi si tende a evitare il Singleton manuale (quello statico).

20.5.1 Il problema del Testing

Il Singleton introduce uno stato globale nascosto e un accoppiamento forte.

```
1 public class UserService {
2     public void register(User u) {
3         // Accoppiamento forte! Non posso sostituirlo con un Mock.
4         // Se DatabaseConnection prova a connettersi al DB vero,
5         // il mio Unit Test fallisce senza rete.
6         DatabaseConnection.getInstance().save(u);
7     }
8 }
```

Soluzione: Usare la **Dependency Injection** (Spring). Invece di chiamare `getInstance()`, chiedi l'istanza nel costruttore. Spring ti passerà l'unica istanza (Singleton Scope), ma nei test potrai passare un Mock.

20.6 Riepilogo

Concetto	Best Practice
Implementazione Manuale	Usa **Enum** se possibile, o **Double-Checked Locking** con `volatile`.
Uso moderno	Evita i Singleton statici. Usa la **Dependency Injection** di Spring con scope `@Scope("singleton")`.
Visibilità	Il costruttore deve essere sempre **private**.
Testing	I Singleton statici rendono il mocking quasi impossibile. Preferire sempre l'iniezione.

Capitolo 21

Il Pattern Builder (Creational)

Hai mai visto un costruttore con 10 parametri, di cui 7 sono `null`? Questo è il segnale inequivocabile che ti serve il **Builder Pattern**.

Il Builder separa la costruzione di un oggetto complesso dalla sua rappresentazione, permettendo di creare lo stesso oggetto passo dopo passo. In Java, è diventato lo standard de-facto per la creazione di DTO, Configurazioni e Entità immutabili.

21.1 Il Problema: Telescoping Constructors

Immagina di dover istanziare una classe `Pizza`.

```
public class Pizza {
    public Pizza(int size) { ... }
    public Pizza(int size, boolean cheese) { ... }
    public Pizza(int size, boolean cheese, boolean pepperoni) { ... }
    public Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon) { ... }
}

// Utilizzo: Cosa significano questi boolean? È facile scambiarli!
Pizza p = new Pizza(12, true, false, true);
```

L'Anti-Pattern

Questo codice è fragile. Se aggiungi un ingrediente, devi rifare i costruttori. Se scambi il secondo `true` col terzo, il compilatore non ti avvisa, ma la pizza è sbagliata.

21.2 L'Implementazione Classica (Effective Java)

Prima di Lombok, il Builder si scriveva a mano usando una **Static Inner Class**. È importante saperlo scrivere ai colloqui per dimostrare di capire cosa succede "sotto il cofano".

```
public class User {
    // 1. Campi final (Immutabilità)
    private final String firstName; // Required
    private final String lastName;  // Required
    private final int age;          // Optional
    private final String email;     // Optional

    // 2. Costruttore Privato: accetta solo il Builder
    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
```

```
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.email = builder.email;
    }

    // 3. Static Inner Class
    public static class UserBuilder {
        private final String firstName;
        private final String lastName;
        private int age = 0; // Default
        private String email = null;

        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        // 4. Fluent Interface (Ritorna 'this')
        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }

        public UserBuilder email(String email) {
            this.email = email;
            return this;
        }

        // 5. Metodo finale di build
        public User build() {
            // Qui puoi fare validazioni complesse prima di creare l'oggetto
            if (age < 0) throw new IllegalStateException("Age cannot be negative");
            return new User(this);
        }
    }
}
```

Builder Manuale

```
User u = new User.UserBuilder("Mario", "Rossi")
    .age(30)
    .email("mario@test.com")
    .build();
```

Utilizzo Fluente

21.3 L'Era Moderna: Lombok @Builder

Scrivere tutto quel codice boilerplate è noioso. Nelle aziende moderne si usa **Project Lombok**.

```
@Builder
@Getter
@ToString
public class UserDTO {
    private String name;
```

```
    private String surname;

    @Builder.Default // Importante! Altrimenti il builder ignora il valore
    private boolean active = true;
}
```

L'annotazione `@Builder` genera automaticamente la classe statica interna, i metodi setter fluenti e il metodo build.

Deep Dive: Il Potere di toBuilder

Se hai un oggetto immutabile e vuoi crearne una copia modificando solo un campo, Lombok offre una feature potente: `@Builder(toBuilder = true)`.

```
UserDTO user = ...; // Oggetto esistente
// Crea una copia identica ma cambia solo l'email
UserDTO updated = user.toBuilder()
    .email("new@email.com")
    .build();
```

Questo è fondamentale nella programmazione funzionale e reattiva dove gli oggetti non si modificano mai.

21.4 Builder vs JavaBeans (Setters)

Perché non usare semplicemente un costruttore vuoto e poi chiamare `setAge(10)`, `setEmail(...)`?

Colloquio: Immutabilità e Thread Safety

Domanda: Qual è il vantaggio principale del Builder rispetto ai Setter?

Risposta: L'**Immutabilità**.

- Con i **Setter (JavaBeans)**, l'oggetto può trovarsi in uno stato inconsistente (es. hai settato il nome ma non ancora il cognome). Inoltre, l'oggetto è mutabile, quindi non Thread-Safe.
- Con il **Builder**, l'oggetto viene restituito dal metodo `build()` solo quando è completo. L'oggetto risultante può non avere setter (tutti i campi `final`), rendendolo intrinsecamente Thread-Safe e sicuro da passare tra i layer.

21.5 Real World Examples in Spring

Non usi il Builder solo per le tue classi. Spring è pieno di Builder.

21.5.1 1. UriComponentsBuilder (Costruzione URL)

Invece di concatenare stringhe per creare URL (rischiando errori di encoding), Spring offre questo builder.

```
String url = UriComponentsBuilder.fromHttpUrl("https://api.google.com")
    .path("/search")
    .queryParam("q", "Spring Boot")
    .queryParam("lang", "it")
    .build()
```

```
    .toUriString();
// Risultato: https://api.google.com/search?q=Spring\%20Boot&lang=it
```

21.5.2 2. ResponseEntity (Rest Controller)

Quando rispondi da un Controller, usi un Builder fluente.

```
return ResponseEntity
    .status(HttpStatus.CREATED)
    .header("X-Custom-Header", "Value")
    .body(myDto);
```

21.5.3 3. Spring Security (HttpSecurity)

La configurazione della sicurezza è una catena di builder infinita.

```
http
    .authorizeRequests()
        .antMatchers("/public").permitAll()
        .anyRequest().authenticated()
    .and() // Torna al builder padre
    .formLogin();
```

21.6 Riepilogo

Caratteristica	Builder Pattern
Quando usarlo	Quando un oggetto ha più di 3-4 parametri, specialmente se opzionali.
Vantaggio Key	Codice leggibile (sai cosa stai settando) e possibilità di creare oggetti immutabili.
Best Practice	Usa **Lombok** `@Builder` per DTO ed Entity. Usa `@Builder.Default` per valori di default.
Attenzione	Non abusarne per oggetti semplici con 2 campi. Lì il costruttore `AllArgs` è più veloce.

Capitolo 22

Il Pattern Factory Method (Creational)

"New is Glue". Ogni volta che usi la keyword `new` nel tuo codice di business, stai incollando (accoppiando) la tua classe a un'implementazione specifica.

Il **Factory Method** serve a delegare la creazione degli oggetti a un componente specializzato, nascondendo la logica di istanziazione al client. In Java Moderno e Spring, questo pattern è ovunque, spesso mascherato da metodi statici.

22.1 Il Problema: Accoppiamento Rigido

Immagina un sistema di reportistica.

```
public class ReportService {
    public void generateReport(String type) {
        Report report;
        if (type.equals("PDF")) {
            report = new PdfReport(); // Accoppiamento forte!
        } else if (type.equals("CSV")) {
            report = new CsvReport(); // Accoppiamento forte!
        }
        report.render();
    }
}
```

Approccio Senza Factory (Bad)

Se domani aggiungi "Excel", devi modificare il `ReportService`. Questo viola l'**Open/Closed Principle** (aperto all'estensione, chiuso alla modifica).

22.2 L'Implementazione Classica (Polimorfismo)

La definizione GoF (Gang of Four) prevede di usare l'ereditarietà: una classe astratta definisce il metodo `create()`, e le sottoclassi decidono cosa istanziare.

Tuttavia, nel Java moderno, si preferisce spesso la variante **Simple Factory** o **Static Factory Method**.

```
public class ReportFactory {

    // Metodo statico: il client non deve istanziare la Factory
```

```
    public static Report create(String type) {
        return switch (type) {
            case "PDF" -> new PdfReport();
            case "CSV" -> new CsvReport();
            default -> throw new IllegalArgumentException("Unknown type");
        };
    }
}

// Utilizzo nel Service
Report report = ReportFactory.create("PDF");
```

Simple Static Factory

22.3 Modern Java: Static Factory Methods

Questa è la declinazione più importante per un Senior Developer. Da Java 8 in poi, i costruttori sono considerati "limitati". Si preferiscono metodi statici con nomi parlanti.

Colloquio: Costruttore vs Static Factory Method

Domanda: Perché dovrei usare `User.of(...)` invece di `new User(...)`?

Risposta: I metodi statici hanno 3 vantaggi enormi:

1. **Hanno un nome:** Il costruttore si chiama sempre come la classe. Un metodo statico può chiamarsi `fromEmail()`, `fromId()`, `createGuest()`. È documentazione viva.
2. **Non devono per forza creare un nuovo oggetto:** Possono restituire un'istanza cachata (Singleton) o una costante (es. `Optional.empty()`).
3. **Possono restituire un sottotipo:** Un metodo che dichiara di ritornare `List` può restituire una `ArrayList` o una `LinkedList` o una `ImmutableCollections$ListN` senza che il client lo sappia.

```
// 1. List.of (Ritorna una lista immutabile ottimizzata)
List<String> list = List.of("A", "B");

// 2. Integer.valueOf (Usa la cache per numeri piccoli)
Integer i = Integer.valueOf(10);

// 3. Optional.ofNullable (Gestione null)
Optional<String> opt = Optional.ofNullable(maybeNullString);
```

Esempi nel JDK

22.4 Spring Framework: La Fabbrica Gigante

Spring è, essenzialmente, un contenitore IoC che agisce come una Factory. L'interfaccia radice di Spring non a caso si chiama **`BeanFactory`**.

Quando usi `@Autowired`, Spring sta facendo il lavoro della Factory per te: decide quale istanza creare (o recuperare se Singleton) e te la inietta.

22.4.1 Pattern Avanzato: FactoryBean

A volte la creazione di un oggetto è così complessa che non basta l'XML o l'annotazione `@Bean`. Spring offre l'interfaccia `FactoryBean<T>`.

Deep Dive: FactoryBean in azione

Hai mai usato JPA? Quando configuri Spring Data, definisci un `LocalContainerEntityManagerFactoryBean`. Questo **non** è l'EntityManager. È una Factory che costruisce l'EntityManager (leggendo il `persistence.xml`, configurando il DataSource, scansionando le entità, etc.). Spring vede che la classe implementa `FactoryBean`, quindi quando chiedi di iniettare l'EntityManager, chiama automaticamente `getObject()` sulla factory.

22.5 Esempio Architetturale: Factory + Strategy

Un pattern molto potente in Spring è combinare Factory e Strategy per eliminare gli `if` dai Service.

```
@Service
public class PaymentFactory {

    // Spring inietta automaticamente tutti i Bean che implementano
    // l'interfaccia PaymentService in una Mappa!
    // Key = nome del bean ("paypalService"), Value = istanza
    @Autowired
    private Map<String, PaymentService> services;

    public PaymentService getService(String type) {
        PaymentService service = services.get(type + "Service");
        if (service == null) throw new IllegalArgumentException("No provider found");
        return service;
    }
}
```

Service Locator Pattern (Spring Style)

22.6 Riepilogo

Concetto	Best Practice
Codice Client	Non usare `new` per logica complessa o polimorfica. Usa una Factory.
Naming	Usa convenzioni standard: `of()`, `from()`, `getInstance()`, `newInstance()`.
JDK Usage	Preferisci `List.of()` a `new ArrayList<>()` doppia graffa.
Spring	Sfrutta `BeanFactory` e l'iniezione di Mappe/Liste per creare Factory dinamiche senza switch-case.

Capitolo 23

Il Pattern Strategy (Behavioral)

Se apri una classe Service e trovi uno `switch` o una catena di `if-else` che controlla il "tipo" di un'operazione per decidere cosa fare, sei di fronte a un problema di design.

Il **Strategy Pattern** definisce una famiglia di algoritmi, li incapsula e li rende intercambiabili. In Spring, questo pattern diventa potentissimo perché il framework gestisce l'iniezione delle strategie automaticamente, eliminando quasi del tutto il codice di controllo.

23.1 Il Problema: "If-Else Hell"

Immagina un e-commerce che deve gestire pagamenti.

```java
@Service
public class CheckoutService {

    public void processPayment(String type, double amount) {
        if ("PAYPAL".equals(type)) {
            // Logica PayPal (50 righe)
            // Chiamate API esterne...
        } else if ("CREDIT_CARD".equals(type)) {
            // Logica Stripe (50 righe)
        } else if ("BITCOIN".equals(type)) {
            // Logica Crypto (50 righe)
        } else {
            throw new IllegalArgumentException("Metodo non supportato");
        }
    }
}
```

Approccio Procedurale (Bad)

Perché è male?

1. **Violazione OCP (Open/Closed Principle):** Se vuoi aggiungere "Apple Pay", devi modificare questa classe (rischiando di rompere gli altri pagamenti).
2. **Testabilità:** Per testare questa classe devi mockare tutte le dipendenze di tutti i metodi di pagamento.
3. **Complessità:** La classe diventa un "God Object" illeggibile.

23.2 Implementazione Classica (Interfaccia)

Il primo passo è estrarre la logica in classi separate che implementano un'interfaccia comune.

```java
public interface PaymentStrategy {
    void pay(double amount);

    // Metodo utile per l'identificazione in Spring (vedi dopo)
    String getType();
}

public class PaypalStrategy implements PaymentStrategy {
    public void pay(double amount) { /* API PayPal */ }
    public String getType() { return "PAYPAL"; }
}

public class CreditCardStrategy implements PaymentStrategy {
    public void pay(double amount) { /* API Stripe */ }
    public String getType() { return "CREDIT_CARD"; }
}
```

23.3 La "Magia" di Spring: Map Injection

In Java puro, dovresti comunque avere uno switch da qualche parte per fare `new PaypalStrategy()`. In Spring, possiamo usare l'IoC Container per eliminare anche quello.

Spring ha una feature killer: se chiedi di iniettare una `Map<String, Interfaccia>`, lui la popola automaticamente con tutti i Bean che implementano quell'interfaccia.

```java
@Service
@RequiredArgsConstructor
public class PaymentFactory { // O Context

    // Key: Nome del Bean (es. "paypalStrategy")
    // Value: L'istanza del Service
    private final Map<String, PaymentStrategy> strategies;

    public void executePayment(String type, double amount) {
        // 1. Recupero la strategia dalla mappa
        // Assumiamo che i bean si chiamino "PAYPAL", "CREDIT_CARD"
        // (oppure usiamo una logica di mapping)
        PaymentStrategy strategy = strategies.get(type);

        if (strategy == null) {
            throw new IllegalArgumentException("Metodo non supportato: " + type);
        }

        // 2. Eseguo (Polimorfismo)
        strategy.pay(amount);
    }
}
```

Strategy Pattern "Spring Way"

Deep Dive: Come nominare i Bean?

Per far funzionare la mappa con chiavi custom (es. "PAYPAL" invece di "paypalStrategy"), hai due vie:

1. Annotare l'implementazione: `@Service("PAYPAL")`
2. (Migliore) Usare un metodo `getType()` nell'interfaccia e convertire la `List<PaymentStrategy>` in una `Map` nel costruttore della Factory.

23.4 Enum Strategy (Per logica semplice)

Se le strategie non hanno dipendenze esterne (es. calcolo tasse, sconti), non serve creare classi e bean. Gli **Enum** in Java possono implementare interfacce e avere metodi astratti.

```java
public enum DiscountStrategy {
    NONE {
        @Override double apply(double price) { return price; }
    },
    SUMMER_SALE {
        @Override double apply(double price) { return price * 0.90; }
    },
    BLACK_FRIDAY {
        @Override double apply(double price) { return price * 0.50; }
    };

    abstract double apply(double price);
}

// Utilizzo:
double finalPrice = DiscountStrategy.valueOf("BLACK_FRIDAY").apply(100.0);
```

Strategy leggero con Enum

23.5 Interview Questions

Colloquio: Strategy vs State Pattern

Domanda: I diagrammi UML di Strategy e State sono quasi identici. Qual è la differenza?

Risposta: L'**Intento**.

- **Strategy:** Il client *sceglie* l'algoritmo specifico (es. "Voglio pagare con PayPal"). Le strategie solitamente non sanno l'una dell'altra.
- **State:** L'oggetto cambia comportamento internamente in base al suo stato (es. Ordine: "Creato" → "Spedito"). Spesso uno stato contiene la logica per passare allo stato successivo.

Colloquio: Strategy e Bean Lifecycle

Domanda: Le strategie in Spring sono Singleton? **Risposta:** Sì, di default. Questo significa che le classi `PaypalStrategy`, etc., devono essere **Stateless** (senza stato). Non salvare dati specifici della transazione (es. `amount`) come campi di istanza, ma passali sempre come argomenti del metodo `pay()`.

23.6 Riepilogo

Concetto	Best Practice
Obiettivo	Eliminare `if-else` complessi basati su "tipo". Rispetta OCP.
Implementazione Spring	Inietta `Map<String, Strategy>` o `List<Strategy>` per selezionare l'implementazione a runtime.
Stateless	Le strategie devono essere bean puri senza stato. I dati viaggiano nei parametri dei metodi.
Alternative	Per logica matematica semplice, usa **Enum** con metodi astratti.

Capitolo 24

Il Pattern Proxy (Structural)

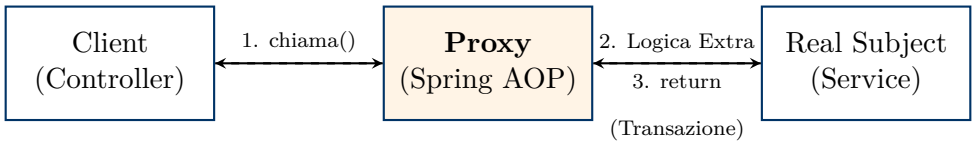
Se Spring e Hibernate fossero un film, il **Proxy** sarebbe l'attore protagonista. È il meccanismo fondamentale che permette ai framework di aggiungere comportamenti (transazioni, sicurezza, lazy loading) al tuo codice senza che tu debba scriverli esplicitamente.

Il Proxy è un oggetto che agisce come un surrogato o segnaposto per un altro oggetto per controllarne l'accesso.

24.1 Il Concetto: L'Intermediario

Immagina di voler chiamare il metodo `save()` di un Repository.

- **Senza Proxy:** Il tuo Controller chiama direttamente il Repository.
- **Con Proxy (Spring):** Il tuo Controller chiama un oggetto wrapper (il Proxy). Il Proxy apre la transazione, chiama il Repository vero, e poi chiude la transazione.



24.2 JDK Dynamic Proxy vs CGLIB

Ai colloqui tecnici avanzati, ti chiederanno: *"Come fa Spring a creare questi Proxy a runtime?"*. Esistono due tecnologie.

24.2.1 1. JDK Dynamic Proxy (L'originale)

Fa parte del cuore di Java (`java.lang.reflect.Proxy`).

- **Requisito:** L'oggetto target DEVE implementare un'**Interfaccia**.
- **Funzionamento:** Il Proxy implementa la stessa interfaccia del target e delega le chiamate.
- **Limite:** Se la tua classe non ha interfacce, questo metodo fallisce.

24.2.2 2. CGLIB (Code Generation Library)

È una libreria di terze parti (inclusa in Spring Core).

- **Funzionamento:** Genera al volo una **Sottoclasse** del tuo oggetto target e fa *Override* dei metodi.

- **Vantaggio:** Funziona anche senza interfacce (per questo in Spring Boot possiamo autowirare le classi concrete).
- **Limite:** Non può proxare metodi o classi `final` (perché non si possono estendere/overridare).

Deep Dive: Spring Boot Default

Dalla versione 2.0, Spring Boot usa **CGLIB** di default per tutto, anche se hai delle interfacce. Questo per evitare comportamenti inconsistenti (es. casting exceptions se provi a castare il proxy alla classe concreta).

24.3 Spring AOP: Il Proxy in Azione

Prendiamo l'annotazione `@Transactional`. È solo un metadato. Non fa nulla da sola. È il **BeanPostProcessor** di Spring che, all'avvio, vede l'annotazione e decide di avvolgere il tuo Bean in un Proxy.

```
public class UserServiceProxy extends UserService { // CGLIB

    private UserService target; // Il vero service
    private TransactionManager txManager;

    @Override
    public void createUser(User u) {
        // 1. Pre-Processing (Aspect)
        Transaction tx = txManager.begin();

        try {
            // 2. Chiamata al metodo reale
            target.createUser(u);

            // 3. Post-Processing (Success)
            txManager.commit(tx);
        } catch (RuntimeException e) {
            // 4. Error Handling
            txManager.rollback(tx);
            throw e;
        }
    }
}
```

Cosa fa il Proxy (Codice simulato)

24.4 La Trappola della Self-Invocation

Questo è il problema numero uno causato dai Proxy.

Colloquio: Perché @Transactional viene ignorato?

Domanda: Ho un metodo `a()` che chiama `b()` nella stessa classe. `b()` è annotato con `@Transactional`, ma la transazione non parte. Perché?

```
public class MyService {
```

```
    public void a() {
        this.b(); // Chiamata interna
    }

    @Transactional
    public void b() { ... }
}
```

Risposta: Quando chiami `this.b()`, stai usando il riferimento `this`, che punta all'istanza **Reale**, non al Proxy. Spring non può intercettare la chiamata. Il codice di apertura transazione non viene mai eseguito. **Soluzione:**

1. Spostare `b()` in un altro Service (e iniettarlo).
2. (Brutta) Iniettare il Service dentro se stesso (`@Lazy @Autowired MyService self`) e chiamare `self.b()`.

24.5 Hibernate: Il Lazy Loading Proxy

Anche Hibernate usa i Proxy, ma per un motivo diverso: le performance.

Quando carichi un oggetto che ha una relazione `LAZY` (es. `user.getDepartment()`), Hibernate non fa la query per il dipartimento. Al suo posto, mette nel campo `department` un Proxy CGLIB/Javassist.

- Questo Proxy ha solo l'ID popolato. Tutti gli altri campi sono null.
- Appena chiami un metodo (es. `dept.getName()`), il Proxy si "sveglia", apre una connessione al DB, esegue la query e si riempie i dati (*Initialization*).

Deep Dive: LazyInitializationException spiegata col Proxy

Se provi a chiamare `dept.getName()` quando la Sessione Hibernate è chiusa:

1. Il Proxy intercetta la chiamata.
2. Cerca di chiedere alla Sessione di eseguire l'SQL.
3. Trova la Sessione chiusa.
4. Lancia l'eccezione.

L'eccezione non viene lanciata dal tuo oggetto, ma dal codice generato dentro il Proxy.

24.6 Riepilogo

Concetto	Dettaglio
Scopo	Controllare l'accesso a un oggetto (Lazy Load) o aggiungere comportamenti (AOP).
Spring	Usa Proxy per `@Transactional`, `@Async`, `@Cacheable`, Security.
Limitazione Key	**Self-Invocation:** Le chiamate interne (`this.method()`) scavalcano il proxy e perdono le funzionalità AOP.
Final	Le classi o i metodi `final` non possono essere proxati con CGLIB (Spring darà errore all'avvio).

Capitolo 25

Il Pattern Adapter (Structural)

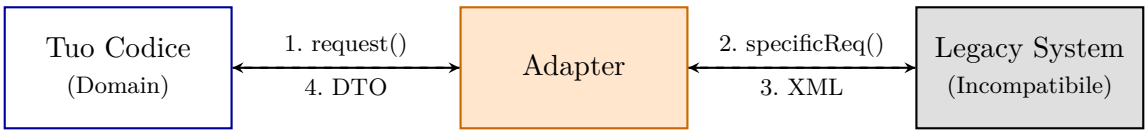
"Se qualcosa non si incastra, forza finché non entra". Questa è una pessima idea in meccanica, ma è la base dell'integrazione software.

Il **Adapter Pattern** permette a interfacce incompatibili di lavorare insieme. Agisce come un convertitore (pensa all'adattatore della presa elettrica USA → EU) che traduce le chiamate del client in un formato comprensibile per il sistema target.

25.1 Il Problema: L'Integrazione Legacy

Stai costruendo un moderno e-commerce che usa `JSON` e oggetti DTO puliti. Tuttavia, devi integrare un vecchio sistema di fatturazione (Legacy) fornito dalla banca, che accetta solo `XML` e ha nomi dei metodi incomprensibili.

- **Opzione A (Bad):** Sporchi il tuo codice moderno con logica XML e chiamate al sistema vecchio ovunque.
- **Opzione B (Good):** Crei un Adapter che nasconde la "bruttezza" del sistema vecchio dietro un'interfaccia pulita.



25.2 Implementazione Pratica

25.2.1 1. L'Adaptee (Il sistema vecchio)

Questa è la classe che non puoi modificare (es. una libreria esterna chiusa).

```
// Classe chiusa, metodi final, logica oscura
public class OldBankSystem {
    public String sendPaymentXML(String xmlData) {
        System.out.println("Processing XML: " + xmlData);
        return "<response>SUCCESS</response>";
    }
}
```

25.2.2 2. Il Target (La tua interfaccia ideale)

Questo è come il tuo codice vorrebbe interagire con il mondo.

```
public interface PaymentGateway {
    boolean processPayment(PaymentDTO payment);
}
```

25.2.3 3. L'Adapter (Il ponte)

Implementa la tua interfaccia e delega al sistema vecchio, facendo la traduzione.

```
@Service // In Spring diventa un Bean iniettabile
@RequiredArgsConstructor
public class BankAdapter implements PaymentGateway {

    private final OldBankSystem oldSystem;

    @Override
    public boolean processPayment(PaymentDTO payment) {
        // 1. Traduzione Input (DTO -> XML)
        String xmlPayload = convertToXml(payment);

        // 2. Chiamata al sistema Legacy
        String response = oldSystem.sendPaymentXML(xmlPayload);

        // 3. Traduzione Output (XML -> boolean)
        return response.contains("SUCCESS");
    }

    private String convertToXml(PaymentDTO dto) {
        return "<payment amount='" + dto.getAmount() + "' />";
    }
}
```

25.3 Java Core Examples

L'Adapter è ovunque nel JDK.

25.3.1 Arrays.asList()

Un array `String[]` e una `List<String>` sono incompatibili. Il metodo statico `Arrays.asList()` funge da Adapter: avvolge l'array e lo fa sembrare una lista (pur mantenendo i dati nell'array originale).

25.3.2 InputStreamReader

I flussi di I/O sono pieni di adapter.

- `InputStream`: Legge byte grezzi.
- `Reader`: Legge caratteri (char).

`InputStreamReader` adatta un `InputStream` per farlo diventare un `Reader`.

25.4 Spring MVC: HandlerAdapter

Come fa Spring a chiamare il tuo metodo nel Controller, indipendentemente da come lo hai scritto (con o senza parametri, con o senza `@RequestBody`)?

Usa il pattern **HandlerAdapter**. Il `DispatcherServlet` (il cuore di MVC) non chiama direttamente i controller. Delega a un Adapter.

- Esiste un `RequestMappingHandlerAdapter` che sa come invocare i metodi annotati con `@RequestMapping`.
- Se domani inventi un nuovo modo di scrivere controller, basta scrivere un nuovo Adapter e Spring potrà usarlo senza modificare il core.

25.5 Interview Questions

Colloquio: Adapter vs Decorator vs Proxy

Domanda: Sono tutti wrapper. Qual è la differenza?

Risposta:

- **Adapter:** Cambia l'**Interfaccia**. Fa collaborare oggetti incompatibili (presa tonda in spina quadrata).
- **Decorator:** Mantiene l'interfaccia ma aggiunge **Responsabilità/Comportamento** (es. aggiungere la compressione a uno stream).
- **Proxy:** Mantiene l'interfaccia ma controlla l'**Accesso** (es. sicurezza, transazioni, lazy loading).

Deep Dive: DDD: Anti-Corruption Layer (ACL)

Nel Domain-Driven Design, l'Adapter è il blocco costruttivo dell'ACL. Se il tuo dominio deve parlare con un sistema ERP "sporco", costruisci uno strato di Adapter che impedisce ai termini e alle strutture dati dell'ERP di infiltrarsi nel tuo codice pulito. Il tuo codice conosce solo i DTO dell'ACL, mai gli oggetti dell'ERP.

25.6 Riepilogo

Concetto	Best Practice
Scopo	Integrare classi incompatibili o Legacy senza modificare il codice esistente.
Principio SOLID	Rispetta l'**Interface Segregation Principle** e l'**Open/Closed** (estendi il sistema con l'adapter senza toccare il vecchio codice).
Composizione	Preferisci l'Adapter a Oggetti (composizione) rispetto all'Adapter di Classe (ereditarietà multipla, che in Java non esiste).

Capitolo 26

Il Pattern Observer (Behavioral)

Nelle architetture monolitiche disordinate, i Service si chiamano tutti a vicenda. Il `UserService` chiama il `EmailService`, che chiama il `AuditService`, che chiama il `AnalyticsService`... creando un groviglio inestricabile (Spaghetti Code).

Il **Observer Pattern** risolve questo problema invertendo la dipendenza. Invece di ordinare agli altri di fare qualcosa, un oggetto dice semplicemente: *"È successo questo evento!"*. Chi è interessato, reagirà.

26.1 Il Problema: Accoppiamento Stretto

Immagina il caso d'uso: **Registrazione Utente**. Dopo aver salvato l'utente, devi fare 3 cose: inviare una mail di benvenuto, tracciare l'evento su Google Analytics e scrivere un log di audit.

```
@Service
@RequiredArgsConstructor
public class UserService {
    private final UserRepository repo;
    private final EmailService emailService;       // Dipendenza 1
    private final AnalyticsService analyticsService; // Dipendenza 2
    private final AuditService auditService;       // Dipendenza 3

    public void register(User user) {
        repo.save(user);

        // Il Service sa TROPPO. Se aggiungi un quarto step, devi modificare qui.
        emailService.sendWelcomeEmail(user);
        analyticsService.trackRegistration(user);
        auditService.logAction("REGISTER", user);
    }
}
```

Approccio Procedurale (Bad)

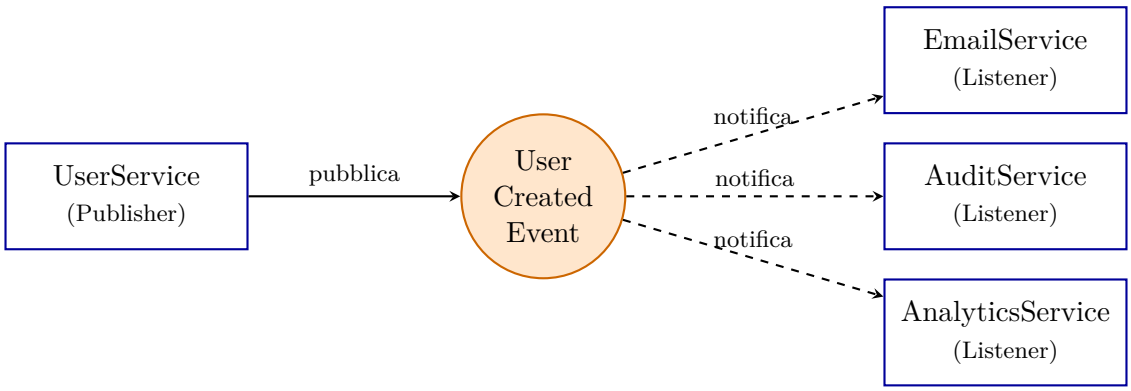
26.2 La Soluzione Spring: Application Events

Spring implementa l'Observer Pattern in modo nativo e potentissimo tramite l'**ApplicationContext**.

Il flusso è composto da tre attori:

1. **Event:** Un semplice POJO che contiene i dati (il messaggio).

2. **Publisher:** Colui che lancia l'evento (il Subject).
3. **Listener:** Colui che ascolta e reagisce (l'Observer).



26.2.1 1. L'Evento (POJO)

In Spring moderno, non serve più estendere classi speciali. Basta un oggetto Java.

```
// I Record sono perfetti per gli eventi (immutabili)
public record UserCreatedEvent(String email, String username) {}
```

26.2.2 2. Il Publisher (UserService)

Il Service non conosce più gli altri componenti. Inietta solo il `ApplicationEventPublisher`.

```
@Service
@RequiredArgsConstructor
public class UserService {
    private final UserRepository repo;
    private final ApplicationEventPublisher publisher; // Interfaccia Spring

    public void register(User user) {
        repo.save(user);

        // Lancia e dimentica. Disaccoppiamento totale.
        publisher.publishEvent(new UserCreatedEvent(user.getEmail(), user.getUsername
    ↪ ()));
    }
}
```

26.2.3 3. I Listener (Observers)

Gli altri service ascoltano l'evento usando l'annotazione `@EventListener`.

```
@Component
public class EmailListener {

    @EventListener
    public void handleUserCreated(UserCreatedEvent event) {
        // Manda la mail...
        System.out.println("Invio email a " + event.email());
    }
}
```

```

@Component
public class AuditListener {

    @EventListener
    public void handleUserCreated(UserCreatedEvent event) {
        // Scrive il log...
        System.out.println("Audit log per " + event.username());
    }
}
```

26.3 Sincrono vs Asincrono

Qui casca l'asino (e il Junior).

Colloquio: Spring Events sono Async?

Domanda: Se uso `@EventListener`, l'invio della mail avviene in background?

Risposta: No, di default è Sincrono. Tutto avviene nello stesso Thread del `UserService`. Se l'invio della mail impiega 5 secondi, la risposta HTTP della registrazione ritarderà di 5 secondi. Se l'invio della mail lancia un'eccezione non gestita, la transazione di registrazione potrebbe fare Rollback!

26.3.1 Come renderlo Asincrono (@Async)

Se vuoi che l'evento sia gestito in un thread separato (Fire-and-Forget reale), devi combinare due annotazioni.

```
@Component
public class AsyncEmailListener {

    @Async // Esegui in un thread separato (TaskExecutor)
    @EventListener
    public void sendEmailAsync(UserCreatedEvent event) {
        // ... operazione lenta ...
    }
}
```

Nota: Ricorda di abilitare `@EnableAsync` nella configurazione.

26.4 Transactional Events (Livello Senior)

Cosa succede se salvi l'utente, pubblichi l'evento, mandi la mail (sincrona), ma poi **la transazione del DB fallisce** e fa Rollback?

- L'utente **non** è stato creato nel DB.
- Ma la mail di benvenuto è partita!

Questo è uno stato inconsistente ("Phantom Notification").

26.4.1 La Soluzione: @TransactionalEventListener

Spring permette di ascoltare l'evento solo in una specifica fase della transazione.

```java
@Component
public class SafeEmailListener {

    // Esegui questo metodo SOLO se la transazione del Publisher
    // ha fatto COMMIT con successo.
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void sendEmailOnlyIfCommitted(UserCreatedEvent event) {
        // Ora siamo sicuri che l'utente esiste nel DB.
        emailService.send(event.email());
    }
}
```

26.5 Riepilogo

Caratteristica	Dettaglio
Scopo	Disaccoppiare chi produce un dato da chi lo consuma. Relazione 1-a-Molti.
Implementazione	Usa `ApplicationEventPublisher` e `@EventListener`. Evita l'interfaccia `Observer` di Java (deprecata).
Default	Sincrono. Blocca il thread chiamante.
Consistency	Usa `@TransactionalEventListener` per garantire che l'evento parta solo a transazione conclusa.

Capitolo 27

Il Pattern Decorator (Structural)

"Preferire la Composizione all'Ereditarietà". Il **Decorator Pattern** è la massima espressione di questo principio.

Permette di aggiungere funzionalità a un oggetto dinamicamente, avvolgendolo in un guscio (wrapper). È come una matrioska: l'oggetto esterno aggiunge qualcosa e poi delega il lavoro all'oggetto interno.

27.1 Il Problema: L'Esplosione delle Classi

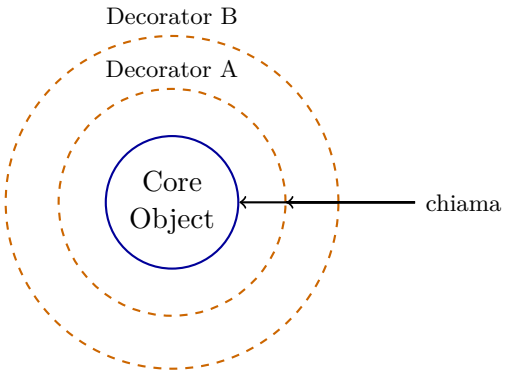
Immagina di dover gestire dei flussi di dati. Vuoi supportare:

- Lettura dati base.
- Crittografia.
- Compressione.

Se usi l'ereditarietà, ti trovi in un incubo combinatorio:

- `FileStream`
- `EncryptedFileStream`
- `CompressedFileStream`
- `EncryptedCompressedFileStream` ...

Ogni nuova feature raddoppia il numero di classi necessarie.



27.2 Esempio Reale: Java I/O

Il miglior esempio di Decorator è la libreria standard `java.io`. Hai mai notato questa sintassi "a cipolla"?

```
// File -> Buffer -> GZip -> Object
InputStream in = new ObjectInputStream(
    new GZIPInputStream(
        new BufferedInputStream(
            new FileInputStream("data.txt")
        )
    )
);
```

- **FileInputStream:** Componente Concreto (legge i byte).
- **BufferedInputStream:** Decorator (aggiunge un buffer in RAM per performance).
- **GZIPInputStream:** Decorator (decomprime i dati al volo).

Tutti implementano la stessa interfaccia (`InputStream`). Il client non sa e non si preoccupa di quanti strati ci siano.

27.3 Implementazione Manuale

Vediamo come costruire un sistema di notifiche modulare (Email, SMS, Slack) usando il Decorator.

27.3.1 1. Interfaccia Comune

```
public interface Notifier {
    void send(String message);
}
```

27.3.2 2. Componente Base

```
public class EmailNotifier implements Notifier {
    public void send(String message) {
        System.out.println("Invio Email: " + message);
    }
}
```

27.3.3 3. Decorator Base (Astratto)

Mantiene il riferimento all'oggetto "wrappato".

```
@RequiredArgsConstructor
public abstract class NotifierDecorator implements Notifier {
    protected final Notifier wrapped;

    @Override
    public void send(String message) {
        wrapped.send(message); // Delega standard
    }
}
```

27.3.4 4. Decoratori Concreti

Aggiungono comportamento prima o dopo la delega.

```
public class SMSDecorator extends NotifierDecorator {
    public SMSDecorator(Notifier wrapped) { super(wrapped); }

    @Override
    public void send(String message) {
        super.send(message); // Manda l'email (o il precedente)
        System.out.println("Invio SMS: " + message); // Funzionalità aggiunta
    }
}

public class SlackDecorator extends NotifierDecorator {
    public SlackDecorator(Notifier wrapped) { super(wrapped); }

    @Override
    public void send(String message) {
        super.send(message);
        System.out.println("Post su Slack: " + message);
    }
}
```

27.3.5 Utilizzo

```
// Voglio Email + SMS + Slack
Notifier stack = new SlackDecorator(
                    new SMSDecorator(
                        new EmailNotifier()
                    )
                );

stack.send("Server Down!");
// Output:
// Invio Email...
// Invio SMS...
// Post su Slack...
```

27.4 Real World in Spring/Enterprise

27.4.1 1. Collections.unmodifiableList()

Quando chiami `Collections.unmodifiableList(list)`, Java non copia i dati. Restituisce un Decorator che avvolge la tua lista originale.

- I metodi di lettura (`get`) sono delegati alla lista originale.
- I metodi di scrittura (`add`, `remove`) sono sovrascritti per lanciare `UnsupportedOperationException`.

27.4.2 2. Servlet Filters (HttpServletRequestWrapper)

In Spring Security o nei filtri custom, spesso vogliamo modificare la richiesta HTTP (es. fare il trim delle stringhe in input o sanificare XSS). Non possiamo modificare l'oggetto `HttpServletRequest` originale.

Usiamo `HttpServletRequestWrapper` (un Decorator).

```
public class XSSRequestWrapper extends HttpServletRequestWrapper {
    public XSSRequestWrapper(HttpServletRequest request) {
        super(request);
    }

    @Override
    public String getParameter(String name) {
        String value = super.getParameter(name);
        return sanitizeXSS(value); // Aggiunge comportamento al volo
    }
}
```

27.5 Interview Questions

Colloquio: Decorator vs Proxy

Domanda: Strutturalmente sono identici (un wrapper che delega a un oggetto interno). Qual è la differenza?

Risposta: L'**Intento**.

- **Proxy:** Controlla l'**Accesso**. Non cambia la funzionalità di business, ma aggiunge logica "amministrativa" (Lazy Loading, Security, Transaction). Spesso il client non sa di usare un proxy.
- **Decorator:** Aggiunge **Funzionalità/Responsabilità**. Arricchisce l'oggetto (es. aggiunge compressione, bordi grafici, notifiche extra). Il client spesso compone la catena manualmente.

Colloquio: Decorator vs Inheritance

Domanda: Quando l'ereditarietà è preferibile al Decorator? **Risposta:** Quando la funzionalità aggiunta dipende strettamente dalla struttura interna della classe padre e non deve essere combinata dinamicamente. Il Decorator è potente ma crea tanti piccoli oggetti in memoria e rende il debugging più difficile (stack trace profondi).

27.6 Riepilogo

Caratteristica	Dettaglio
Obiettivo	Aggiungere responsabilità a singoli oggetti dinamicamente, senza estendere classi.
Vantaggio	Evita l'esplosione combinatoria delle sottoclassi. Rispetta il Single Responsibility Principle.
Esempi Java	`java.io.*` (Streams), `Collections.synchronizedList()`, `HttpServletRequestWrapper`.

Capitolo 28

Il Pattern Template Method (Behavioral)

Hai mai notato che molte operazioni seguono uno schema fisso, dove cambiano solo pochi dettagli? Aprire file → **Processare riga** → Chiudere file. Aprire connessione DB → **Mappare ResultSet** → Chiudere connessione.

Il **Template Method** definisce lo scheletro di un algoritmo in una classe base, deferendo alcuni passaggi alle sottoclassi. È l'incarnazione del **Hollywood Principle**: *"Non chiamateci, vi chiameremo noi"*.

28.1 Il Problema: Duplicazione del Flusso

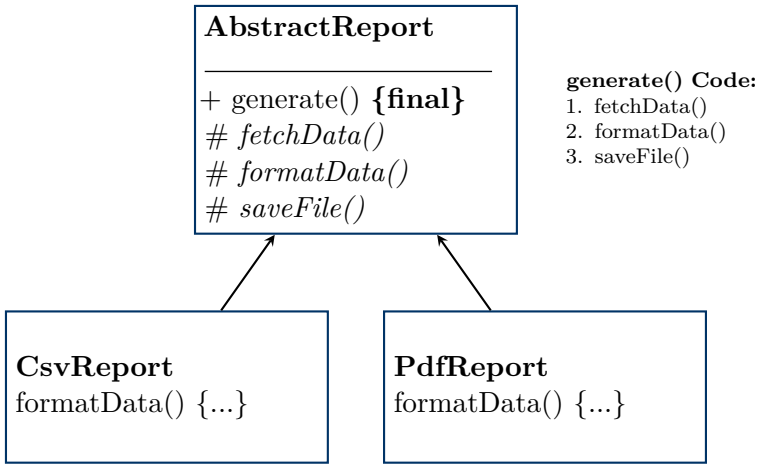
Immagina di dover generare report in formati diversi (CSV e PDF).

- **CsvReport:** Connette al DB, recupera dati, converte in stringhe separate da virgola, salva su file.
- **PdfReport:** Connette al DB, recupera dati, disegna tabelle grafiche, salva su file.

Il 90% del codice (connessione, recupero dati, gestione errori I/O, salvataggio) è identico. Solo la fase di "formattazione" cambia. Se copi e incolli il codice in due classi, al primo bug dovrai correggere in due posti.

28.2 La Soluzione: Lo Scheletro

Creiamo una classe astratta che possiede un metodo `final` (il Template) che detta legge sull'ordine di esecuzione.



28.3 Implementazione Pratica

28.3.1 1. La Classe Astratta (Template)

```java
public abstract class DataProcessor {

    // IL TEMPLATE METHOD
    // È final per impedire che le sottoclassi cambino l'algoritmo.
    public final void process() {
        try {
            openResource();

            // Passaggio variabile (astratto)
            processData();

            // Hook (opzionale)
            if (shouldLog()) {
                System.out.println("Processing completato");
            }
        } finally {
            closeResource(); // Eseguito sempre
        }
    }

    // Parte fissa (comune a tutti)
    private void openResource() { System.out.println("Apro File..."); }
    private void closeResource() { System.out.println("Chiudo File..."); }

    // Parte variabile (obbligatoria)
    protected abstract void processData();

    // Hook (facoltativo, ha implementazione di default vuota)
    protected boolean shouldLog() { return true; }
}
```

28.3.2 2. Le Sottoclassi Concrete

```java
public class CsvProcessor extends DataProcessor {
```

```
    @Override
    protected void processData() {
        System.out.println("Converto in CSV...");
    }
}

public class JsonProcessor extends DataProcessor {
    @Override
    protected void processData() {
        System.out.println("Serializzo in JSON...");
    }

    @Override
    protected boolean shouldLog() {
        return false; // Override dell'hook
    }
}
```

28.4 Real World: Spring Templates

Spring usa massicciamente questo pattern, ma con una variante moderna: invece di usare l'ereditarietà (classe astratta), usa le **Callback** (interfacce funzionali).

28.4.1 JdbcTemplate

Pensaci: quando usi JDBC puro devi aprire la connessione, creare lo statement, ciclare il ResultSet, gestire `SQLException` e chiudere tutto nel finally. Spring `JdbcTemplate` fa tutto questo per te.

- **Parte Fissa (Template):** Gestione connessione, transazione, try-catch-finally.
- **Parte Variabile:** "Come mappo questa singola riga SQL su un oggetto Java?".

Tu fornisci solo la parte variabile tramite un `RowMapper`:

```
// JdbcTemplate.query è il Template Method
jdbcTemplate.query(
    "SELECT * FROM users",

    // Questa lambda è l'implementazione del passaggio variabile
    (resultSet, rowNum) -> new User(resultSet.getString("name"))
);
```

28.5 Template vs Strategy

Questa è la domanda da colloquio per eccellenza per distinguere i pattern.

Colloquio: Differenza tra Template Method e Strategy

Domanda: Entrambi permettono di variare parti di un algoritmo. Qual è la differenza?

Risposta:

- **Template Method (Ereditarietà):** La struttura è definita nella classe padre. Modifichi il comportamento **estendendo** la classe e sovrascrivendo i metodi. È statico (compilazione).

- **Strategy (Composizione):** La logica è delegata a un oggetto esterno (interfaccia). Modifichi il comportamento **iniettando** una diversa implementazione. È dinamico (runtime).

Verdetto Senior: Oggi si preferisce quasi sempre lo **Strategy** ("Favor composition over inheritance"), tranne quando si scrivono framework o librerie base dove si vuole forzare uno scheletro rigido.

28.6 Modern Java: Execute Around Pattern

Con le Lambda di Java 8, il Template Method è evoluto nel pattern **Execute Around**. Invece di creare una classe figlia per ogni variante, passi un comportamento (Lambda) a un metodo.

```
public class TransactionHelper {

    // Metodo che accetta una funzione (la parte variabile)
    public static void runInTransaction(Runnable businessLogic) {
        System.out.println("BEGIN TX");
        try {
            businessLogic.run(); // Esegue il pezzo variabile
            System.out.println("COMMIT");
        } catch (Exception e) {
            System.out.println("ROLLBACK");
        }
    }
}

// Utilizzo (senza estendere classi!)
TransactionHelper.runInTransaction(() -> {
    System.out.println("Salvataggio Utente...");
});
```

Template Method Funzionale

28.7 Riepilogo

Caratteristica	Dettaglio
Obiettivo	Definire lo scheletro di un algoritmo, permettendo alle sottoclassi di ridefinire certi passaggi senza cambiare la struttura.
Hook	Metodi opzionali (spesso vuoti nel padre) che le sottoclassi possono decidere di estendere o ignorare.
Spring	Usato in `JdbcTemplate`, `RestTemplate`, `JmsTemplate` (nella variante con Callback).
Contro	L'ereditarietà è rigida. Se puoi, preferisci Strategy o le Lambda (Execute Around).

Capitolo 29

Il Pattern Chain of Responsibility (Behavioral)

"Passarsi la palla". In un sistema complesso, spesso una richiesta deve superare una serie di controlli prima di essere elaborata: Autenticazione, Autorizzazione, Validazione, Logging, Caching.

Se mettessimo tutta questa logica in un unico metodo, avremmo un mostro procedurale. Il **Chain of Responsibility Pattern** permette di passare una richiesta lungo una catena di "Gestori" (Handlers). Ogni gestore decide se:

1. Elaborare la richiesta.
2. Passarla al prossimo anello della catena.
3. Bloccare la catena (Short-circuit).

29.1 Il Problema: Validazione Sequenziale

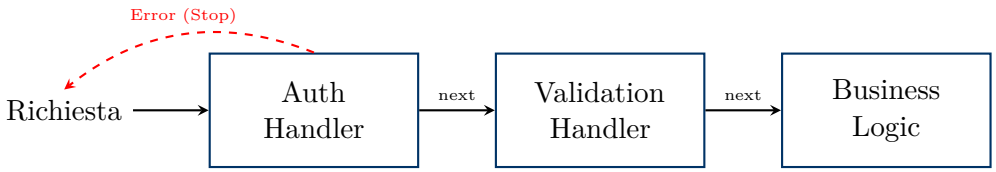
Immagina un sistema di approvazione spese aziendali.

- Spese < 100€: Approvate automaticamente.
- Spese < 1000€: Richiedono approvazione del Manager.
- Spese > 1000€: Richiedono approvazione del CEO.

Scrivere questo con `if-else` annidati crea codice rigido. Se domani aggiungiamo il "VP Finance", dobbiamo toccare la logica di tutti.

29.2 La Soluzione: La Catena

Creiamo una serie di oggetti collegati. Ognuno conosce solo il suo successore.



29.3 Implementazione Classica (GoF)

In questo approccio (Linked List), ogni handler ha un riferimento al prossimo.

```java
public abstract class Handler {
    protected Handler next;

    public void setNext(Handler next) { this.next = next; }

    public void handleRequest(Request req) {
        if (canHandle(req)) {
            process(req);
        } else if (next != null) {
            next.handleRequest(req); // Passa la palla
        }
    }

    protected abstract boolean canHandle(Request req);
    protected abstract void process(Request req);
}
```

29.4 Spring Security: The Ultimate Chain

Il caso d'uso più famoso nel mondo Java è la **Security Filter Chain**.

Quando una richiesta HTTP arriva al tuo server (Tomcat), prima di toccare il tuo Controller, deve attraversare una catena di filtri gestita da Spring Security.

1. **CorsFilter:** Controlla gli header CORS. Se fallisce, blocca tutto.
2. **CsrfFilter:** Controlla il token anti-forgery.
3. **JwtAuthenticationFilter:** (Custom) Controlla se c'è il Token Bearer, lo valida e setta l'utente nel contesto.
4. **UsernamePasswordAuthenticationFilter:** Gestisce il login classico.
5. **FilterSecurityInterceptor:** L'ultimo della catena. Controlla se l'utente ha il ruolo giusto (es. `ROLE_ADMIN`) per accedere all'URL.

Deep Dive: Short-Circuiting (Il Muro)

La potenza della Chain è la capacità di interrompere il flusso. Se il `JwtAuthenticationFilter` scopre che il token è scaduto:

1. Lancia un'eccezione o scrive direttamente nella Response (HTTP 401).
2. **NON** chiama `chain.doFilter()`.
3. La richiesta muore lì. Il Database non viene nemmeno toccato. Questo protegge il sistema da carico inutile.

29.5 Implementazione "Spring Style" (List Injection)

Invece di linkare manualmente gli handler (‘h1.setNext(h2)‘), usiamo la Dependency Injection di Spring per iniettare una lista ordinata. È più pulito.

```java
// 1. Interfaccia comune
public interface OrderValidator {
    void validate(Order order);
}

// 2. Implementazioni con ordine specifico
@Component
```

```java
@Order(1)
public class StockValidator implements OrderValidator { ... }

@Component
@Order(2)
public class FraudValidator implements OrderValidator { ... }

@Component
@Order(3)
public class PaymentValidator implements OrderValidator { ... }

// 3. Il "Manager" della catena
@Service
@RequiredArgsConstructor
public class OrderProcessingService {

    // Spring inietta tutti i bean trovati, ordinati per @Order!
    private final List<OrderValidator> validators;

    public void process(Order order) {
        // Esegue la catena sequenzialmente
        for (OrderValidator v : validators) {
            v.validate(order); // Se uno lancia eccezione, il processo si ferma
        }

        saveOrder(order);
    }
}
```

Chain con @Order

29.6 Spring MVC Interceptors

Un'altra incarnazione del pattern è `HandlerInterceptor`. Permette di eseguire codice prima e dopo l'esecuzione del Controller.

- `preHandle()`: Prima del controller. Se ritorna `false`, la catena si ferma.
- `postHandle()`: Dopo il controller, prima del rendering della vista.
- `afterCompletion()`: Dopo che la risposta è stata inviata al client (utile per pulizia risorse o logging finale).

29.7 Interview Questions

Colloquio: Chain vs Decorator

Domanda: Entrambi avvolgono logica in sequenza. Qual è la differenza?

Risposta:

- **Decorator:** Aggiunge funzionalità e **passa sempre** la chiamata all'oggetto decorato. L'obiettivo è arricchire il risultato.
- **Chain of Responsibility:** Ogni handler può decidere di **interrompere** il flusso e non chiamare il successivo. L'obiettivo è processare (o bloccare) una richiesta. Inoltre, nella Chain classica, spesso solo *uno* degli handler elabora la richiesta, mentre nel Decorator tutti contribuiscono.

Colloquio: Ordine di Esecuzione

Domanda: In Spring Security, l'ordine dei filtri è importante? **Risposta: Assolutamente sì.** Non puoi controllare se l'utente è "ADMIN" (Authorization) se prima non hai capito "Chi è" (Authentication). Ecco perché il filtro JWT deve girare prima del filtro di Autorizzazione.

29.8 Riepilogo

Caratteristica	Dettaglio
Scopo	Disaccoppiare il mittente dai destinatari, permettendo a più oggetti di gestire la richiesta.
Spring Security	È l'esempio principe. Una catena di filtri che protegge l'applicazione.
Short-Circuit	La capacità di un anello di bloccare la catena (es. Auth fallita) è la feature chiave per la sicurezza e la validazione.
@Order	Usare l'annotazione di Spring per definire la sequenza dei Bean in una lista iniettata.

Parte V

JDBC, JPA e Hibernate

Capitolo 30

JDBC: L'Interfaccia Standard per Database

Prima di Hibernate, prima di Spring Data, c'era (e c'è tuttora) JDBC (Java Database Connectivity). Anche se usi framework di alto livello, sotto il cofano tutto viene tradotto in chiamate JDBC. Capire JDBC è essenziale per il debugging, per le operazioni di batching massivo e per evitare problemi di sicurezza (SQL Injection).

JDBC è un insieme di interfacce (pacchetto `java.sql`) che definiscono uno standard. Sono i produttori dei Database (Oracle, PostgreSQL, MySQL) a fornire l'implementazione concreta tramite i **Driver JDBC**.

I componenti fondamentali sono:

1. **DriverManager:** La classe di utility che gestisce i driver registrati e crea le connessioni.
2. **Connection:** Rappresenta la sessione fisica (socket TCP) con il database. È un oggetto "pesante" da creare.
3. **Statement / PreparedStatement:** L'oggetto che trasporta la query SQL al database.
4. **ResultSet:** L'oggetto che contiene i dati restituiti (una tabella in memoria puntata da un cursore).

30.1 Il DriverManager: Il "Centralino" delle Connessioni

Il `DriverManager` è il punto di ingresso storico di JDBC. Immaginalo come un centralino: la tua applicazione non parla direttamente con il driver specifico (MySQL, Oracle), ma chiede al DriverManager una connessione generica, e lui si occupa di trovare il traduttore giusto.

30.1.1 Come funziona la selezione del Driver?

Quando chiami `getConnection`, il DriverManager scorre una lista interna di driver registrati e pone a ciascuno una domanda: *"Sei in grado di comprendere questa URL?"*.

Il primo driver che risponde "Sì" viene selezionato per creare la connessione.

```
// 1. Definiamo la stringa di connessione (JDBC URL)
// Sintassi: jdbc:<vendor>:<proprietà>
String url = "jdbc:postgresql://localhost:5432/mio_db";
String user = "admin";
String pwd = "password123";

try {
```

```
    // 2. Chiediamo la connessione al DriverManager
    Connection conn = DriverManager.getConnection(url, user, pwd);

    System.out.println("Connessione aperta!");

    // ... usa la connessione ...

    // 3. È FONDAMENTALE chiudere la connessione
    conn.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

Listing 30.1: Ottenere una Connessione (Il modo classico)

30.1.2 La JDBC URL: La chiave di tutto

Il DriverManager capisce quale driver attivare analizzando esclusivamente la stringa di connessione. La struttura è standardizzata:

`jdbc:<sottoprotocollo>:<altri-dati>`

- **jdbc:** Prefisso fisso.
- **sottoprotocollo:** Identifica il database (es. `mysql`, `postgresql`, `oracle:thin`, `h2`).
- **altri-dati:** Indirizzo IP, porta, nome DB e parametri extra.

Deep Dive: Il mito del `Class.forName()`

Se guardi tutorial vecchi (pre-Java 6), troverai spesso questa riga prima della connessione:

```
Class.forName("com.mysql.cj.jdbc.Driver"); // Legacy!
```

Questo serviva per caricare forzatamente la classe del driver in memoria. Oggi non serve più: grazie al meccanismo **SPI (Service Provider Interface)** di Java, il DriverManager scansiona automaticamente i file `.jar` nel classpath alla ricerca dei driver disponibili.

30.1.3 Limiti del DriverManager: Perché serve un Pool?

Usare il `DriverManager` direttamente nel codice di produzione di un'applicazione web è considerato una **bad practice**.

Colloquio: Perché non usare DriverManager in produzione?

Domanda: "Se DriverManager funziona, perché usiamo librerie come HikariCP o il DataSource di Spring?"

Risposta: Per questioni di **Performance** e **Latenza**.

1. **Handshake Costoso:** Creare un oggetto `Connection` fisico richiede l'apertura di un socket TCP, l'autenticazione col DB e l'allocazione di buffer. È un'operazione lenta (può richiedere dai 50ms ai 500ms).
2. **Nessun Riutilizzo:** Il DriverManager crea una *nuova* connessione ogni volta e la chiude alla fine. Se hai 1000 utenti al secondo, ucciderai il database con 1000 aperture/chiusure.

Soluzione: Il **Connection Pooling** (DataSource). Il Pool tiene aperte, ad esempio, 10 connessioni "calde". Quando l'app chiede una connessione, il Pool ne "presta" una

già pronta. Quando l'app fa `close()`, la connessione non viene chiusa realmente, ma restituita al Pool per essere riusata.

30.2 L'Oggetto Connection: Il Canale Fisico

L'interfaccia `java.sql.Connection` non è un semplice oggetto Java che vive nell'Heap: è un "puntatore" a una risorsa esterna, un **Socket TCP** aperto verso il server del database. Finché l'oggetto `Connection` è vivo ("open"), c'è un cavo virtuale dedicato che collega la tua JVM al DB.

Le sue responsabilità principali sono due:

1. **Factory di Statement**: È la fabbrica che crea gli oggetti per eseguire le query (`createStatement`, `prepareStatement`).
2. **Gestore della Transazione**: Controlla quando le modifiche diventano permanenti.

30.2.1 Il Controllo della Transazione (AutoCommit)

In JDBC, il comportamento di default è spesso controintuitivo per chi viene dal mondo Enterprise. Ogni singola connessione nasce in modalità **AutoCommit = true**.

Significa che ogni singola query SQL (INSERT, UPDATE) viene trattata come una transazione autonoma e committata immediatamente. Non c'è modo di fare rollback se la query successiva fallisce.

Per gestire una transazione atomica (più operazioni insieme), dobbiamo disabilitare questo automatismo:

```
Connection conn = DriverManager.getConnection(url, user, pwd);

try {
    // 1. DISABILITIAMO l'automatismo
    // Ora inizia ufficialmente la transazione
    conn.setAutoCommit(false);

    // 2. Eseguiamo varie operazioni
    // (codice per insert 1...)
    // (codice per insert 2...)

    // 3. Se arriviamo qui senza errori, confermiamo tutto
    conn.commit();

} catch (SQLException e) {
    // 4. Se qualcosa va storto, annulliamo tutto
    conn.rollback();
} finally {
    conn.close();
}
```

Listing 30.2: Gestione Transazionale Manuale con JDBC

Deep Dive: Isolation Levels

Attraverso l'oggetto `Connection` possiamo anche definire quanto la nostra transazione deve essere isolata dalle modifiche altrui.

```
conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```

Questo è il meccanismo di basso livello che Hibernate usa quando configuriamo l'isolamento su `@Transactional`.

30.2.2 Approfondimento: I Livelli di Isolamento

Quando più utenti accedono contemporaneamente al database, si verifica un compromesso fondamentale tra **Performance** (velocità) e **Coerenza** (precisione dei dati). L'oggetto `Connection` ci permette di scegliere questo bilanciamento tramite il metodo `setTransactionIsolation(int level)`.

Più l'isolamento è alto, più i dati sono sicuri, ma il database diventa lento (a causa dei lock). Più è basso, più il database è veloce, ma rischiamo di leggere dati incoerenti.

Le Anomalie di Concorrenza

Per capire i livelli di isolamento, dobbiamo prima definire i tre "mostri" che cercano di combattere:

Dirty Read (Lettura Sporca)
Avviene quando leggi un dato che un'altra transazione ha modificato ma **non ha ancora committato**. Se l'altra transazione fa Rollback, tu hai letto un dato che non è mai esistito realmente.

Non-Repeatable Read (Lettura Non Ripetibile)
Avviene quando leggi la **stessa riga** due volte nella stessa transazione e ottieni valori diversi. *Esempio:* Leggi "Saldo: 100". Nel frattempo qualcuno fa commit di un prelievo. Rileggi la stessa riga: "Saldo: 50".

Phantom Read (Lettura Fantasma)
Avviene quando esegui la stessa query di ricerca (es. "utenti di Roma") due volte e ottieni un **numero di righe diverso**. Non sono cambiati i dati delle righe esistenti, ma sono apparse (INSERT) o sparite (DELETE) intere righe.

I 4 Livelli Standard JDBC

La tabella seguente mostra quali anomalie sono permesse (Sì) e quali sono prevenute (No) in ogni livello.

Livello JDBC	Dirty	Non-Rep.	Phantom	Note e Utilizzo
`READ_UNCOMMITTED`	**Sì**	**Sì**	**Sì**	Performance massima, ma dati inaffidabili. Usato raramente.
`READ_COMMITTED`	**No**	**Sì**	**Sì**	**Default** in PostgreSQL, Oracle, SQL Server. Ottimo compromesso.
`REPEATABLE_READ`	**No**	**No**	**Sì**	**Default** in MySQL. Blocca le modifiche alle righe lette.
`SERIALIZABLE`	**No**	**No**	**No**	Esegue le transazioni in serie. Lento, alto rischio di Deadlock.

Tabella 30.1: Matrice dei Livelli di Isolamento (Sì = Anomalia permessa)

Configurazione Programmatica

Ecco come impostare il livello di isolamento direttamente sulla connessione JDBC prima di avviare operazioni critiche.

```
Connection conn = dataSource.getConnection();

// Best Practice: Salvare il livello precedente per ripristinarlo dopo
int oldLevel = conn.getTransactionIsolation();

try {
    // Settiamo il livello PARANOICO per operazioni finanziarie
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
    conn.setAutoCommit(false); // Inizio Transazione

    // ... operazioni di business critiche ...
    // Qui siamo sicuri che nessuno può inserire/modificare dati
    // che interferiscono con la nostra vista

    conn.commit();

} catch (SQLException e) {
    conn.rollback();
    throw e;
} finally {
    // Ripristino dello stato originale (fondamentale per il Pooling!)
    conn.setTransactionIsolation(oldLevel);
    conn.close();
}
```

Listing 30.3: Cambiare l'Isolation Level in JDBC

Colloquio: Differenza tra Non-Repeatable e Phantom?

Domanda: "Spesso si confondono. Qual è la differenza tecnica tra Non-Repeatable Read e Phantom Read?"

Risposta:

- **Non-Repeatable Read** riguarda le **modifiche (UPDATE)** su righe esistenti che hai già letto. Il lock sulla singola riga può prevenirlo.
- **Phantom Read** riguarda l'apparizione di **nuove righe (INSERT)** o la sparizione (DELETE) che soddisfano una clausola `WHERE`.

Metafora: Il Non-Repeatable Read è come rileggere un libro e trovare una parola cambiata. Il Phantom Read è come rileggere un libro e trovare una pagina in più.

30.2.3 Chiudere le risorse: Il "Try-with-resources"

Poiché la `Connection` (e gli Statement che ne derivano) occupa risorse di rete e memoria nel database, non chiuderla è un peccato capitale. Prima di Java 7, il blocco `finally` era obbligatorio e verboso.

Oggi, poiché `Connection` implementa l'interfaccia `AutoCloseable`, usiamo il **try-with-resources**. Java chiuderà automaticamente la connessione alla fine del blocco, anche in caso di eccezione.

```
String sql = "SELECT * FROM utenti WHERE id = ?";

```

```java
// Le risorse aperte nelle parentesi tonde vengono chiuse automaticamente
try (Connection conn = dataSource.getConnection();
     PreparedStatement ps = conn.prepareStatement(sql)) {

    ps.setInt(1, 100);

    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            System.out.println(rs.getString("nome"));
        }
    }
    // Qui rs.close() è chiamato automaticamente
} catch (SQLException e) {
    e.printStackTrace();
}
// Qui ps.close() e conn.close() sono chiamati automaticamente
```

Listing 30.4: Pattern Moderno (Java 7+)

Colloquio: Cos'è un Connection Leak?

Domanda: "Cosa succede se il programmatore dimentica di chiamare `conn.close()` e non usa il try-with-resources?"

Risposta: Si verifica un **Connection Leak** (perdita di connessione). L'oggetto Java viene distrutto dal Garbage Collector, ma il database vede la connessione TCP ancora attiva dall'altra parte. Dopo un po' di tempo, il database esaurirà il numero massimo di connessioni disponibili (spesso default 100 o 150) e l'intera applicazione smetterà di funzionare, rifiutando nuove richieste con errori di timeout.

30.3 Statement vs PreparedStatement: Sicurezza e Performance

Una volta ottenuta una `Connection`, abbiamo bisogno di un "veicolo" per trasportare la nostra query SQL verso il database. JDBC ci offre due interfacce principali: `Statement` e `PreparedStatement`.

Scegliere quella sbagliata non è solo una questione di stile: può compromettere la sicurezza dell'intero sistema.

30.3.1 Lo Statement (e il pericolo della concatenazione)

L'interfaccia `Statement` è la versione base. Si usa per eseguire query SQL statiche. Il problema nasce quando dobbiamo inserire parametri dinamici (es. un nome utente inserito in un form). Con lo `Statement`, siamo costretti a usare la **concatenazione di stringhe**.

```java
Statement stmt = conn.createStatement();
String userInput = "Mario";

// CONCATENAZIONE DIRETTA: PERICOLO!
String sql = "SELECT * FROM utenti WHERE nome = '" + userInput + "'";

ResultSet rs = stmt.executeQuery(sql);
```

Listing 30.5: Statement: Il modo pericoloso

Il disastro della SQL Injection

Cosa succede se un utente malintenzionato, invece di scrivere `"Mario"`, inserisce nel form di login questa stringa?

```
Mario' OR '1'='1
```

La query risultante concatenata diventerà:

```
SELECT * FROM utenti WHERE nome = 'Mario' OR '1'='1'
```

Poiché `'1'='1'` è sempre vero, il database restituirà **tutti gli utenti** della tabella, permettendo al pirata di loggarsi come amministratore senza sapere la password. Questo attacco si chiama **SQL Injection**.

30.3.2 Il PreparedStatement: La Soluzione

Il `PreparedStatement` (estensione di Statement) risolve il problema alla radice separando la struttura della query dai dati. Invece di concatenare stringhe, usiamo dei **segnaposto** (placeholder) indicati dal punto di domanda `?`.

```
String sql = "SELECT * FROM utenti WHERE nome = ?"; // Notare il ?

PreparedStatement ps = conn.prepareStatement(sql);

// Il driver si occupa di fare l'escaping corretto dei caratteri speciali
ps.setString(1, "Mario' OR '1'='1");

ResultSet rs = ps.executeQuery();
```

Listing 30.6: PreparedStatement: Il modo sicuro

In questo caso, il database cercherà letteralmente un utente che si chiama *"Mario' OR '1'='1"*. Ovviamente non lo troverà e l'attacco fallirà.

Colloquio: Perché PreparedStatement è più veloce?

Domanda: "Oltre alla sicurezza, c'è un motivo di performance per preferire PreparedStatement?"

Risposta: Sì, grazie al **Database Plan Caching**. Quando il DB riceve una query, deve:

1. Fare il Parsing (controllare la sintassi).
2. Creare l'Execution Plan (decidere quali indici usare).

Con uno `Statement` classico, ogni query con parametri diversi (es. `WHERE id=1`, `WHERE id=2`) è vista come una query **diversa**. Il DB deve ricompilarla ogni volta.

Con un `PreparedStatement`, la struttura `WHERE id = ?` è inviata una volta sola. Il DB la compila e la mette in cache. Le esecuzioni successive inviano solo i nuovi parametri, saltando la fase di compilazione.

30.3.3 Recuperare i Dati: Il ResultSet

Sia `Statement` che `PreparedStatement` restituiscono un `ResultSet` quando si esegue una `executeQuery()`.

Il ResultSet non contiene tutti i dati in memoria (immagina di selezionare 1 milione di righe!). È un **cursore** che punta a una riga alla volta.

```java
try (ResultSet rs = ps.executeQuery()) {
    // rs.next() sposta il cursore alla riga successiva.
    // Restituisce false quando non ci sono piu' righe.
    while (rs.next()) {
        // Recupero per nome colonna (più leggibile)
        String nome = rs.getString("nome");

        // Recupero per indice (parte da 1, leggermente più veloce)
        int eta = rs.getInt(2);

        System.out.println(nome + " - " + eta);
    }
}
```

Listing 30.7: Iterare su un ResultSet

Deep Dive: Attenzione all'Indice 1

In Java (array, liste) gli indici partono da 0. In JDBC (`PreparedStatement` e `ResultSet`), gli indici partono da **1**. Scrivere `ps.setString(0, "...")` lancerà una `SQLException`.

30.4 Il ResultSet: Navigare i Risultati

Quando eseguiamo una query `SELECT`, il database non ci restituisce immediatamente tutti i dati (che potrebbero essere gigabyte), ma ci restituisce un **ResultSet**.

Il `ResultSet` non è una `List` o un array che contiene i dati. È un **Cursore Attivo** mantenuto aperto tra la nostra applicazione e il database. Immaginalo come un puntatore che, inizialmente, è posizionato *prima* della prima riga (header).

30.4.1 Navigazione e Estrazione

Il metodo principale è `rs.next()`. Esegue due operazioni:

1. Sposta il cursore alla riga successiva.
2. Restituisce `true` se c'è una riga valida, `false` se siamo arrivati alla fine.

```java
// Il cursore parte "Before First"
try (ResultSet rs = stmt.executeQuery("SELECT nome, eta FROM utenti")) {

    while (rs.next()) {
        // Ora il cursore è su una riga valida

        // Estrazione per Nome Colonna (Consigliato per leggibilità)
        String nome = rs.getString("nome");

        // Estrazione per Indice (1-based, leggermente più performante)
        int eta = rs.getInt(2);

        System.out.println(nome + " ha " + eta + " anni");
    }
}
```

Listing 30.8: Iterazione standard su ResultSet

30.4.2 La trappola dei NULL: Primitivi vs Wrapper

Una delle fonti di bug più insidiose in JDBC riguarda la lettura di colonne numeriche (INT, FLOAT) che permettono valori `NULL` nel database.

In Java, i tipi primitivi (`int`, `double`) **non possono essere null**. Se nel DB hai una colonna `eta = NULL` e chiami `rs.getInt("eta")`, JDBC non lancerà un'eccezione (come farebbe un NullPointerException), ma restituirà silenziosamente il valore di default: **0**.

Questo crea una pericolosa ambiguità: quel valore **0** significa che l'utente è un neonato (età 0) o che non ha inserito l'età (NULL)?

Soluzione 1: Il controllo manuale (Legacy)

Il metodo classico prevede di controllare subito dopo la lettura se l'ultimo valore letto era nullo.

```
int valoreLetto = rs.getInt("eta"); // Restituisce 0 se NULL

Integer eta; // Usiamo il Wrapper per permettere il null

if (rs.wasNull()) {
    eta = null; // Era davvero NULL nel DB
} else {
    eta = valoreLetto; // Era davvero 0 (neonato)
}
```

Listing 30.9: Gestione manuale con wasNull()

Soluzione 2: Usare i Wrapper e getObject (Consigliata)

Da Java 7 in poi, possiamo evitare questo codice verboso chiedendo al ResultSet di restituirci direttamente un oggetto **Wrapper**. Il metodo `getObject` con il tipo specificato gestisce automaticamente la conversione: se il DB ha `NULL`, restituisce `null`; altrimenti restituisce l'oggetto popolato.

```
// Restituisce un Integer (che può essere null), non un int
Integer eta = rs.getObject("eta", Integer.class);

if (eta == null) {
    System.out.println("Età non specificata");
} else {
    System.out.println("Età: " + eta);
}
```

Listing 30.10: Lettura diretta tramite Wrapper

Regola d'oro: Se una colonna del database è `nullable`, mappala sempre nel codice Java con la corrispondente classe Wrapper (`Integer`, `Long`, `Double`, `Boolean`) invece del primitivo.

Deep Dive: ResultSet Scrollabili e Aggiornabili

Di default, un ResultSet è **Forward-Only** (puoi solo andare avanti) e **Read-Only**. Tuttavia, configurando lo Statement, possiamo creare ResultSet avanzati:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, // Possiamo fare rs.previous()
    ResultSet.CONCUR_UPDATABLE         // Possiamo modificare i dati
);
```

Con un ResultSet aggiornabile, possiamo modificare il DB senza scrivere SQL:

```
rs.absolute(5);          // Vai alla riga 5
rs.updateString("nome", "Luigi"); // Modifica colonna
rs.updateRow();          // Commit della riga
```

Nota: Queste funzionalità sono usate raramente nelle web app moderne per via dell'overhead di memoria.

30.4.3 L'evoluzione: Il RowSet (Disconnected)

Abbiamo detto che il `ResultSet` è legato a doppio filo alla connessione: se chiudi la `Connection`, il ResultSet muore e non puoi più leggere i dati. Questo è un problema se vogliamo passare i dati a un'altra parte dell'applicazione (es. una GUI Swing o una pagina web) senza tenere il database bloccato.

Qui entra in gioco il `javax.sql.RowSet`, e in particolare la sua implementazione più famosa: il CachedRowSet.

Il `CachedRowSet` funziona in modalità Disconnected (Disconnessa):

1. Si connette al DB ed esegue la query.
2. Copia **tutti** i dati in memoria (RAM).
3. Chiude immediatamente la connessione al DB.
4. Permette all'applicazione di navigare, modificare e scorrere i dati offline.

```
// 1. Creazione tramite Factory (Java 7+)
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();

// 2. Configurazione (contiene al suo interno i dati di connessione)
crs.setUrl("jdbc:mysql://localhost:3306/mydb");
crs.setUsername("user");
crs.setPassword("pass");
crs.setCommand("SELECT * FROM utenti WHERE eta > ?");
crs.setInt(1, 18);

// 3. Esecuzione: Scarica i dati e CHIUDE la connessione fisica
crs.execute();

// --- Qui la connessione al DB è già chiusa! ---

// Possiamo passare 'crs' in giro per l'app
while (crs.next()) {
System.out.println(crs.getString("nome"));
}
```

Listing 30.11: Uso di CachedRowSet (Disconnesso)

Deep Dive: Perché oggi si usano i DTO e non i RowSet?

Se il RowSet sembra così comodo (dati in memoria, scollegati dal DB), perché nelle applicazioni moderne (Spring/Hibernate) usiamo le `List<User>` (DTO/Entity) invece dei RowSet?

Risposta:

- **Performance:** Il `RowSet` è un oggetto pesante e complesso, pieno di metadati JDBC. Una `List` di oggetti Java puri (POJO) è molto più leggera per la RAM.

- **Astrazione:** Il `RowSet` costringe il resto dell'applicazione a conoscere concetti SQL (eccezioni `SQLException`). Convertendo in Oggetti/DTO, nascondiamo completamente la natura del database al resto del codice.

Colloquio: ResultSet vs RowSet vs List

Domanda: "Perché spesso convertiamo subito il ResultSet in una `List<Oggetto>`?"

Risposta: Il `ResultSet` è legato alla connessione fisica.

1. Finché scorri il ResultSet, la `Connection` deve restare occupata. Non puoi passarlo alla View (HTML/JSP) per il rendering, perché la connessione sarebbe già chiusa o bloccherebbe il pool.
2. Trasformando i dati in una `List` di oggetti (DTO o Entity), ci "disaccoppiamo" dal database (Pattern DAO/Repository) e possiamo chiudere la connessione immediatamente.

Capitolo 31

Hibernate Internals: Architettura e Lifecycle

Molti sviluppatori usano Hibernate come una "scatola nera": annotano le classi, chiamano `save()` e sperano che funzioni. Questo approccio funziona finché non incontri una `LazyInitializationException`, un `NonUniqueObjectException` o un aggiornamento involontario sul database.

Per padroneggiare Hibernate, bisogna smettere di pensare in termini di "Query SQL" e iniziare a pensare in termini di **Gestione degli Stati**.

31.1 JPA vs Hibernate: Facciamo ordine

Prima di entrare nel tecnico, chiariamo la terminologia.

Colloquio: Differenza tra JPA e Hibernate

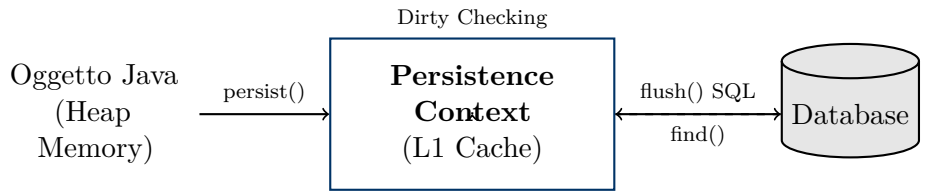
- **JPA (Java Persistence API):** È la **Specifica**. È un documento PDF che definisce le interfacce (es. `EntityManager`) e le annotazioni (es. `@Entity`). Non fa nulla da solo.
- **Hibernate:** È l'**Implementazione**. È il motore che scarichi nel progetto. Rispetta la specifica JPA ma aggiunge funzionalità proprietarie (es. la L2 Cache avanzata).

Analogia: JPA sta a Hibernate come l'interfaccia `List` sta alla classe `ArrayList`.

31.2 L'EntityManager e il Persistence Context

In JDBC, l'oggetto principale era la `Connection`. In JPA, il protagonista è l'**EntityManager**.

L'EntityManager non è solo un passacarte verso il DB. Gestisce il **Persistence Context**, che agisce come una **Cache di Primo Livello (L1)**.



Deep Dive: Perché la L1 Cache è importante?

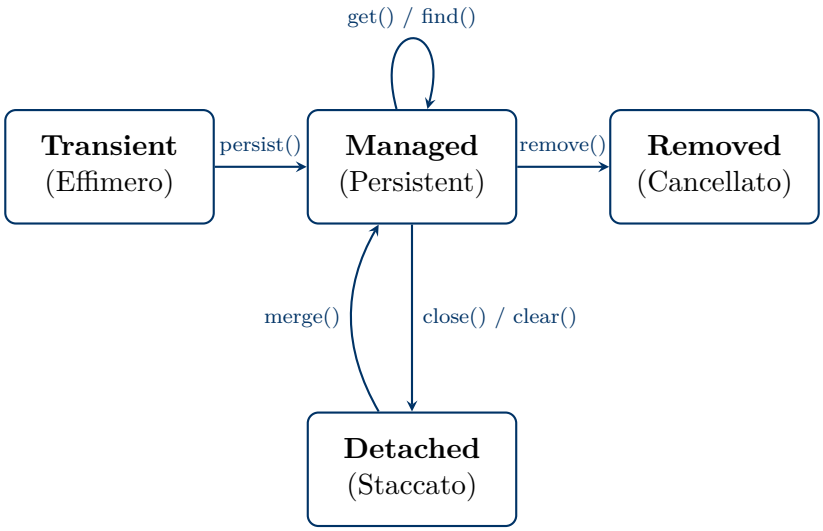
La L1 Cache garantisce l'**Identity Map Pattern**. Se nella stessa transazione chiedi due volte l'utente con ID 1:

```
User u1 = em.find(User.class, 1L); // Select SQL
User u2 = em.find(User.class, 1L); // Nessuna SQL!
System.out.println(u1 == u2); // TRUE: Sono lo stesso identico oggetto in
    ↪ memoria
```

Senza questo meccanismo, avresti due oggetti Java diversi che rappresentano la stessa riga DB, creando conflitti di dati.

31.3 Entity Lifecycle: La Macchina a Stati

In JPA, un oggetto non è semplicemente "nel DB" o "non nel DB". Vive in uno di 4 stati ben precisi. Capire questi stati è la differenza tra un Junior e un Senior.



31.3.1 1. Transient (Effimero)

L'oggetto è appena stato creato con `new`.

- Vive solo nella memoria Heap di Java.
- Hibernate non sa che esiste.
- Non ha un ID (o è null).
- Se il programma termina, l'oggetto svanisce.

31.3.2 2. Managed (Gestito)

L'oggetto è "sotto sorveglianza" nel Persistence Context.

- Ha un ID database.
- Ogni modifica ai suoi campi viene tracciata (Dirty Checking).
- Al `commit()`, le modifiche verranno sincronizzate col DB.

31.3.3 3. Detached (Staccato)

È l'incubo degli sviluppatori. L'oggetto ha un ID (quindi rappresenta una riga DB), ma l'`EntityManager` è stato chiuso o l'oggetto è stato espulso.

- Le modifiche ai set non vengono salvate.
- Se provi ad accedere a collezioni Lazy, ottieni `LazyInitializationException`.

31.3.4 4. Removed

L'oggetto è marcato per la cancellazione. La `DELETE` SQL partirà al prossimo flush.

31.4 Le Transizioni Critiche

31.4.1 persist() vs merge()

- **persist(entity):** Prende un oggetto *Transient* e lo rende *Managed*. Se l'oggetto aveva già un ID generato manualmente, potrebbe lanciare eccezione.
- **merge(entity):** Serve per salvare oggetti *Detached*. **Attenzione:** `merge` non "riattacca" l'oggetto che gli passi. Ne crea una copia, copia i dati, salva la copia e restituisce la copia.

```
User u = new User("Mario");
u.setId(1L); // Detached (ha ID ma non è in sessione)

// SBAGLIATO
em.merge(u);
u.setName("Luigi"); // Modifica PERSA! 'u' è ancora Detached.

// CORRETTO
User managedU = em.merge(u);
managedU.setName("Luigi"); // Modifica salvata.
```

La trappola del merge

31.5 Il Dirty Checking: L'Update Automatico

Questa è la funzionalità più potente e pericolosa di Hibernate.

Quando un oggetto è **Managed**, Hibernate mantiene uno **Snapshot** (una copia segreta) dei suoi dati originali al momento del caricamento. Quando fai `flush()` (o commit), Hibernate confronta l'oggetto attuale con lo Snapshot.

Se trova differenze, lancia l'`UPDATE` SQL automaticamente. **Non serve chiamare save()!**

```
@Transactional
public void updatePassword(Long userId, String newPass) {
    // 1. Caricamento (Diventa Managed)
    User u = repository.findById(userId).orElseThrow();

    // 2. Modifica dello stato
    u.setPassword(newPass);

    // 3. Fine Metodo -> Commit Automatico
    // Hibernate vede la differenza e fa UPDATE users SET password=...
}
```

Dirty Checking in Azione

Colloquio: Insidia del Dirty Checking

Domanda: Ho caricato un'entità solo per calcolare un dato e stamparlo a video, ho modificato un campo ma non volevo salvarlo. Perché il DB è cambiato?

Risposta: Perché l'entità era **Managed** e la transazione ha fatto commit. **Soluzione:** Usa `em.detach(entity)` se vuoi modificarlo senza salvare, oppure usa DTO (che non sono managed), oppure usa transazioni `readOnly=true`.

Capitolo 32

Advanced Mapping Strategy

Se il Capitolo 1 riguardava il "Motore" (EntityManager), questo riguarda la "Carrozzeria". Mappare oggetti Java su tabelle relazionali è un'operazione piena di compromessi. Un mapping sbagliato può rendere il database inconsistente o le query disastrosamente lente.

In questo capitolo definiremo il "Contratto" tra Java e SQL, esplorando le relazioni e le strategie di ereditarietà.

32.1 Il Contratto Entity-Database

Non basta annotare una classe con `@Entity`. Bisogna definire con precisione i vincoli.

32.1.1 Strategie di Generazione delle Chiavi Primarie (Primary Keys)

Per decenni, lo standard è stato utilizzare un numero intero progressivo (`Long id`) che si incrementa da solo (1, 2, 3...). Sebbene sia semplice e performante, questa scelta comporta dei rischi che un architetto moderno deve considerare:

- **Sicurezza e Business Intelligence:** Se mi iscrivo al tuo sito e ottengo l'ID 1000, e il giorno dopo un mio amico si iscrive e ottiene 1050, so esattamente che hai avuto 50 nuovi utenti. È un'informazione riservata che stai esponendo pubblicamente nell'URL (es. `/users/1000`).
- **Merge dei Dati:** Se domani devi unire due database diversi (es. acquisizione aziendale), avrai sicuramente conflitti perché l'ID "1" esiste in entrambi i database per persone diverse.

Hibernate (JPA) offre diverse strategie per generare le chiavi, configurabili tramite l'annotazione `@GeneratedValue`. Vediamole nel dettaglio:

1. GenerationType.IDENTITY (Auto-Increment)

È la strategia classica di MySQL (`AUTO_INCREMENT`). Il database assegna l'ID **dopo** aver inserito la riga.

- **Come funziona:** Hibernate invia la `INSERT`, il DB salva, genera l'ID e lo restituisce a Hibernate.
- **Controindicazione (Importante):** Disabilita il **Batch Insert**. Poiché Hibernate deve conoscere l'ID dell'oggetto *prima* di poterlo associare ad altri oggetti in memoria, non può raggruppare 100 insert in una volta sola. Deve farle una per una.

2. GenerationType.SEQUENCE (Lo Standard Enterprise)

È la strategia preferita per PostgreSQL e Oracle.

- **Come funziona:** Il database mantiene un contatore separato (Sequence). Hibernate chiede alla sequenza: "Dammi i prossimi 50 ID". Il DB risponde. Hibernate assegna gli ID agli oggetti in memoria e poi fa le INSERT tutte insieme.
- **Vantaggio:** Permette il **Batch Insert** ed è molto performante.

3. GenerationType.UUID (Lo Standard Moderno)

L'ID non è più un numero, ma una stringa esadecimale a 128-bit unica al mondo (es. `550e8400-e29b-41d4-a716-446655440000`).

- **Vantaggi:** Impossibile da indovinare (sicurezza), unico globalmente (merge facile), generabile da Java senza chiedere al Database.
- **Svantaggi:** Occupa più spazio su disco rispetto a un numero (16 byte vs 8 byte) e indicizzarlo è leggermente più lento.

```java
@Entity
@Table(name = "users")
public class User {

    // Hibernate 6 genera automaticamente UUID ottimizzati (RFC 4122)
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    // ESEMPIO DI ALTRI VINCOLI UTILI SULLE COLONNE

    // nullable=false: Obbligatorio (NOT NULL)
    // unique=true: Non possono esserci due username uguali (UNIQUE INDEX)
    // updatable=false: Una volta creato, non può essere modificato
    @Column(nullable = false, unique = true, updatable = false)
    private String username;

    // precision=10, scale=2: FONDAMENTALE per i soldi (BigDecimal).
    // Significa: 10 cifre totali, di cui 2 decimali (es. 12345678.99).
    // Se lo ometti, il DB potrebbe arrotondare e farti perdere centesimi!
    @Column(precision = 10, scale = 2)
    private BigDecimal salary;
}
```

Esempio completo di Mapping con UUID

Strategia	Quando usarla
`IDENTITY`	Progetti semplici, DB MySQL, quando non serve il Batch Insert.
`SEQUENCE`	Progetti Enterprise su PostgreSQL/Oracle con alti volumi di scrittura.
`UUID`	Microservizi, sistemi distribuiti, o quando la sicurezza dell'ID è prioritaria.

32.2 Relazioni Fondamentali: Owner vs Inverse

Questa è la regola numero uno di Hibernate. Nelle relazioni bidirezionali, i due oggetti si puntano a vicenda, ma **solo uno comanda la Foreign Key**.

- **Owner Side (@JoinColumn):** È l'entità che contiene fisicamente la chiave esterna. Hibernate guarda SOLO questo lato per salvare le modifiche.
- **Inverse Side (mappedBy):** È l'altro lato. Serve solo per navigare la relazione in Java.

32.2.1 1. One-to-Many / Many-to-One

Il caso classico: Un Dipartimento, Molti Impiegati.

```
// LATO OWNER (Chi ha la FK 'dept_id')
@Entity
public class Employee {
    @Id private UUID id;

    // SENIOR RULE: Sempre LAZY. Il default è EAGER (Male!)
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "dept_id")
    private Department department;
}

// LATO INVERSE (Chi subisce il riferimento)
@Entity
public class Department {
    @Id private UUID id;

    // mappedBy punta al campo 'department' nella classe Employee
    @OneToMany(mappedBy = "department",
               cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<Employee> employees = new ArrayList<>();
}
```

Configurazione Corretta

32.2.2 La Trappola Bidirezionale (Helper Methods)

Se fai `dept.getEmployees().add(emp)` e salvi, **non succederà nulla nel DB**. Perché? Perché hai aggiornato il lato Inverse (che Hibernate ignora in scrittura). Devi aggiornare il lato Owner (`emp.setDepartment(dept)`).

Deep Dive: Helper Methods: Sincronizzazione

Per evitare bug, incapsula la logica nell'entità Padre:

```
// In Department.java
public void addEmployee(Employee employee) {
    this.employees.add(employee);
    employee.setDepartment(this); // Sincronizzo l'Owner!
}

```

```
public void removeEmployee(Employee employee) {
    this.employees.remove(employee);
    employee.setDepartment(null); // Dissocio
}
```

32.3 Relazioni Avanzate

32.3.1 2. One-to-One (Shared Primary Key)

Relazione Utente ↔ Profilo. Evita di avere una FK in entrambe le tabelle. Usa `@MapsId`. Significa: "La Primary Key del Profilo è ANCHE la Foreign Key verso l'Utente".

```
@Entity
public class UserProfile {
    @Id private UUID userId; // Stesso ID dello User

    @OneToOne(fetch = FetchType.LAZY)
    @MapsId
    @JoinColumn(name = "user_id")
    private User user;
}
```

32.3.2 3. Many-to-Many (Set vs List)

Relazione Studente ↔ Corso. Richiede una tabella di mezzo (*Join Table*).

```
@Entity
public class Student {
    @Id private UUID id;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    // CRUCIALE: Usa SET, mai LIST!
    private Set<Course> courses = new HashSet<>();
}
```

Colloquio: Perché Set e non List?

Domanda: Perché l'uso di `List` è sconsigliato nelle `@ManyToMany`?

Risposta: Per un problema di performance sulle cancellazioni. Se rimuovi un elemento da una `List`, Hibernate cancella **tutte** le righe della join table per quell'ID e reinserisce quelle rimaste (Delete All + Insert N-1). Con `Set`, grazie all'unicità della chiave, Hibernate cancella esattamente la singola riga specifica.

32.4 Value Objects: @Embeddable

Non tutto merita una tabella. Un indirizzo (Via, Città, CAP) ha senso solo se legato a un utente.

```
@Embeddable
public class Address {
    private String street;
    private String city;
}

@Entity
public class User {
    @Id private UUID id;

    @Embedded
    private Address homeAddress; // Colonne: street, city

    // Se ho due indirizzi, devo rinominare le colonne!
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="street", column=@Column(name="work_street")),
        @AttributeOverride(name="city", column=@Column(name="work_city"))
    })
    private Address workAddress;
}
```

32.5 Ereditarietà (Inheritance Strategies)

I DB relazionali non supportano `extends`. Hibernate offre 3 strategie.

32.5.1 1. SINGLE_TABLE (Default)

Unica tabella gigante con una colonna discriminante (`DTYPE`).

- **Pro:** Velocissimo (No Join).
- **Contro:** I campi delle sottoclassi devono essere **Nullable**. Viola l'integrità dei dati.

32.5.2 2. JOINED (Normalizzato)

Tabelle separate (Padre e Figli) collegate da FK.

- **Pro:** Integrità referenziale, Not Null possibili.
- **Contro:** Query lente (Hibernate deve fare JOIN per caricare l'entità completa).

32.5.3 3. TABLE_PER_CLASS

Una tabella per ogni classe concreta, duplicando le colonne del padre.

- **Contro:** Le query polimorfiche (`SELECT v FROM Veicolo v`) usano `UNION ALL`. Pessime performance. Da evitare.

32.6 Riepilogo Best Practices

Situazione	Soluzione Senior
Chiavi Primarie	Usa `UUID` per sistemi nuovi/distribuiti.
Fetch Type	Imposta **SEMPRE** `LAZY` su `@ManyToOne` e `@OneToOne`.
Bidirezionalità	Usa Helper Methods (‘add/remove‘) nel lato Inverse.
ManyToMany	Usa `Set<T>` e sovrascrivi `equals/hashCode` usando la Business Key (non l’ID).
Soldi	Usa `BigDecimal` con `precision` e `scale`.

Capitolo 33

JPQL, HQL e Native Queries

Mappare le entità è solo l'inizio. In un'applicazione reale, non puoi caricare l'intero database in memoria per filtrarlo con Java stream. Devi saper chiedere al Database esattamente ciò che ti serve.

Esistono tre linguaggi per parlare con Hibernate:

1. **JPQL (Java Persistence Query Language):** Lo standard JPA. Opera su *Entità* e *Attributi*. È portabile (funziona su MySQL, Postgres, Oracle).
2. **HQL (Hibernate Query Language):** Il dialetto proprietario di Hibernate. È un superset di JPQL (ha funzioni in più).
3. **Native SQL:** SQL puro. Opera su *Tabelle* e *Colonne*. Dipende dal database specifico.

33.1 Sintassi JPQL: Pensare a Oggetti

La differenza mentale fondamentale è che in JPQL non esistono le tabelle. Esistono solo le classi.

```
// SQL (Database centrico)
SELECT * FROM users u WHERE u.email_address = 'test@test.com';

// JPQL (Oggetto centrico)
// 'User' è la classe Java, 'email' è il campo Java (non la colonna!)
SELECT u FROM User u WHERE u.email = :email
```

SQL vs JPQL

Deep Dive: Case Sensitivity

In JPQL/HQL:

- Le keyword (`SELECT`, `FROM`, `WHERE`) sono **case-insensitive**.
- I nomi delle Entità (`User`) e degli Attributi (`email`) sono **CASE-SENSITIVE**.

Scrivere `FROM user` (minuscolo) causerà `QuerySyntaxException` se la classe si chiama `User`.

33.2 Parametri Named: Sicurezza

Mai concatenare stringhe nelle query. È la porta principale per la **SQL Injection**. Usa sempre i parametri nominati (Named Parameters).

```
// Esempio con Spring Data @Query
@Query("SELECT u FROM User u WHERE u.status = :status AND u.age > :minAge")
```

```
List<User> findByStatusAndAge(
    @Param("status") UserStatus status,
    @Param("minAge") int minAge
);
```

33.3 Proiezioni DTO: Il Performance Booster

Questo è il trucco che distingue i Senior. Caricare un'Entity è costoso: Hibernate deve gestire il proxy, lo snapshot per il dirty checking e la cache L1. Se devi solo mostrare una lista di nomi e email in una tabella, **non caricare le Entity**. Proietta su un DTO (o Record).

33.3.1 La keyword 'new'

Puoi istanziare oggetti Java direttamente dentro la query.

```
// Java Record (DTO leggero)
public record UserSummary(String username, String email) {}

// Query Ottimizzata
@Query("""
    SELECT new com.example.dto.UserSummary(u.username, u.email)
    FROM User u
    WHERE u.active = true
""")
List<UserSummary> findActiveSummaries();
```

Costruzione DTO in Query

Vantaggi:

- **No Overhead:** Nessun oggetto entra nel Persistence Context.
- **No Dirty Checking:** Hibernate non traccia modifiche (risparmio CPU/RAM).
- **Network:** Scarichi dal DB solo le colonne che servono, non `SELECT *`.

33.4 Native Query e Raw Mapping

A volte JPQL non basta. Se devi usare *Window Functions*, *Common Table Expressions (CTE)*, o funzioni specifiche JSONB di Postgres, devi usare SQL Nativo.

33.4.1 Mapping su Interfaccia (Spring Projection)

Il modo più pulito per mappare il risultato di una Native Query senza creare classi DTO manuali.

```
// 1. Definisci un'interfaccia con i getter che corrispondono agli ALIAS SQL
public interface SalesReport {
    String getCategory();     // corrisponde a c.name
    Double getTotalAmount();  // corrisponde a SUM(o.amount)
}

// 2. Scrivi la query Nativa
@Query(value = """
    SELECT c.name as category, SUM(o.amount) as totalAmount
    FROM orders o
    JOIN categories c ON o.cat_id = c.id
```

```
    GROUP BY c.name
""", nativeQuery = true)
List<SalesReport> getSalesReport();
```

Deep Dive: JPQL vs Native: Quando usare cosa?

Usa JPQL per il 90% dei casi (CRUD, ricerche, filtri). Mantiene il codice agnostico dal DB. Usa Native Query solo per reportistica complessa o ottimizzazioni estreme dove l'SQL generato da Hibernate è inefficiente.

33.5 Modifying Queries (Bulk Update/Delete)

Aggiornare 10.000 righe caricando le entity una a una è follia (10.000 SELECT + 10.000 UPDATE). Si usa JPQL per fare un'unica UPDATE massiva.

```
@Modifying(clearAutomatically = true) // FONDAMENTALE
@Query("UPDATE User u SET u.active = false WHERE u.lastLogin < :cutoffDate")
int deactivateInactiveUsers(@Param("cutoffDate") LocalDate cutoffDate);
```

Colloquio: Perché clearAutomatically

Domanda: Cosa succede se non metto `clearAutomatically = true`?

Risposta: Inconsistenza dei dati. Le query `@Modifying` bypassano la cache di primo livello (L1) e scrivono direttamente sul DB. Immagina di aver caricato l'utente Mario (active=true) in memoria. Poi lanci la query di update che mette active=false sul DB. Se non pulisci la cache, l'oggetto Java `mario` in memoria rimarrà `active=true`. Le logiche successive lavoreranno su dati vecchi. `clearAutomatically` svuota il contesto, costringendo Hibernate a ricaricare i dati freschi alla prossima richiesta.

33.6 Pagination e Sorting

Non scrivere mai `LIMIT` e `OFFSET` a mano in JPQL. Spring Data e JPA gestiscono la paginazione in modo standard.

```
// Nel Repository
@Query("SELECT u FROM User u WHERE u.dept = :dept")
Page<User> findByDept(@Param("dept") String dept, Pageable pageable);

// Nel Service
PageRequest pageRequest = PageRequest.of(0, 10, Sort.by("username"));
Page<User> page = repository.findByDept("IT", pageRequest);
```

33.7 Riepilogo Best Practices

Scenario	Soluzione Senior
Lettura Dati per UI (Read-Only)	Usa **DTO Projections** (keyword `new`). Evita di caricare Entity.
Query Complessa / DB Specific	Usa **Native Query** con Mapping su Interfaccia.
Aggiornamento Massivo	Usa `@Modifying` con `clearAutomatically=true`.
Parametri Query	Usa sempre `:paramName` (mai concatenazione stringhe).
Paginazione	Usa sempre `Pageable`, mai `findAll()`.

Capitolo 34

Transazioni e Concorrenza (JPA Internals)

Fino ad ora abbiamo visto come mappare e interrogare i dati. Ma le applicazioni enterprise sono multi-utente. Cosa succede se due utenti modificano lo stesso record nello stesso millisecondo?

In questo capitolo scendiamo nel livello più basso: la gestione dell'integrità dei dati (ACID), i livelli di isolamento e le strategie di Locking (Ottimistico vs Pessimistico).

34.1 Anatomia di una Transazione (Unit of Work)

Prima di usare la magia di Spring (`@Transactional`), dobbiamo capire cosa succede "a mano". In JPA, la transazione è gestita dall'interfaccia `EntityTransaction`.

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

try {
    // 1. BEGIN: Apre la connessione fisica e disabilita auto-commit
    tx.begin();

    // 2. BUSINESS LOGIC
    BankAccount conto = em.find(BankAccount.class, 1L);
    conto.setBalance(conto.getBalance() - 100); // Dirty Checking attivo

    // 3. FLUSH: Invia le SQL (UPDATE) al DB, ma non conferma
    em.flush();

    // 4. COMMIT: Rende le modifiche permanenti e visibili agli altri
    tx.commit();

} catch (Exception e) {
    // 5. ROLLBACK: Annulla tutto se c'è un errore
    if (tx.isActive()) tx.rollback();
} finally {
    em.close();
}
```

Gestione Manuale (Cosa fa Spring per te)

Deep Dive: Flush vs Commit

È vitale distinguere questi due momenti:

- **Flush():** Sincronizza la memoria Java con il Database. Esegue le istruzioni SQL (`INSERT`, `UPDATE`). I dati sono nel DB, ma sono "sporchi" (bloccati dalla transazione corrente).
- **Commit():** Chiama implicitamente `flush()` e poi invia il comando di commit al DB, rendendo le modifiche definitive e rilasciando i lock.

34.2 Isolation Levels: I Fenomeni di Concorrenza

Il database promette isolamento, ma quanto isolamento? Più isoli, più il sistema è lento. JPA delega questo controllo al database sottostante, ma devi conoscere i rischi.

Livello	Fenomeni Evitati
READ UNCOMMITTED	Nessuno. Leggi dati non ancora committati (Dirty Read). Veloce ma pericoloso.
READ COMMITTED	Evita **Dirty Reads**. Leggi solo dati confermati. È il default di Postgres, Oracle e SQL Server.
REPEATABLE READ	Evita **Non-Repeatable Reads**. Se leggi una riga due volte nella stessa TX, ottieni lo stesso valore. Default di MySQL.
SERIALIZABLE	Evita **Phantom Reads**. Esecuzione sequenziale. Lento ma sicuro al 100%.

Colloquio: Dirty Read vs Phantom Read

Domanda: Qual è la differenza? **Risposta:**

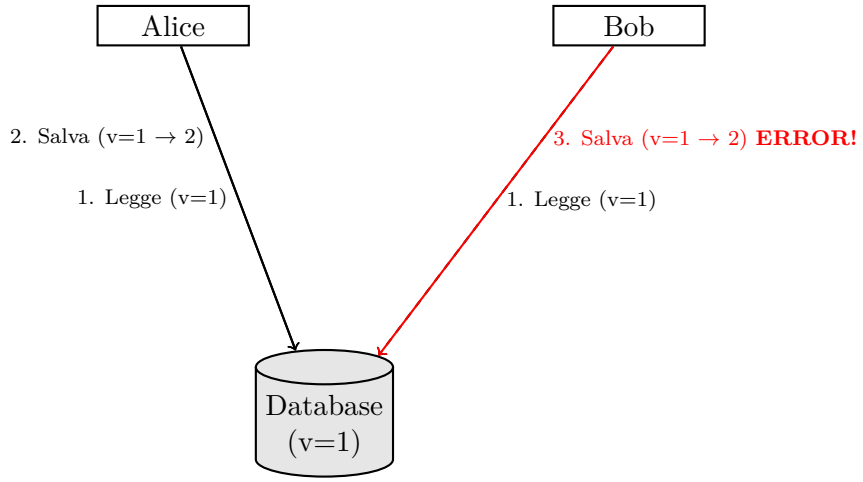
- **Dirty Read:** Leggo un dato modificato da un'altra transazione che poi fa Rollback. Ho letto un dato che "non è mai esistito".
- **Phantom Read:** Eseguo `SELECT COUNT(*)` e ottengo 10. Un altro utente inserisce una riga. Rifaccio la query e ottengo 11. I dati esistenti non sono cambiati, ma sono apparse righe "fantasma".

34.3 Gestione della Concorrenza: Il Locking

Il problema principale nelle web app è il **Lost Update**. *Alice e Bob leggono lo stesso prodotto (Qty: 10). Entrambi vendono 1 unità. Entrambi salvano Qty: 9. Risultato finale: 9 (invece di 8).*

34.3.1 1. Optimistic Locking (@Version)

È la strategia "Web Scale". Non usiamo lock sul database. Usiamo una colonna `version`.




```java
@Entity
public class Product {
    @Id private Long id;

    private int quantity;

    @Version // Gestito automaticamente da Hibernate
    private Long version;
}
```

Quando Bob prova a salvare, Hibernate esegue: `UPDATE product SET quantity=9, version=2 WHERE id=1 AND version=1` Poiché Alice ha già portato la versione a 2, la condizione `WHERE version=1` fallisce (0 righe aggiornate). Hibernate lancia **`OptimisticLockException`**. Spetta al codice applicativo catturare l'eccezione e chiedere a Bob di ricaricare la pagina.

34.3.2 2. Pessimistic Locking

In scenari critici (es. bonifici, biglietti concerto), non possiamo rischiare. Blocchiamo la riga fisicamente.

```java
// JPQL con Lock
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("SELECT p FROM Product p WHERE p.id = :id")
Optional<Product> findByIdWithLock(@Param("id") Long id);
```

Questo genera una SQL: `SELECT ... FOR UPDATE`. Nessun altro potrà leggere o scrivere quella riga finché la transazione corrente non è finita.

Deep Dive: Attenzione ai Deadlock

Il Pessimistic Locking è pericoloso. Se Transazione A blocca Risorsa 1 e vuole Risorsa 2. E Transazione B blocca Risorsa 2 e vuole Risorsa 1. Il sistema si congela (Deadlock). **Best Practice:** Imposta sempre un timeout: `@QueryHints(@QueryHint(name = "javax.persistence.lock.timeout", value = "3000"))`

34.4 Riepilogo Best Practices

Scenario	Soluzione
Web App Standard (CRUD)	**Optimistic Locking** (`@Version`). Scalabilità massima, nessun lock DB.
Operazioni Critiche (Finanza)	**Pessimistic Locking** (`PESSIMISTIC_WRITE`). Sicurezza totale, bassa scalabilità.
Reportistica Lunga	Usa **READ COMMITTED** o snapshot isolation per non bloccare chi scrive.
Gestione Errori	Cattura sempre `OptimisticLockException` nel Service e rilancia un errore leggibile per l'utente ("Dato modificato da altri, ricarica").

Capitolo 35

Hibernate Performance Tuning & Caching

Hibernate è famoso per essere facile da iniziare ma difficile da scalare. Se non configurato correttamente, può generare migliaia di query inutili (N+1 Problem) o consumare tutta la memoria heap.

In questo capitolo vedremo come trasformare un'applicazione lenta in una scheggia, usando strategie di fetching intelligenti e livelli di caching multipli.

35.1 Il Problema N+1 Select: Il Killer Silenzioso

È la causa numero uno dei rallentamenti nelle applicazioni Java.

Scenario: Hai una lista di `Ordini`. Ogni ordine ha una lista di `Articoli`. La relazione è `LAZY` (come raccomandato).

```
// 1. Carica 1000 ordini (1 Query)
List<Order> orders = orderRepository.findAll();

for (Order order : orders) {
    // 2. Accedi alla collezione Lazy
    // Hibernate esegue una query SELECT * FROM items WHERE order_id = ?
    System.out.println("Articoli: " + order.getItems().size());
}
```

Risultato: Hibernate esegue **1001 Query**. 1 per la lista + 1000 per i dettagli di ogni ordine. Il database viene bombardato.

35.1.1 Soluzione 1: JOIN FETCH (JPQL)

Forziamo il caricamento immediato in una sola query.

```
@Query("SELECT o FROM Order o JOIN FETCH o.items")
List<Order> findAllWithItems();
```

Risultato: 1 Query singola con `INNER JOIN`.

35.1.2 Soluzione 2: @EntityGraph (Dichiarativo)

Se non vuoi scrivere JPQL, puoi usare l'annotazione JPA standard.

```java
// Nel Repository
@EntityGraph(attributePaths = {"items", "customer"})
List<Order> findAll();
```

Questo dice a Spring Data: "Esegui il findAll standard, ma fai una Left Join Fetch su items e customer".

35.2 Batch Fetching: La via di mezzo intelligente

A volte `JOIN FETCH` non va bene. Se hai **due** collezioni ('items' e 'historyLogs') e provi a fare `JOIN FETCH` su entrambe, crei un **Prodotto Cartesiano** (le righe si moltiplicano esponenzialmente in memoria). Hibernate lancerà un errore `MultipleBagFetchException`.

Soluzione Senior: `@BatchSize` Manteniamo il caricamento LAZY, ma ottimizzato. Invece di caricare gli articoli uno per uno, li carichiamo a pacchetti.

```java
@Entity
public class Order {
    @Id private Long id;

    @OneToMany(mappedBy = "order")
    @BatchSize(size = 20) // LA MAGIA
    private List<Item> items;
}
```

Cosa succede ora nel loop?

1. Carichi 100 ordini.
2. Accedi al primo `order.getItems()`.
3. Hibernate vede che ci sono altri proxy in memoria e invece di caricare solo l'ID 1, carica gli ID 1, 2, 3... fino a 20.
4. **SQL Generato:** `SELECT * FROM items WHERE order_id IN (?, ?, ... 20 volte)`.

Risultato: Invece di 1000 query, ne fai $1000/20 = 50$ query. Performance ottime senza prodotto cartesiano.

35.3 LazyInitializationException: Il Nemico

Questo errore accade quando accedi a un campo `LAZY` (Proxy) ma la sessione Hibernate è già chiusa.

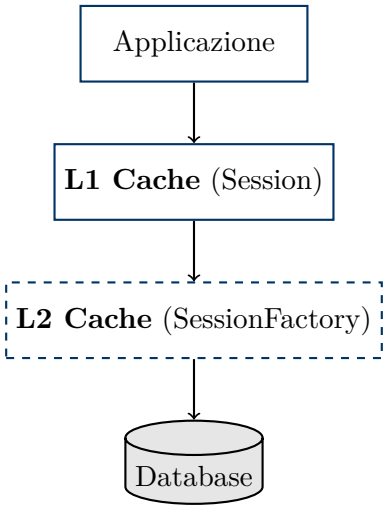
Deep Dive: L'Anti-Pattern "enable_lazy_load_no_trans"

Molti sviluppatori trovano su StackOverflow questa proprietà da mettere nell'`application.properties`: `hibernate.enable_lazy_load_no_trans=true`.

NON FARLO MAI. Questa impostazione dice a Hibernate: "Se la sessione è chiusa, aprine una nuova temporanea, fai la query e richiudila". Trasforma un errore visibile (l'eccezione) in un problema di performance invisibile (N+1 silenzioso). Se hai bisogno dei dati, caricali prima con `JOIN FETCH`.

35.4 Hibernate Caching Architecture

Hibernate ha un'architettura di caching a due livelli.



35.4.1 Level 1 Cache (Session Scope)

- **Abilitata:** Sempre (non disattivabile).
- **Vita:** Dura quanto la transazione.
- **Funzione:** Garantisce che `findById(1)` ritorni lo stesso oggetto Java se chiamato due volte nella stessa transazione.

35.4.2 Level 2 Cache (Global Scope)

- **Abilitata:** No (va configurata).
- **Vita:** Dura quanto l'applicazione. Condivisa tra tutti gli utenti.
- **Provider:** EhCache (locale), Redis (distribuito).
- **Uso:** Ideale per dati "Reference" (es. Regioni, Province, Categorie prodotti) che vengono letti spesso e modificati raramente.

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Product { ... }
```

Abilitare L2 Cache su un'Entity

35.4.3 Query Cache

La L2 Cache salva le *Entità* per ID. Ma se fai una query `SELECT p FROM Product p WHERE p.price > 100`? Hibernate non può usare la L2 Cache perché non conosce gli ID. La **Query Cache** salva i risultati delle query (solo gli ID) basandosi sui parametri di input. **Attenzione:** Se la tabella viene modificata spesso, la Query Cache si invalida continuamente, peggiorando le performance invece di migliorarle.

35.5 Riepilogo Strategie di Performance

Problema	Soluzione Senior
N+1 Select (Singola Relazione)	Usa `JOIN FETCH` in JPQL.
N+1 Select (Multiple Relazioni)	Usa `@BatchSize(size=20)` per evitare il prodotto cartesiano.
LazyInitializationException	Non riaprire la sessione. Progetta meglio la query iniziale caricando ciò che serve.
Dati statici letti spesso	Abilita la **L2 Cache** (es. Redis) con strategia READ_ONLY.
Reportistica pesante	Non usare Hibernate. Usa **Native SQL** o JDBC Template per bypassare l'overhead dell'Entity Manager.

Capitolo 36

DTO: Architettura, Mapping e Proiezioni JPA

Finora abbiamo parlato di come mappare il DB sugli oggetti (Entity). Ma se provi a costruire un'applicazione Enterprise usando solo le Entity, fallirai. Il **Data Transfer Object (DTO)** non è burocrazia: è l'unica barriera che protegge il tuo Database dal mondo esterno.

In questo capitolo vedremo:

- **Il problema strutturale:** Perché le Entity nel Controller corrompono i dati.
- **Il flusso architetturale:** Come avviene lo "Swap" (Entity ↔ DTO) nel Service.
- **JPA Projections:** Come usare i DTO per scrivere query SQL performanti senza caricare intere Entity in memoria.

36.1 Il Design Pattern Architetturale

Prima del codice, devi visualizzare il flusso dei dati. Un'applicazione Java ben fatta a strati funziona così:

Il Flusso dei Dati (Data Flow)

Frontend (JSON) $\xrightarrow{\text{Request}}$ **Controller** $\xrightarrow{\text{DTO}}$ **Service** $\xrightarrow{\text{Entity}}$ **Repository/DB**

Frontend (JSON) $\xleftarrow{\text{Response}}$ **Controller** $\xleftarrow{\text{DTO}}$ **Service** $\xleftarrow{\text{Entity}}$ **Repository/DB**

- Il **Controller** parla solo DTO. Non deve sapere cosa sia una Entity.
- Il **Repository** parla solo Entity (o Projections).
- Il **Service** è il "convertitore": è l'unico posto dove avviene lo **Swap**.

36.2 Il Pericolo: Dirty Checking e Dati Esposti

Perché non posso fare `return repository.findById(id)`? Guardiamo cosa succede "sotto il cofano" di Hibernate quando fai questa operazione ingenua.

```
@GetMapping("/users/{id}")
@Transactional // Spesso la transazione e' aperta fino alla vista (OSIV)
public UserEntity getUser(@PathVariable Long id) {
    // 1. Hibernate carica l'Entity. È in stato "MANAGED".
    // Significa che Hibernate osserva ogni modifica.
    UserEntity user = repository.findById(id).orElseThrow();
```

```
7
8       // 2. Il Dev vuole nascondere la password hashata prima di mandare il JSON
9       user.setPassword("*******");
10
11      // 3. Ritorna l'oggetto. Jackson lo serializza.
12      return user;
13
14      // 4. DISASTRO: Alla fine del metodo, Hibernate fa il "Flush".
15      // Vede che la password in memoria è diversa da quella originale.
16      // Esegue AUTOMATICAMENTE: UPDATE users SET password = '*******' ...
17  }
```

L'errore del Junior Dev: Entity nel Controller

Risultato: Hai appena cancellato la password dell'utente dal Database semplicemente leggendo il suo profilo. Usando un DTO, lavori su una copia scollegata (Detached). Puoi modificarla quanto vuoi, Hibernate non se ne accorgerà mai.

36.3 Implementazione Moderna: Record e MapStruct

Oggi non si scrivono più DTO a mano (troppo codice) né si usano mapper lenti basati su Reflection (ModelMapper). Lo standard industriale è **Java Records** + **MapStruct**.

36.3.1 1. Definizione dei DTO (Java 14+ Records)

I record sono immutabili, veloci e concisi.

```
1  // DTO per la risposta (Output)
2  public record UserResponse(
3      Long id,
4      String username,
5      String email, // Niente password qui!
6      LocalDateTime subscriptionDate
7  ) {}
8
9  // DTO per la richiesta (Input)
10 public record UserRegistrationRequest(
11     @NotBlank String username,
12     @Email String email,
13     @Size(min=8) String password
14 ) {}
```

DTO definition

36.3.2 2. Il Mapper (MapStruct)

MapStruct genera il codice di conversione a **compile time**. È veloce come se lo scrivessi tu a mano.

```
1  @Mapper(componentModel = "spring")
2  public interface UserMapper {
3
4      // Da Entity a DTO (Output)
5      UserResponse toResponse(UserEntity entity);
6
```

```
    // Da DTO a Entity (Input)
    @Mapping(target = "id", ignore = true) // L'ID lo genera il DB
    @Mapping(target = "createdAt", expression = "java(java.time.LocalDateTime.now())")
    UserEntity toEntity(UserRegistrationRequest request);
}
```

UserMapper.java

36.4 Lo "Swap": Il Service Layer in Azione

Ecco il codice che devi scrivere per gestire correttamente la transizione.

```
@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository repository;
    private final UserMapper mapper;

    // SCENARIO 1: LETTURA (Entity -> DTO)
    @Transactional(readOnly = true)
    public UserResponse getUserProfile(Long id) {
        // 1. Recupero Entity dal DB
        UserEntity entity = repository.findById(id)
            .orElseThrow(() -> new NotFoundException("Utente non trovato"));

        // 2. SWAP: Converto in DTO
        // Qui l'entity rimane nel contesto di persistenza, ma il DTO no.
        UserResponse dto = mapper.toResponse(entity);

        return dto;
    }

    // SCENARIO 2: SCRITTURA (DTO -> Entity)
    @Transactional
    public UserResponse registerUser(UserRegistrationRequest request) {
        // 1. SWAP: Converto il DTO in ingresso in una nuova Entity
        UserEntity newUser = mapper.toEntity(request);

        // 2. Business Logic (es. hash password)
        newUser.setPassword(BCrypt.hashpw(request.password(), BCrypt.gensalt()));

        // 3. Salvataggio
        UserEntity savedUser = repository.save(newUser);

        // 4. SWAP Finale: Ritorno il DTO dell'utente salvato
        return mapper.toResponse(savedUser);
    }
}
```

UserService.java - La Logica di Business

36.5 DTO Projections: La "Magia" nelle Query

Finora abbiamo caricato l'intera Entity per poi convertirla. Ma se la tabella `Users` ha 50 colonne e a me servono solo `nome` e `email`? Caricare l'intera Entity è uno spreco di memoria RAM e banda di rete.

Qui entrano in gioco le **JPA Projections**. Possiamo istruire Hibernate per popolare **direttamente** il DTO tramite la query SQL, saltando completamente la fase di creazione dell'Entity.

36.5.1 1. JPQL Constructor Expression (Class-based Projection)

Questa è la tecnica più potente per reportistica e liste massive.

```
public interface UserRepository extends JpaRepository<UserEntity, Long> {

    // MAGIA: Hibernate esegue la SELECT e istanzia direttamente il Record!
    // Non viene creata nessuna Entity UserEntity.
    // La query SQL estrarrà SOLO le colonne username ed email.

    @Query("""
        SELECT new com.example.dto.UserSummary(u.username, u.email)
        FROM UserEntity u
        WHERE u.active = true
    """)
    List<UserSummary> findAllActiveSummaries();
}
```

Repository con Query DTO

Deep Dive: Perché usare le Projections?

Immagina di avere una lista di 10.000 utenti.

- **Approccio Classico (findAll + MapStruct):** Hibernate istanzia 10.000 oggetti Entity (con tutti i campi, relazioni Lazy, proxy). Poi tu crei 10.000 DTO. Totale: 20.000 oggetti in heap + Overhead pesante.
- **Approccio Projection (new DTO...):** Hibernate esegue `SELECT col1, col2...`. Istanzia **soltanto** i 10.000 DTO leggeri. Nessuna Entity viene caricata nel Persistence Context. **Risultato:** Performance 10x più veloci e consumo di memoria dimezzato.

36.5.2 2. Interface-based Projections (Spring Data Magic)

Se non vuoi scrivere la query JPQL a mano, Spring Data offre una magia ancora più oscura: le proiezioni tramite interfaccia.

```
// 1. Definisci un'interfaccia con i getter che corrispondono ai campi dell'Entity
public interface UserNamesOnly {
    String getUsername();
    String getEmail();
}

// 2. Nel Repository usi l'interfaccia come tipo di ritorno
public interface UserRepository extends JpaRepository<UserEntity, Long> {

```

```
    // Spring genera al volo un proxy che implementa l'interfaccia
    // e ottimizza la query SQL per estrarre solo quei due campi.
    List<UserNamesOnly> findByActiveTrue();
}
```

Interface Projection

36.6 Riepilogo Best Practices

Cosa fare	Perché
Usare **Record** per i DTO	Immutabilità, meno codice, serializzazione JSON pulita.
Usare **MapStruct**	Performance a compile-time (no reflection lenta).
Swap nel **Service Layer**	Mantiene la transazione atomica e il Controller pulito.
Usare **DTO Projections**	Per liste lunghe o report, evita di caricare Entity pesanti.
Validazione nel DTO Input	`@NotNull`, `@Size` vanno messi nel DTO, non nell'Entity.

Parte VI

Le fondamenta che hanno portato a Spring

Capitolo 37

L'Ecosistema Web: Protocolli e Server

Ora che abbiamo compilato e pacchettizzato la nostra applicazione, dobbiamo farla parlare con il mondo. Non importa quanto sia complessa la tua logica di business in Java: se non comprendi come viaggiano i dati sulla rete, non sarai mai in grado di debuggare un problema di connessione, configurare un firewall o progettare una API RESTful corretta.

In questo capitolo esploreremo l'architettura fisica della rete e il linguaggio universale del web: il protocollo HTTP.

37.1 Architettura Client-Server: Le basi fisiche

Il Web si basa su un modello di interazione molto semplice: c'è qualcuno che chiede (Client) e qualcuno che risponde (Server).

37.1.1 IP e DNS: L'indirizzo

Ogni macchina collegata a una rete (Internet o Intranet) possiede un indirizzo IP (es. `192.168.1.50`). È l'equivalente dell'indirizzo civico di un palazzo. Poiché gli umani faticano a ricordare numeri, usiamo il **DNS (Domain Name System)**, che funge da rubrica telefonica: traduce `google.com` nell'IP reale del server.

37.1.2 La Porta (Port): Il varco d'ingresso

Questo è un concetto vitale per chi sviluppa in Spring Boot. Se l'IP è l'indirizzo del palazzo, la **Porta** è il numero dell'interno (o dell'ufficio). Un server esegue molti servizi contemporaneamente:

- Porta 80/443: Standard per il traffico Web (HTTP/HTTPS).
- Porta 3306: Standard per MySQL.
- Porta 22: SSH (amministrazione remota).
- **Porta 8080:** Standard per i Web Server Java (Tomcat).

Colloquio: Errore: Address already in use

Cosa significa l'errore `BindException: Address already in use` all'avvio di Spring Boot? Significa che un altro processo sulla tua macchina sta già occupando la porta 8080. Non possono esistere due processi in ascolto sulla stessa porta. Soluzione: Uccidi il processo zombie o cambia la porta di Spring (`server.port=8081`).

37.2 Il Protocollo HTTP: La lingua del Web

L'HyperText Transfer Protocol (HTTP) è un protocollo testuale. Non c'è magia binaria: client e server si scambiano stringhe di testo formattate in modo specifico. Capire l'anatomia di questi messaggi è prerequisito per capire le Annotazioni di Spring MVC.

37.2.1 Anatomia di una REQUEST (Richiesta)

Quando il browser (o Postman) chiama il server, invia un pacchetto composto da tre parti:

1. **Start Line:** Contiene il **Metodo** (Verbo), l'**URI** (il percorso) e la versione HTTP.
2. **Headers:** Metadati (chiave-valore). Es: `Content-Type: application/json` (ti sto mandando un JSON), `Authorization: Bearer xyz` (sono loggato).
3. **Body (Payload):** Il contenuto vero e proprio. Opzionale (es. presente in POST, assente in GET).

Deep Dive: I Verbi HTTP e l'Idempotenza

Nei colloqui Senior, non chiedono "cos'è una GET", ma "cos'è l'idempotenza".

- **GET:** Legge dati. È *Safe* (non modifica il server) e *Idempotente* (se la chiami 100 volte, il risultato è lo stesso).
- **POST:** Crea una nuova risorsa. **NON è Idempotente**. Se invii due volte la richiesta di pagamento, paghi due volte.
- **PUT:** Aggiorna una risorsa (sostituzione completa). È *Idempotente* (se imposti `nome="Mario"` 100 volte, il nome resta "Mario").
- **DELETE:** Cancella una risorsa. È *Idempotente* (cancellare il nulla restituisce sempre successo o 404, ma lo stato del server non cambia dopo la prima volta).

37.2.2 Anatomia di una RESPONSE (Risposta)

Il server elabora e risponde con:

1. **Status Line:** Contiene lo **Status Code** numerico.
2. **Headers:** Es. `Set-Cookie` (salva questo dato), `Content-Type`.
3. **Body:** L'HTML della pagina o il JSON dei dati.

37.2.3 Gli Status Codes: La grammatica della risposta

Non puoi restituire sempre 200 OK. Devi essere semantico.

- **2xx (Successo):** 200 OK (Standard), 201 Created (dopo una POST), 204 No Content (dopo una DELETE).
- **3xx (Redirezione):** 301 Moved Permanently, 302 Found (vai altrove).
- **4xx (Errore Client):** È colpa tua. 400 Bad Request (dati invalidi), 401 Unauthorized (chi sei?), 403 Forbidden (sai chi sono ma non puoi entrare), 404 Not Found.
- **5xx (Errore Server):** È colpa mia. 500 Internal Server Error (NullPointerException non gestita), 503 Service Unavailable.

37.3 Statelessness e Sessioni

Una caratteristica fondamentale di HTTP è di essere **Stateless** (Senza Stato). Il server non ha memoria. Se fai una richiesta adesso, e ne fai un'altra tra un secondo, il server non sa che sei la stessa persona. Tratta ogni richiesta come se fosse la prima volta assoluta.

Il problema: Come facciamo a mantenere un utente "loggato" o un carrello della spesa pieno mentre navighiamo tra le pagine?

37.3.1 La Soluzione Classica: I Cookie e la Sessione (Stateful)

1. Al primo login, il server crea un oggetto `HttpSession` in memoria RAM e gli assegna un ID unico (es. `JSESSIONID=123`).
2. Il server invia questo ID al browser tramite l'header `Set-Cookie`.
3. Il browser salva il biscottino e lo rimanda indietro automaticamente **ad ogni richiesta successiva**.
4. Il server legge l'ID, recupera la memoria associata e "si ricorda" chi sei.

37.3.2 La Soluzione Moderna: Token JWT (Stateless)

Nelle architetture a Microservizi e Cloud, la sessione server-side è un problema (se ho 10 server, quale server ha la mia sessione in memoria?). Si preferisce l'approccio **Token-Based (JWT)**: tutto lo stato è criptato dentro un token che il client conserva e invia ogni volta. Il server non salva nulla in memoria.

37.4 Web Server vs Application Server vs Reverse Proxy

Spesso i termini vengono confusi. Facciamo chiarezza per capire l'architettura di produzione.

- **Web Server (es. Apache HTTPD, Nginx):** È specializzato nel servire contenuti **statici** (HTML, CSS, Immagini) alla massima velocità. Non sa eseguire Java o logica complessa. Spesso agisce da "portiere".
- **Application Server / Servlet Container (es. Apache Tomcat, Jetty):** È il motore che sa eseguire codice Java, gestire Servlet e connettersi ai Database. È "l'intelligenza".

Colloquio: Architettura di Produzione: Il Reverse Proxy

In produzione, raramente esponi Tomcat (porta 8080) direttamente su Internet. È pericoloso e poco efficiente. Si usa il pattern **Reverse Proxy**:

- Davanti a tutto c'è **Nginx** (sulla porta 80/443).
- Nginx gestisce la crittografia HTTPS (certificati SSL) e serve le immagini statiche.
- Se la richiesta richiede logica Java, Nginx la "gira" (proxy pass) a **Tomcat** (che sta al sicuro sulla porta interna 8080).

Questo aggiunge sicurezza e permette il **Load Balancing** (Nginx può smistare il traffico su 10 Tomcat diversi).

Capitolo 38

Java Enterprise e la Servlet API

Quando si chiede a un principiante come funziona Spring Boot, di solito risponde: *"Parte il metodo main e l'applicazione si avvia."*

Un sviluppatore più esperto invece dirà qualcosa come: *"Spring Boot avvia un Web Container embedded (come Tomcat), registra la DispatcherServlet, e questa inoltra le richieste ai Controller."*

In questo capitolo togliamo la patina di magia e osserviamo con calma cosa succede davvero sotto al cofano. La chiave per capirlo è la **Servlet API**: il livello più basso su cui si appoggia tutto il modello web di Java. Ogni concetto che userai in Spring MVC (Header, Cookie, Sessione, Errori, Request/Response) deriva direttamente da qui, e Spring non fa altro che offrirti un modo più elegante per usarli.

38.1 Il Web Container (Tomcat): Il motore nascosto

La Java Standard Edition (J2SE) non sa gestire il web da sola. Offre solo le Socket, strumenti troppo basilari per costruire un server HTTP completo.

Per trasformare un'applicazione Java in un vero Web Server serve un software dedicato: il **Servlet Container** (o Web Container). I più diffusi sono Apache Tomcat, Jetty e Undertow.

Sono loro che:

- ricevono le richieste HTTP,
- creano gli oggetti Request/Response,
- gestiscono i thread,
- e invocano il tuo codice al momento giusto.

38.1.1 Il Modello "Thread-per-Request"

Questa è la caratteristica architetturale più importante del modello servlet. Se ne comprendi il funzionamento, capisci metà delle performance di un'app Java.

Come fa Tomcat a gestire centinaia o migliaia di utenti contemporaneamente?

1. **Il Thread Pool:** Quando parte, Tomcat crea un gruppo di thread (di solito circa 200 in Spring Boot) e li tiene pronti in attesa.
2. **L'Assegnazione:** Ogni volta che arriva una richiesta HTTP, Tomcat assegna un thread libero per gestirla (es. `http-nio-8080-exec-1`).
3. **Il Blocco:** Quel thread resta dedicato a quella richiesta. Se durante l'elaborazione bisogna attendere (es. lettura di un file grande, chiamata al database), il thread rimane fermo senza poter servire altri utenti.

4. **Il Rilascio:** Quando la risposta è pronta, il thread torna nella pool e attende un nuovo lavoro.

Deep Dive: Thread Starvation: il vero limite del modello classico

Cosa succede se arrivano più richieste dei thread disponibili?

La 201esima richiesta, se il pool ha 200 thread, finisce in una **coda**. Se i thread impiegano troppo a liberarsi (per esempio perché il database risponde lentamente), la coda cresce fino a riempirsi, e Tomcat inizia a rifiutare connessioni con errori come *Connection Refused*.

Le operazioni bloccanti diventano quindi il principale collo di bottiglia della scalabilità. È esattamente per superare questo limite che sono nati strumenti non bloccanti come `WebClient` e lo stack reattivo.

Deep Dive: La novità dei Virtual Threads: lo stesso modello ma senza i suoi limiti

I Virtual Threads rivoluzionano il modello tradizionale senza costringerti a cambiare stile di programmazione.

- **Costano pochissimo:** Un Virtual Thread pesa molto meno di un thread classico, quindi l'applicazione può crearne anche decine di migliaia.
- **Il blocco non blocca:** Se una chiamata è bloccante (come JDBC o una REST call), il Virtual Thread può essere "sospeso" senza occupare risorse del sistema operativo.
- **Niente pool rigido:** Non serve più limitare i thread a poche centinaia. Si può avere praticamente un thread per ogni richiesta senza intasare la memoria.
- **Starvation ridotta:** Poiché il numero di thread non è più una risorsa scarsa, la probabilità di saturare la capacità del server diminuisce molto.
- **Codice semplice:** Si continua a scrivere codice imperativo, sequenziale, bloccante, ma con una scalabilità molto superiore.

In breve: i Virtual Threads mantengono la semplicità del modello servlet tradizionale ma eliminano i suoi principali limiti di performance.

38.2 La Servlet: il mattoncino fondamentale

Una **Servlet** è una classe Java pensata per ricevere una richiesta HTTP e restituire una risposta. Per essere gestita da Tomcat, deve rispettare un contratto stabilito dalla Servlet API: può implementare direttamente `javax.servlet.Servlet` oppure, più spesso, estendere una delle classi già pronte offerte dal framework (come `HttpServlet`).

38.2.1 Il Ciclo di Vita (Lifecycle)

Le Servlet non vengono create con `new` dal tuo codice: è Tomcat che decide quando istanziarle e quando eliminarle.

1. **init():** Viene chiamato una sola volta, al momento della creazione. Qui si inizializzano le risorse pesanti.
2. **service():** Viene chiamato per ogni richiesta in arrivo. Tomcat passa due oggetti chiave:
   - `HttpServletRequest`: contiene tutti i dati in arrivo (header, parametri, corpo, cookie).
   - `HttpServletResponse`: è l'oggetto su cui scrivere la risposta.

   In base al tipo di richiesta (GET, POST, ecc.), `service()` delega a `doGet()` o `doPost()`.
3. **destroy():** Chiamato allo spegnimento dell'applicazione per liberare le risorse.

Colloquio: Le Servlet sono Thread-Safe?

No.

Di solito esiste una sola istanza di una Servlet per tutta l'applicazione, e questa istanza è usata da tanti thread in parallelo. Per questo non bisogna mai salvare nei campi della classe dati che appartengono alla singola richiesta.

```
public class BadServlet extends HttpServlet {
    private String user; // Variabile condivisa da tutti i thread!

    protected void doGet(...) {
        this.user = req.getParameter("user"); // Mario sovrascrive Luigi
    }
}
```

Regola pratica: le Servlet (e i Controller Spring) devono essere **stateless**. I dati della richiesta devono vivere solo nelle variabili locali dei metodi.

38.3 Dall'origine a Spring: perché serviva qualcosa di meglio

Per apprezzare la comodità di Spring, basta guardare come si sviluppava prima.

38.3.1 Era 1: CGI e Servlet "nude"

In origine, ogni pagina web era generata da una Servlet.

```
// Java che stampa HTML. Molto difficile da mantenere.
out.println("<html><body><h1>Ciao " + user + "</h1></body></html>");
```

Persino cambiare un colore richiedeva modificare codice Java e ricompilare.

38.3.2 Era 2: JSP (JavaServer Pages)

Per semplificare, si passò a scrivere HTML con dentro pezzi di Java.

```
<html>
    <body>
        <h1>Ciao <\%= request.getParameter("user") \%></h1>
    </body>
</html>
```

Questo mescolava logica e grafica, generando il famoso *spaghetti code*.

38.3.3 Era 3: MVC (Model-View-Controller)

La soluzione fu separare i ruoli:

- **Model**: i dati dell'applicazione,
- **View**: la rappresentazione grafica,
- **Controller**: la Servlet che riceve la richiesta, chiama i servizi e passa i dati alla View.

38.4 Il Pattern Front Controller e la DispatcherServlet

Nel modello MVC originale bisognava configurare una Servlet per ogni funzionalità nel file `web.xml`: `/login` → `LoginServlet`

`/cart` → `CartServlet`
`/home` → `HomeServlet`

Tutte queste Servlet duplicavano codice per logging, sicurezza, eccezioni, gestione degli input...

38.4.1 Spring semplifica tutto: una Servlet per governarle tutte

Spring applica il pattern **Front Controller**. Esiste una sola Servlet reale registrata nel Web Container: la **DispatcherServlet**.

1. **Centralizza:** tutte le richieste entrano da lei.
2. **Indirizza:** capisce quale Controller deve rispondere grazie alle mappature interne (**HandlerMapping**).
3. **Standardizza:** gestisce conversioni, errori, JSON, sicurezza e molto altro, lasciando al Controller solo la logica applicativa.

Ecco perché in Spring Boot non configuri più file `web.xml` e non estendi `HttpServlet`: Spring nasconde i dettagli della Servlet API, ma sotto il cofano Tomcat sta ancora invocando `doGet()` e `doPost()`.

38.5 Riepilogo del Flusso: Prima di Spring vs Con Spring

Comprendere tutti i pezzi è difficile finché non si osserva il flusso completo, confrontando come funzionava un'applicazione web in Java **prima di Spring** e come funziona **con Spring e Spring Boot**. Questa sezione mette ordine mostrando i passaggi essenziali e il ruolo di ogni componente.

38.5.1 Prima di Spring: Il Flusso Basato su Servlet Pure

Senza Spring, il Web Container (Tomcat, Jetty, ecc.) era responsabile di quasi tutto. Lo sviluppatore doveva configurare manualmente ogni Servlet e occuparsi della maggior parte della logica.

1. **Il Browser invia una richiesta HTTP.**
2. **Il Web Container (Tomcat) riceve la richiesta** e crea gli oggetti `HttpServletRequest` e `HttpServletResponse`.
3. **Tomcat sceglie quale Servlet invocare** consultando il file di configurazione `web.xml`.
4. **La Servlet elabora la richiesta**: legge parametri, interagisce con servizi, gestisce errori.
5. **La Servlet genera direttamente l'output**: HTML, JSON o qualunque testo.
6. **Tomcat invia la risposta al client**.

In questo modello:

- ogni URL richiedeva una Servlet dedicata;
- la logica era spesso mescolata con la presentazione;
- bisognava gestire manualmente conversioni, errori, sicurezza, mapping, factory di oggetti, ecc.;
- il codice diventava velocemente difficile da mantenere.

38.5.2 Con Spring: Il Flusso Mediato dalla DispatcherServlet

Spring introduce una struttura che ripulisce il caos delle Servlet manuali. Al posto di decine di Servlet, ce n'è una sola: la **DispatcherServlet**.

1. **Il Browser invia una richiesta HTTP.**
2. **Tomcat (embedded) riceve la richiesta** e crea gli oggetti standard Request/Response.

3. **Tomcat invoca la DispatcherServlet**, che è l'unica Servlet registrata da Spring.
4. **La DispatcherServlet analizza la richiesta** (URL, metodo, parametri, header).
5. **Trova il Controller corretto** consultando i suoi meccanismi interni (HandlerMapping).
6. **Invoca il metodo del Controller** passando i dati già convertiti (es. JSON → POJO).
7. **Il Controller restituisce un oggetto Java** (Model, DTO, ecc.).
8. **La DispatcherServlet costruisce la risposta HTTP** (HTML, JSON, ecc.).
9. **La risposta torna a Tomcat**, che la invia al client.

In questo modello:

- esiste una sola Servlet reale, la DispatcherServlet;
- i Controller sono semplici classi Java senza responsabilità infrastrutturali;
- Spring si occupa della conversione degli input, della serializzazione in output, della gestione degli errori, del binding dei parametri;
- l'applicazione è più ordinata, modulare e testabile.

38.5.3 Vista d'insieme: Prima vs Dopo

Prima di Spring

`Browser` → `Tomcat` → `TuaServlet` → `Tomcat` → `Browser`

Con Spring

`Browser` → `Tomcat` → `DispatcherServlet` → `Controller` → `DispatcherServlet` → `Tomcat` → `Browser`

La differenza fondamentale è che Spring prende il posto di tutte le Servlet manuali e si occupa lui della parte "difficile". Tu scrivi solo i Controller, mentre DispatcherServlet gestisce tutto il resto in modo coerente, standardizzato e pulito.

Capitolo 39

Architettura a Strati (N-Tier) in Spring

Abbiamo appreso come il Web comunica tramite HTTP e come Java gestisce queste comunicazioni tramite le Servlet. Tuttavia, in un'applicazione Enterprise, sapere "come ricevere una richiesta" è solo l'inizio. La sfida principale è organizzare la complessità.

Se scrivessimo codice di validazione, logica di business, calcoli e query SQL tutto all'interno di un'unica classe, creeremmo quello che viene definito **"God Class Anti-Pattern"**. Un codice monolitico, fragile e impossibile da testare.

La soluzione standard nell'ingegneria del software è l'**Architettura a Strati (Layered Architecture)**.

39.1 Il Principio: Separation of Concerns (SoC)

Il principio guida è la **Separazione delle Responsabilità**. Ogni componente deve avere un unico motivo per cambiare. Possiamo immaginare l'architettura di Spring come un ristorante ben organizzato:

1. **La Reception (DispatcherServlet):** Accoglie i clienti e li smista ai tavoli corretti.
2. **Il Cameriere (@Controller):** Prende l'ordine specifico, controlla che sia comprensibile e lo porta in cucina.
3. **Lo Chef (@Service):** Esegue la ricetta. Non vede il cliente, si concentra solo sulla preparazione.
4. **Il Magazziniere (@Repository):** Fornisce gli ingredienti grezzi presi dagli scaffali (Database).

39.2 1. Presentation Layer (L'Interfaccia)

È lo strato più esterno. In Spring, questo livello non è gestito da un singolo componente, ma dalla collaborazione di due attori distinti: l'infrastruttura e l'applicativo.

39.2.1 L'Infrastruttura: La `DispatcherServlet`

La `DispatcherServlet` è il vero "Front Controller". È una **Servlet** standard di Java EE che funge da punto di ingresso unico.

- **Routing:** Analizza l'URL in ingresso e consulta l'*Handler Mapping* per decidere quale Controller chiamare.

- **Gestione degli Errori:** Intercetta le eccezioni non gestite per mostrare pagine di errore o JSON coerenti.
- **Gestione della Risposta:** Scrive fisicamente i byte della risposta HTTP (headers, status code 200/404/500).

39.2.2 L'Applicativo: Il `@Controller` / `@RestController`

Il Controller è un semplice oggetto Java (POJO) che definisce l'interfaccia dell'applicazione.
Responsabilità:

- **Mapping:** Definisce la "forma" delle API (es. `@GetMapping("/users")`).
- **Deserializzazione:** Riceve i dati in ingresso. Qui entra in gioco il pattern **DTO (Data Transfer Object)**. Il Controller non dovrebbe ricevere le entità del database, ma oggetti "usa e getta" ottimizzati per la richiesta.
- **Validazione:** Verifica la correttezza formale (es. "L'email ha una @?", "La password è di 8 caratteri?").
- **Delega:** Passa i dati puliti al Service Layer.

  Nota Bene: Il Controller è "ignorante" riguardo alla logica di business. Non sa *come* si crea un utente, sa solo *chi* chiamare per farlo.

39.3 2. Business Logic Layer (Il Cuore)

Questo livello è rappresentato dalle classi annotate con `@Service`. Qui risiede il valore dell'applicazione.

Responsabilità:

- **Logica di Dominio:** Implementa le regole aziendali (es. "Non si possono emettere fatture nei giorni festivi").
- **Orchestrazione:** Coordina più Repository o servizi esterni (es. "Salva l'ordine su DB" → "Scala i soldi dalla carta di credito" → "Invia Email").
- **Gestione delle Transazioni (`@Transactional`):** È fondamentale. Il Service garantisce l'atomicità: se l'invio dell'email fallisce, il salvataggio dell'ordine deve essere annullato (Rollback).
- **Indipendenza:** Il codice del Service non deve mai dipendere da oggetti HTTP (come `HttpServletRequest`).

39.4 3. Persistence Layer (I Dati)

Questo livello gestisce l'accesso ai dati ed è annotato con `@Repository`. In Spring Data JPA, questo strato è spesso definito tramite **Interfacce**.

Responsabilità:

- **CRUD:** Fornire operazioni di base (Create, Read, Update, Delete).
- **Astrazione:** Nascondere i dettagli del database (SQL, NoSQL, File). Il Service chiede "dammi l'utente", il Repository decide se fare una `SELECT` SQL o una chiamata a MongoDB.
- **Traduzione Eccezioni:** Converte gli errori tecnici del driver (es. `ConstraintViolationException`) in eccezioni generiche di Spring.

39.5 Riepilogo del Flusso di una Richiesta

1. **HTTP Request** → Arriva alla `DispatcherServlet`.

2. `DispatcherServlet` → Trova e chiama il `UserController`.
3. `UserController` → Valida il DTO e chiama `UserService`.
4. `UserService` → Apre una transazione e chiama `UserRepository`.
5. `UserRepository` → Esegue SQL sul Database e restituisce una `Entity`.
6. **Ritorno:** I dati risalgono la catena, vengono convertiti in JSON e inviati come **HTTP Response**.

Capitolo 40

Dal Codice Rigido all'Inversion of Control

Siamo arrivati al culmine della nostra preparazione. Abbiamo un'architettura a strati, sappiamo come funziona il web, ma c'è ancora un problema fondamentale nel modo in cui scriviamo codice Java "standard".

Il problema è la parola chiave più comune del linguaggio: **`new`**.

In questo capitolo vedremo perché istanziare manualmente le dipendenze è il nemico numero uno della manutenibilità e come un semplice cambio di prospettiva (l'Inversione del Controllo) abbia dato vita a tutto l'ecosistema Spring.

40.1 Analisi del Tight Coupling (Accoppiamento Stretto)

Torniamo al nostro Business Layer. Immagina di dover scrivere un servizio per processare pagamenti. L'approccio istintivo di un programmatore Java base è questo:

```
1 public class PaymentService {
2     // Dipendenza diretta: PaymentService CONOSCE e CREA PaypalProvider
3     private PaypalProvider provider = new PaypalProvider();
4
5     public void pay(double amount) {
6         provider.sendMoney(amount);
7     }
8 }
```

Listing 40.1: Il problema della dipendenza diretta

Questo codice sembra innocuo, ma architettonicamente è un disastro. Perché? Si è creato un **Tight Coupling** (Accoppiamento Stretto) tra `PaymentService` e `PaypalProvider`.

40.1.1 I 3 problemi capitali del "new"

1. **Rigidità:** Se domani l'azienda decide di passare da PayPal a Stripe, devi aprire la classe `PaymentService`, modificare il codice sorgente, ricompilare e ridistribuire l'applicazione. Hai violato l'Open/Closed Principle.
2. **Configurazione Impossibile:** Se `PaypalProvider` ha bisogno di una password per funzionare, chi gliela passa? Il `PaymentService` dovrebbe leggere file di configurazione? Non è il suo lavoro.
3. **Testabilità (Il punto critico):** Questo è il motivo principale per cui usiamo Spring.

40.2 L'Incubo della Testabilità

Proviamo a scrivere uno Unit Test per il codice sopra.

```
@Test
public void testPayment() {
    PaymentService service = new PaymentService();
    service.pay(100.0);
    // AIUTO! Questo metodo chiama DAVVERO i server di PayPal!
    // Stiamo addebitando soldi veri sulla carta di credito durante i test.
}
```

Poiché `PaymentService` crea l'istanza di `PaypalProvider` al suo interno tramite `new`, non c'è modo per noi (dall'esterno) di intervenire e sostituire PayPal con un oggetto finto (Mock) per il test. Siamo bloccati. Il codice non è testabile in isolamento.

40.3 Inversion of Control (IoC): La Filosofia

Per risolvere il problema, dobbiamo ribaltare il paradigma. Invece di lasciare che sia l'oggetto a cercare e creare le sue dipendenze (Controllo Attivo), facciamo in modo che le dipendenze gli vengano fornite dall'esterno (Controllo Passivo).

Questo principio si chiama **Inversion of Control (IoC)**. Spesso viene riassunto con l'**Hollywood Principle**:

> *"Non chiamarci, ti chiameremo noi."*

In termini software:

- **Senza IoC:** La classe A chiede "Ho bisogno di un oggetto B, me lo creo".
- **Con IoC:** La classe A dichiara "Ho bisogno di un oggetto B". Qualcuno (un Container, un Framework, o il Main) glielo passerà quando serve.

40.4 Dependency Injection (DI): La Tecnica

L'IoC è il principio astratto. La **Dependency Injection (DI)** è il modo pratico in cui lo implementiamo in Java. L'idea è semplice: spostiamo la creazione delle dipendenze fuori dalla classe e le passiamo ("iniettiamo") attraverso il costruttore o i setter.

40.4.1 Refactoring verso la DI

Passo 1: Astrarre la dipendenza (usare un'interfaccia).

```
public interface PaymentProvider {
    void sendMoney(double amount);
}
```

Passo 2: Usare la Constructor Injection.

```
public class PaymentService {
    private final PaymentProvider provider;

    // Non uso 'new'. Chiedo 'chiunque implementi l'interfaccia'.
    public PaymentService(PaymentProvider provider) {
        this.provider = provider;
    }
```

```
8
9       public void pay(double amount) {
10          provider.sendMoney(amount);
11      }
12  }
```

Listing 40.2: Codice disaccoppiato con DI

40.4.2 Il Risultato: Flessibilità Totale

Ora guardiamo cosa succede.

In Produzione:

```
PaymentProvider realPaypal = new PaypalProvider("api-key-reale");
PaymentService service = new PaymentService(realPaypal);
```

Nei Test:

```
// Posso passare un oggetto finto che non chiama internet!
PaymentProvider mockProvider = new MockPaymentProvider();
PaymentService service = new PaymentService(mockProvider);
```

Abbiamo disaccoppiato la logica di business (`PaymentService`) dall'implementazione tecnica (`PaypalProvider`).

40.5 Il ruolo del Container (Spring)

La Dependency Injection è bellissima, ma sposta il problema un livello più in alto. Chi fa `new`? Chi collega tutti questi oggetti?

Se hai 100 classi, scrivere a mano nel `main` tutto il cablaggio è un incubo:

```
Repo r = new Repo();
Service s = new Service(r);
Controller c = new Controller(s);
...
```

Qui entra in gioco **Spring Framework**. Spring è, essenzialmente, un gigantesco **Container IoC** (o DI Container).

1. Tu dichiari le classi (i Bean) e le loro dipendenze (tramite annotazioni o costruttori).
2. All'avvio dell'applicazione, Spring legge tutto, crea le istanze nell'ordine giusto e le inietta dove servono.
3. Tu non scrivi mai più `new Service()`. Chiedi a Spring l'istanza pronta.

Deep Dive: Riassunto: Perché Spring?

Ora hai la risposta completa per il colloquio. Usiamo Spring perché fornisce un **Container IoC** che gestisce automaticamente la **Dependency Injection**. Questo ci permette di scrivere codice **Disaccoppiato** (Loose Coupling) e facilmente **Testabile**, delegando al framework la gestione del ciclo di vita e della configurazione degli oggetti.

Parte VII

Ecosistema Spring

Capitolo 41

Spring Core e Inversion of Control

Spring non è solo una collezione di librerie, è un modo diverso di concepire il flusso di un'applicazione. Al centro di tutto c'è lo **Spring Container**, che gestisce la creazione e il cablaggio degli oggetti (Bean).

41.1 IoC Container: BeanFactory vs ApplicationContext

L'**Inversion of Control (IoC)** è il principio secondo cui il controllo del flusso del programma non è in mano allo sviluppatore, ma al framework. "Don't call us, we'll call you".

In Spring, il container IoC è responsabile di istanziare, configurare e assemblare i Bean. Esistono due interfacce principali:

1. **BeanFactory:** L'interfaccia radice. Fornisce il meccanismo di base per la DI.
   - Caricamento **Lazy**: I bean vengono creati solo quando richiesti con `getBean()`.
   - Usata raramente oggi (solo in dispositivi con risorse estremamente limitate).
2. **ApplicationContext:** Estende BeanFactory. È il container utilizzato nelle applicazioni Enterprise.
   - Caricamento **Eager**: I bean Singleton vengono istanziati all'avvio dell'applicazione (permettendo di rilevare subito errori di configurazione).
   - Aggiunge supporto per Eventi, AOP, Internazionalizzazione (i18n).

41.2 Dependency Injection: Constructor vs Setter vs Field

Se IoC è il principio (la filosofia), la **Dependency Injection (DI)** è il pattern implementativo. Esistono tre modi per iniettare le dipendenze, ma non sono equivalenti.

41.2.1 1. Field Injection (Sconsigliata)

Questa metodologia è estremamente diffusa nei tutorial online per la sua brevità, ma in ambito professionale è considerata una **cattiva pratica** (*"Bad Smell"*) e dovrebbe essere evitata.

```
@Service
public class OrderService {
    @Autowired
    private UserRepository repo; // Field Injection
}
```

Cosa succede "Dietro le Quinte"?

L'apparente semplicità della Field Injection nasconde un processo intrusivo basato sulla **Java Reflection API**. Quando Spring incontra l'annotazione `@Autowired` su un campo privato, esegue una serie di operazioni che bypassano le normali regole di incapsulamento di Java:

1. **Istanziazione del Bean:** Spring crea un'istanza di `OrderService` chiamando il costruttore di default (spesso vuoto o implicito). In questo preciso momento, l'oggetto esiste ma è in uno stato inconsistente: il campo `repo` è ancora `null`.
2. **Scansione e Reflection:** Il container ispeziona la classe, individua il campo annotato e utilizza la *Reflection* per forzarne l'accessibilità. Esegue un'operazione equivalente a `field.setAccessible(true)`, rendendo di fatto accessibile ciò che avevi dichiarato `private`.
3. **Iniezione Forzata:** Spring inietta l'istanza di `UserRepository` direttamente nel campo, modificando lo stato interno dell'oggetto dall'esterno.

Perché è considerata "Bad Practice"?

Capire il meccanismo interno chiarisce perché questo approccio è sconsigliato:

- **Violazione dell'Incapsulamento:** Stai permettendo al framework di violare la privacy dell'oggetto, modificandone i campi senza passare per un costruttore o un metodo setter.
- **Difficoltà nei Test Unitari:** Poiché non esiste un costruttore per passare le dipendenze (mock), per testare questa classe sei costretto a usare a tua volta la Reflection o ad avviare l'intero contesto di Spring (test di integrazione) anche per logiche semplici.
- **Impossibilità di usare `final`:** I campi iniettati in questo modo non possono essere dichiarati `final`, rendendo l'oggetto mutabile e potenzialmente non thread-safe.
- **Rischio di NullPointerException:** Se istanzi la classe manualmente (es. `new OrderService()`) senza usare il container di Spring, il campo `repo` rimarrà `null`, causando errori a runtime.

41.2.2 2. Setter Injection (Per dipendenze opzionali)

Utile se la dipendenza può cambiare a runtime o non è strettamente necessaria.

```
private UserRepository repo;

@Autowired
public void setRepo(UserRepository repo) {
    this.repo = repo;
}
```

41.2.3 3. Constructor Injection (Raccomandata)

È lo standard industriale moderno (e da Spring 4.3 `@Autowired` è opzionale se c'è un solo costruttore).

```
@Service
public class OrderService {
    private final UserRepository repo; // Immutabile!

    public OrderService(UserRepository repo) {
        this.repo = repo;
    }
}
```

Vantaggi: Garantisce che l'oggetto sia sempre in uno stato valido (non può esistere senza le sue dipendenze), permette l'uso di `final` e facilita i test (basta passare un mock nel costruttore).

41.3 Spring Beans: Scopes

Lo **Scope** definisce il ciclo di vita e la visibilità di un Bean.

Colloquio: Qual è la differenza tra Singleton e Prototype?

- **Singleton (Default):** Spring crea **una sola istanza** del Bean per container. Ogni volta che viene richiesto, viene restituito lo stesso oggetto condiviso.
  - **Attenzione:** I bean Singleton devono essere **Stateless** (senza stato modificabile). Se salvi dati dell'utente in un campo di un Singleton, avrai gravi problemi di concorrenza (tutti gli utenti vedranno i dati dell'altro).
- **Prototype:** Spring crea una **nuova istanza** ogni volta che il bean viene richiesto.
- **Web Scopes:** `Request` (uno per richiesta HTTP), `Session` (uno per sessione utente), `Application`.

41.4 Lazy Initialization e Dipendenze Circolari

Di default, Spring Boot inizializza tutti i bean all'avvio (Eager). Questo è ottimo per rilevare errori subito, ma può rallentare lo startup. L'annotazione `@Lazy` indica a Spring di creare il bean solo quando viene richiesto per la prima volta.

Colloquio: Come risolvi l'errore "BeanCurrentlyInCreationException"?

Questo errore indica una **Dipendenza Circolare**: A dipende da B, e B dipende da A. Spring non riesce a istanziare A perché gli serve B, ma non può creare B perché gli serve A.

Soluzione:

1. **Refactoring (Migliore):** Riprogettare le classi per eliminare il ciclo (spesso indica cattivo design).
2. **@Lazy (Workaround):** Iniettare una delle dipendenze con `@Lazy`.

```
@Service
public class ServiceA {
    private final ServiceB serviceB;

    // Spring inietta un Proxy al posto dell'oggetto reale.
    // L'oggetto reale ServiceB verra' creato solo alla prima chiamata metodo.
    public ServiceA(@Lazy ServiceB serviceB) {
        this.serviceB = serviceB;
    }
}
```

41.5 Lifecycle Hooks: @PostConstruct e @PreDestroy

Un Bean non viene solo fatto con `new`. Attraversa un ciclo complesso.

Il Ciclo di Vita Semplificato:

1. **Istanziazione:** Spring chiama il costruttore.

2. **Popolamento Proprietà:** Spring inietta le dipendenze.
3. **Post-Initialization (@PostConstruct):** Qui puoi eseguire logica di avvio (es. riempire una cache). Nota: Non puoi farlo nel costruttore perché le dipendenze potrebbero non essere ancora pronte.
4. **Bean Ready:** Il bean è in uso.
5. **Pre-Destruction (@PreDestroy):** Prima che il container si spenga, Spring chiama questo metodo per pulire le risorse (chiudere connessioni, thread pool).

```
@Component
public class CacheService {

    @PostConstruct
    public void init() {
        System.out.println("Bean creato e dipendenze iniettate. Carico la cache...");
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("Container in chiusura. Pulisco risorse...");
    }
}
```

41.6 Spring AOP (Aspect Oriented Programming)

La OOP è ottima per separare la logica business, ma pessima per la logica trasversale (**Cross-Cutting Concerns**) come Logging, Sicurezza e Transazioni. AOP permette di separare queste logiche.

Colloquio: Concetti chiave AOP

- **Aspect:** Il modulo che contiene la logica trasversale (es. `LoggingAspect`).
- **Join Point:** Un punto nel programma dove l'aspetto può intervenire (in Spring, è sempre l'esecuzione di un metodo).
- **Pointcut:** Un'espressione che definisce *dove* applicare l'aspetto (es. "tutti i metodi nel package `com.service.*`").
- **Advice:** L'azione da compiere (`@Before`, `@After`, `@Around`).

Deep Dive: @Transactional e AOP

Quando metti `@Transactional` su un metodo, Spring non modifica il tuo metodo. Usa AOP per creare un **Proxy** dinamico che avvolge la tua classe.

1. Il Proxy intercetta la chiamata (**Advice Around**).
2. Apre la connessione DB (**Before**).
3. Chiama il tuo metodo.
4. Se eccezione → Rollback, altrimenti Commit (**AfterReturning/Throwing**).

Capitolo 42

Spring Boot: Convention over Configuration

Immagina Spring Framework come una scatola gigante di Lego sparsi sul pavimento. Puoi costruire qualsiasi cosa, ma devi cercare ogni singolo pezzo, decidere come incastrarlo e configurarlo. È potente, ma lento.

Spring Boot è come comprare il set "Castello di Harry Potter": i pezzi sono già organizzati in buste numerate e hai le istruzioni predefinite. La filosofia si chiama **Convention over Configuration**:

- **Configurazione classica (Spring):** "Ho bisogno di un database, ecco il driver, ecco l'url, ecco il pool di connessioni, ecco l'username..."
- **Convenzione (Spring Boot):** "Vedo che hai aggiunto il driver di MySQL nel progetto. Presumo tu voglia un database MySQL locale. L'ho configurato io per te."

42.1 Il Punto di Partenza: @SpringBootApplication

Questa annotazione, situata nella classe 'Main', è la "chiave di accensione" della tua applicazione.

Deep Dive: Analisi: @SpringBootApplication

Non è una singola annotazione, ma un contenitore che ne raggruppa tre fondamentali:

1. `@Configuration`: Indica che la classe può definire Bean.
2. `@ComponentScan`: Istruisce Spring a cercare componenti (`@Controller`, `@Service`) nel package corrente e nei suoi sotto-package.
3. `@EnableAutoConfiguration`: **È qui che avviene la magia.** Ordina a Boot di analizzare le librerie nel classpath e configurare l'applicazione di conseguenza.

```
@SpringBootApplication
public class MiaApplicazione {
    public static void main(String[] args) {
        // Avvia l'applicazione e il server Tomcat integrato
        SpringApplication.run(MiaApplicazione.class, args);
    }
}
```

Listing 42.1: La classe Main standard

42.2 Come Funziona l'Auto-Configurazione?

Molti pensano che l'auto-configurazione sia magia nera. In realtà, è deterministica e basata sull'annotazione `@Conditional`.

Deep Dive: Concetto Chiave: L'Analogia dell'Arredatore

Immagina di assumere un arredatore (Spring Boot) per casa tua. L'arredatore entra in cucina e fa questo ragionamento:

1. "Vedo che c'è un attacco del gas (*Libreria nel classpath*)..."
2. "...e vedo che NON hai ancora comprato un fornello (*@ConditionalOnMissingBean*)."
3. "**Allora** installerò io un fornello standard per te (*Auto-Configuration*)."

Se tu avessi già comprato un fornello personalizzato (definito un tuo Bean), l'arredatore non avrebbe fatto nulla.

Ecco come appare questo ragionamento nel codice interno di Spring Boot:

```
@Configuration
// 1. Esegui solo se trovi la classe di Jackson (JSON)
@ConditionalOnClass(ObjectMapper.class)
public class JacksonAutoConfig {

    @Bean
    // 2. Crea questo Bean SOLO se l'utente non ne ha creato uno suo
    @ConditionalOnMissingBean
    public ObjectMapper objectMapper() {
        return new ObjectMapper(); // Versione standard
    }
}
```

Listing 42.2: Logica interna (Semplificata)

42.3 Gestire le Configurazioni: Properties

Invece di scrivere valori direttamente nel codice ("Hardcoding"), Spring Boot usa `application.yml` (o `.properties`).

Colloquio: Differenza tra @Value e @ConfigurationProperties

- **@Value("${chiave}")**:
  - Inietta un **singolo valore**.
  - *Pro:* Veloce per configurazioni isolate.
  - *Contro:* Non Type-Safe, difficile da mantenere se i valori sono molti.
- **@ConfigurationProperties(prefix = "app")**:
  - Mappa una gerarchia di proprietà su una **classe Java (POJO)**.
  - *Pro:* **Type-Safe**, supporta validazione, autocompletamento IDE.
  - *Uso:* Ideale per configurazioni complesse (es. parametri mail server, database custom).

```
// Nel file application.yml:
// server-mail:
//   host: smtp.gmail.com
//   port: 587
```

```java

@Configuration
@ConfigurationProperties(prefix = "server-mail")
public class MailConfig {

    private String host;
    private int port; // Converte automaticamente in int!

    // Getter e Setter obbligatori
    public void setHost(String host) { this.host = host; }
    public void setPort(int port) { this.port = port; }
}
```

Listing 42.3: Esempio Best Practice con ConfigurationProperties

42.4 Eseguire Task Periodici: @Scheduled

Per eseguire operazioni batch o ricorrenti (es. invio report settimanale, pulizia cache). **Nota:** Richiede `@EnableScheduling` su una classe di configurazione.

```java
@Component
public class JobService {

    // Esegui ogni 5 secondi (5000 ms) dalla FINE dell'esecuzione precedente
    @Scheduled(fixedDelay = 5000)
    public void puliziaVeloce() {
        System.out.println("Pulizia cache...");
    }

    // Sintassi CRON: Secondi Minuti Ore Giorno Mese GiornoSettimana
    // "Alle ore 04:00:00 di ogni giorno"
    @Scheduled(cron = "0 0 4 * * *")
    public void reportNotturno() {
        System.out.println("Invio report...");
    }
}
```

Listing 42.4: Esempi di Scheduling

Deep Dive: Attenzione: Il Thread Singolo

Di default, lo scheduler di Spring usa un **singolo thread** per tutti i task `@Scheduled` dell'applicazione. Se hai due task programmati allo stesso orario, o se uno si blocca, **tutti gli altri task si bloccheranno o saranno ritardati**.

Soluzione: Configurare un `ThreadPoolTaskScheduler` custom per permettere l'esecuzione parallela.

42.5 Monitoraggio: Spring Actuator

In produzione, devi sapere se l'applicazione è viva. Actuator espone endpoint REST pronti all'uso. Basta aggiungere la dipendenza `spring-boot-starter-actuator`.

- `/actuator/health`: Stato di salute (**UP/DOWN**). Controlla connessioni DB, disco, code.

- `/actuator/metrics`: Metriche dettagliate (CPU, RAM, Thread attivi).
- `/actuator/loggers`: Permette di cambiare il livello di log a runtime (es. da INFO a DEBUG) senza riavviare.

Deep Dive: Security Warning

Gli endpoint di Actuator espongono dettagli sensibili. In produzione, **non devono mai** essere accessibili pubblicamente.

1. Usa Spring Security per proteggerli (richiedi ruolo ADMIN).
2. Oppure esponili su una porta diversa e bloccala dal firewall esterno: `management.server.port=8081`.

Capitolo 43

L'Ecosistema Spring: Storia e Rivoluzione Boot

Arrivati a questo punto, dobbiamo fare un passo indietro e guardare la mappa completa. Spring non è una singola libreria, è un "universo" di progetti costruiti sopra un nucleo comune.

Un Senior Developer deve saper distinguere tra il Framework, Boot e le versioni che definiscono la compatibilità del JDK, comprendendo le rivoluzioni introdotte con la versione 3.

43.1 Architettura a Moduli

Il cuore di Spring è il container di **Inversion of Control (IoC)**. Tutto il resto sono moduli che si agganciano a questo nucleo.

1. **Spring Core Container:** Gestisce i Bean, la Dependency Injection (DI) e il ciclo di vita dell'applicazione. (Moduli: `spring-core`, `spring-beans`, `spring-context`).
2. **Spring AOP:** Programmazione Orientata agli Aspetti (usata per Transazioni e Sicurezza).
3. **Data Access/Integration:** JDBC, ORM (Hibernate), Transaction Management.
4. **Web:** Spring MVC (Servlet-based) e Spring WebFlux (Reactive).
5. **Test:** Moduli per Unit e Integration Testing (MockMvc, TestContext).

43.2 Spring Framework vs Spring Boot

Questa è la domanda di riscaldamento classica.

Colloquio: Differenza tra Spring e Spring Boot

Domanda: Spring Boot sostituisce Spring Framework?

Risposta: Assolutamente no.

- **Spring Framework** è il motore. Fornisce le funzionalità (DI, MVC, Transaction). Richiede molta configurazione manuale (XML o classi Java con tanti `@Bean`).
- **Spring Boot** è l'automobile assemblata. È un tool "opinionated" (con opinioni forti) che:
  1. **Auto-Configuration:** Configura automaticamente Spring basandosi sulle librerie presenti nel classpath.
  2. **Embedded Server:** Include Tomcat/Jetty dentro il JAR (non serve installare un server esterno).

3. **Starters:** Gestisce le dipendenze in gruppi logici (es. `spring-boot-starter-web`).

43.3 Evoluzione delle Versioni (Timeline)

Prima di arrivare a oggi, ecco da dove veniamo:

43.3.1 Spring Framework 4 (Legacy)

Baseline Java 6. Introdotto il supporto completo a Java 8 e `@RestController`. Oggi è debito tecnico.

43.3.2 Spring Framework 5 (Era Boot 2.x)

Baseline Java 8. Ha introdotto lo stack **Reactive** (WebFlux). È stata la versione standard fino al 2022.

43.4 GraalVM e Native Images

Fino a ieri, Java funzionava in un modo. Oggi, con Spring Boot 3 e GraalVM, può funzionare in un modo completamente diverso. Capire questa differenza è essenziale per il Cloud moderno.

43.4.1 L'Analogia del Traduttore (JIT vs AOT)

Per capire la differenza tecnica, usiamo un'analogia semplice. Immagina di dover leggere un libro scritto in Russo (il Codice Java) a un pubblico che parla solo Italiano (la CPU/Hardware).

1. **Approccio Classico (JVM - JIT): Il Traduttore Simultaneo.**
   - Tu inizi a leggere il libro in Russo.
   - Un interprete (la JVM) ascolta e traduce frase per frase in Italiano mentre parli.
   - **Problema:** All'inizio l'interprete è lento, deve scaldarsi, capire il contesto.
   - **Vantaggio:** Se capisce che una frase si ripete spesso, impara a tradurla velocissimamente (Ottimizzazione a Runtime).
2. **Approccio GraalVM (Native - AOT): Il Libro Già Tradotto.**
   - Prima ancora di salire sul palco (durante la Build), traduci tutto il libro in Italiano e lo stampi.
   - Sul palco leggi direttamente l'Italiano.
   - **Vantaggio:** Inizi a leggere all'istante. Non serve l'interprete.
   - **Svantaggio:** Se il libro cambia, devi ristamparlo tutto.

43.4.2 Come funziona tecnicamente (Sotto il cofano)

1. Il vecchio mondo: JIT (Just-In-Time)

Il codice Java viene compilato in **Bytecode** (`.class`). Questo non è comprensibile dalla CPU. Serve la JVM che lo avvia e lo traduce mentre l'applicazione gira.

- **Avvio Lento:** La JVM deve caricare migliaia di classi in memoria.
- **Consumo RAM:** La JVM stessa occupa memoria (overhead) solo per esistere.

2. Il nuovo mondo: AOT (Ahead-Of-Time)

Qui entra in gioco **GraalVM**. GraalVM prende il tuo Bytecode, prende tutte le librerie (Spring, Hibernate, ecc.), prende persino pezzi della JVM (come il Garbage Collector) e **compila tutto staticamente** in un unico file binario (Codice Macchina: 0 e 1).

Il risultato è un file `.exe` (o binario Linux) che non ha bisogno di Java installato per girare.

Deep Dive: Perché Spring Boot 3 è fondamentale qui?

GraalVM ha un limite enorme: il **"Mondo Chiuso"**. Per creare il file binario, deve sapere ESATTAMENTE quali classi userai. Ma Java è dinamico! Usa la **Reflection** (carica classi in base a stringhe di testo, es. i file di config). GraalVM non riesce a vedere queste classi "nascoste" e le elimina per risparmiare spazio. Risultato: l'app crasha.

Soluzione: Spring Boot 3, durante la compilazione, analizza il tuo codice e crea una "mappa" per GraalVM, dicendogli: *"Ehi, non buttare via la classe User, mi servirà!"*.

43.4.3 Confronto Visivo: Il Ciclo di Vita

Differenza nel Build Process

Processo Standard (JVM):

1. Scrivi Codice Java.
2. Compiler `javac` → Crea `.class` (Bytecode).
3. **Esecuzione:** Lanci `java -jar app.jar`. La JVM parte e interpreta.

Processo Native (GraalVM):

1. Scrivi Codice Java.
2. Spring Boot AOT Engine → Analizza e genera configurazioni.
3. GraalVM Native Compiler → Macina per minuti (è lento a compilare!).
4. **Risultato:** Crea un file binario `app.exe`.
5. **Esecuzione:** Lanci `./app.exe`. Parte in 50 millisecondi.

43.4.4 Tabella Decisiva: Quando usare cosa?

Ai colloqui chiedono: "Perché non usiamo sempre GraalVM?". Ecco la risposta da Senior.

Fattore	JVM Classica	GraalVM Native
Tempo di Avvio	Lento (secondi)	**Istantaneo** (ms)
Memoria RAM	Alta (occupa molto)	**Bassissima**
Throughput	**Migliore** (Il JIT ottimizza in base all'uso reale)	Inferiore (Il codice è statico)
Build Time	Veloce (secondi)	Lento (minuti e molta RAM)
Use Case Ideale	Web App tradizionali, Monoliti, Servizi "sempre accesi".	**Serverless** (AWS Lambda), Container Kubernetes che scalano a zero.

43.5 La Rivoluzione di Spring Boot 3

Rilasciato a Novembre 2022, Spring Boot 3 (basato su Framework 6) rappresenta il cambiamento più traumatico e importante degli ultimi 10 anni. Si fonda su tre pilastri che devi conoscere a fondo: **Jakarta EE**, **GraalVM** e **Micrometer**.

43.5.1 1. La Grande Migrazione: Javax vs Jakarta

Se provi a migrare un progetto da Spring Boot 2.7 a 3.0, il codice non compilerà. Ti troverai migliaia di errori sugli `import`.

Storia di un divorzio legale

Fino al 2017, Java Enterprise Edition (Java EE) era proprietà di **Oracle**. I pacchetti si chiamavano `javax.*` (es. `javax.persistence`). Oracle ha donato Java EE alla **Eclipse Foundation**, ma **non ha ceduto il marchio "Java"**. La fondazione ha dovuto rinominare tutto: Java EE è diventato **Jakarta EE** e i pacchetti `javax.*` sono diventati **`jakarta.*`**.

L'Impatto Tecnico

Spring Boot 3 richiede **Jakarta EE 10**.

```
// SPRING BOOT 2 (Old)
import javax.persistence.Entity;
import javax.servlet.http.HttpServletRequest;

// SPRING BOOT 3 (New)
import jakarta.persistence.Entity;
import jakarta.servlet.http.HttpServletRequest;
```

Il cambiamento degli Import

Deep Dive: Tomcat 9 vs Tomcat 10

Questo cambio impatta anche i server.

- **Tomcat 9:** Parla `javax`. Supporta Spring Boot 2.
- **Tomcat 10:** Parla `jakarta`. Supporta Spring Boot 3.

Non puoi deployare un WAR Boot 2 su Tomcat 10: otterrai errori `ClassNotFoundException`.

43.5.2 2. GraalVM e Native Images

Spring Boot 3 è stato progettato ossessivamente per supportare le **Native Images** tramite GraalVM.

43.5.3 3. Observability: Micrometer

Nei sistemi distribuiti, monitorare è difficile. **Micrometer** è diventato lo standard unificato dentro Spring Boot 3 per due aspetti:

Metrics (Numeri)

Raccoglie dati come "Utilizzo CPU", "Numero richieste HTTP", "Errori 500" e li invia a sistemi come Prometheus o Datadog. È un'interfaccia facade: cambi il sistema di monitoraggio senza cambiare il codice.

Tracing (Il viaggio della richiesta)

Prima si usava Spring Cloud Sleuth. Ora è tutto nativo in Micrometer Tracing. Assegna automaticamente ID alle richieste per seguirle tra i microservizi:

- **Trace ID:** Identifica l'intera transazione attraverso N servizi.
- **Span ID:** Identifica la singola operazione (es. query al DB).

43.6 Tabella Riepilogativa Versioni

Da stampare e tenere sulla scrivania.

Versione Boot	Spring Fwk	Java Min	Note Chiave
2.7.x	5.3.x	Java 8	Ultima versione ramo 2.x. "Ponte" per la migrazione. Usa `javax`.
3.0	6.0	**Java 17**	Jakarta EE namespace, AOT/GraalVM nativo.
3.2	6.1	Java 17/21	Supporto ai **Virtual Threads** (Project Loom).
4.0	7.0	Java 17+	Fortemente raccomandato Java 25. Richiede Jakarta EE 11 (Tomcat 11+).

Deep Dive: Spring Boot 3.2 e i Virtual Threads

Da Spring Boot 3.2 (con Java 21), Spring supporta i **Virtual Threads**. Basta una riga: `spring.threads.virtual.enabled=true`. Tomcat smette di usare i thread pesanti del sistema operativo e usa i virtual threads leggeri della JVM, permettendo di gestire milioni di connessioni concorrenti con lo stile di programmazione classico, senza dover riscrivere tutto in Reactive/WebFlux.

Capitolo 44

Spring JDBC Template: SQL Control & Performance

Abbiamo visto come Hibernate/JPA semplifichi la vita mappando oggetti. Ma c'è un prezzo da pagare: overhead di memoria, query imprevedibili e difficoltà nel gestire operazioni massive (Batch).

Spring JDBC Template è il bisturi chirurgico del framework. Non è un ORM. È un layer sottile sopra il JDBC standard che rimuove il "boilerplate code" (aprire/chiudere connessioni, gestire eccezioni) lasciandoti il **controllo totale dell'SQL**.

44.1 Quando scegliere JDBC Template vs JPA?

Il Dilemma Architetturale

Usa JPA/Hibernate quando:

- Stai facendo CRUD standard (Create, Read, Update, Delete).
- Hai un grafo di oggetti complesso da caricare/salvare.
- Vuoi sfruttare la cache di primo/secondo livello.

Usa Spring JDBC Template quando:

- **Performance Estreme:** Devi inserire 100.000 record in pochi secondi (Batch).
- **SQL Complesso:** Reportistica, Window Functions, CTE, Join su 10 tabelle dove JPA impazzirebbe.
- **Controllo:** Vuoi sapere esattamente quale query viene eseguita, byte per byte.

44.2 JdbcTemplate vs NamedParameterJdbcTemplate

La differenza fondamentale risiede nella gestione dei parametri: il classico `JdbcTemplate` usa il binding posizionale, mentre il `NamedParameterJdbcTemplate` usa il binding nominale.

44.2.1 Approccio Classico (JdbcTemplate)

Utilizza il segnaposto `?` (standard JDBC). I parametri dipendono rigorosamente dall'ordine in cui vengono passati nell'array. Questo approccio è fragile: basta scambiare due parametri dello stesso tipo per corrompere i dati senza errori di compilazione.

```
// Rischioso: l'ordine dei ? e' l'unica garanzia
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
```

```
jdbcTemplate.update(sql, "Mario", "mario@email.com");
```

Listing 44.1: Binding Posizionale

44.2.2 Approccio Moderno (NamedParameterJdbcTemplate)

Utilizza i placeholder con nome (es. `:email`). L'ordine non ha importanza, conta solo la corrispondenza della chiave. Inoltre, permette di mappare automaticamente interi oggetti Java sui parametri della query.

```
// Sicuro: mappa esplicitamente il parametro al valore
String sql = "INSERT INTO users (name, email) VALUES (:name, :email)";

MapSqlParameterSource params = new MapSqlParameterSource()
    .addValue("email", "mario@email.com")
    .addValue("name", "Mario");

namedJdbcTemplate.update(sql, params);
```

Listing 44.2: Binding Nominale

44.2.3 Perché usare NamedParameterJdbcTemplate?

- **Leggibilità:** Il codice SQL dichiara esplicitamente cosa si aspetta (es. `:scadenza` vs `?`).
- **Refactoring:** Aggiungere o rimuovere colonne non richiede di rinumerare gli indici dei parametri.
- **Supporto POJO:** Tramite `BeanPropertySqlParameterSource`, è possibile passare un intero oggetto Java e Spring mapperà automaticamente i campi ai parametri SQL con lo stesso nome.

44.3 Lettura Dati (The RowMapper)

In JDBC non c'è "magia". Devi spiegare a Spring come trasformare una riga del ResultSet (SQL) in un Oggetto Java. Questo si fa con il **RowMapper**.

44.3.1 1. Lettura Singola (QueryForObject)

```
public Optional<UserDTO> findByEmail(String email) {
    String sql = "SELECT id, username, email FROM users WHERE email = :email";

    MapSqlParameterSource params = new MapSqlParameterSource()
            .addValue("email", email);

    try {
        UserDTO user = jdbcTemplate.queryForObject(sql, params, new UserRowMapper());
        return Optional.ofNullable(user);
    } catch (EmptyResultDataAccessException e) {
        return Optional.empty(); // Gestione esplicita del "Not Found"
    }
}
```

44.3.2 2. Il RowMapper (Manuale vs Automatico)

Deep Dive: RowMapper vs BeanPropertyRowMapper

Esiste una classe chiamata `BeanPropertyRowMapper` che mappa automaticamente le colonne ai campi usando la Reflection. **Consiglio Senior:** Non usarla in loop critici o ad alto traffico. La Reflection è lenta. Scrivi il tuo Mapper manuale (è noioso, ma è il più veloce in assoluto).

```
// Definizione pulita e riutilizzabile
private static final RowMapper<UserDTO> USER_MAPPER = (rs, rowNum) -> {
    return new UserDTO(
        rs.getLong("id"),
        rs.getString("username"),
        rs.getString("email")
    );
};

// Utilizzo per liste
public List<UserDTO> findAllActive() {
    String sql = "SELECT * FROM users WHERE active = true";
    // params vuoti
    return jdbcTemplate.query(sql, MapSqlParameterSource.create(), USER_MAPPER);
}
```

Best Practice: Mapper come Lambda o Classe Statica

44.4 Scrittura Dati e Chiavi Generate

Le INSERT sono banali, tranne quando devi recuperare l'ID autogenerato (AUTO_INCREMENT) dal database. In JPA è automatico, qui serve il **KeyHolder**.

```
public Long createUser(CreateUserRequest req) {
    String sql = """
        INSERT INTO users (username, email, password)
        VALUES (:username, :email, :pwd)
    """;

    MapSqlParameterSource params = new MapSqlParameterSource()
            .addValue("username", req.username())
            .addValue("email", req.email())
            .addValue("pwd", req.password()); // Hashala prima!

    KeyHolder keyHolder = new GeneratedKeyHolder();

    // update restituisce il numero di righe modificate
    jdbcTemplate.update(sql, params, keyHolder);

    // Recupero magico dell'ID
    return Objects.requireNonNull(keyHolder.getKey()).longValue();
}
```

Insert con ritorno dell'ID

44.5 Batch Processing: La Killer Feature

Questo è il motivo principale per cui si usa JDBC Template. Se devi inserire 10.000 record:

- **JPA `saveAll`:** Spesso esegue 10.000 insert separate o usa dirty checking pesante. Lento.
- **JDBC Batch:** Invia un unico pacchetto al DB. Velocità 10x-50x superiore.

```
public void bulkInsert(List<UserDTO> users) {
    String sql = "INSERT INTO users (username, email) VALUES (:username, :email)";

    // 1. Trasformiamo la lista di oggetti in un array di parametri
    SqlParameterSource[] batchParams = SqlParameterSourceUtils.createBatch(users);

    // 2. Esecuzione atomica
    int[] updateCounts = jdbcTemplate.batchUpdate(sql, batchParams);

    System.out.println("Inseriti " + updateCounts.length + " record.");
}
```

Batch Insert Ottimizzato

44.6 Gestione delle Eccezioni

JDBC standard lancia `SQLException`, che è una *Checked Exception* (ti obbliga al try-catch ovunque). Spring JDBC Template avvolge tutto e lancia `DataAccessException`, che è *Unchecked* (Runtime).

Inoltre, Spring traduce i codici di errore criptici del DB (es. ORA-0001, SQLState 23505) in eccezioni Java parlanti:

- `DuplicateKeyException` (Violazione Unique Constraint)
- `QueryTimeoutException`
- `BadSqlGrammarException`

Colloquio: Exception Translation

Domanda: Perché l'Exception Translation di Spring è vantaggiosa? **Risposta:** Perché disaccoppia il codice dal vendor del Database. Se migri da Oracle a PostgreSQL, i codici di errore SQL cambiano, ma Spring continuerà a lanciarti la stessa `DuplicateKeyException`. Il tuo blocco `try-catch` nel Service non deve essere riscritto.

44.7 Riepilogo e Cheat Sheet

Operazione	Metodo JdbcTemplate
Leggere 1 riga (o null)	`queryForObject` (gestire `EmptyResult...`)
Leggere N righe	`query` (ritorna `List<T>`)
Insert / Update / Delete	`update`
Insert con ritorno ID	`update` con `KeyHolder`
Insert Massivo	`batchUpdate`
Eseguire DDL (Create table)	`execute`

Deep Dive: Senior Tip: Transazioni

Ricorda che `JdbcTemplate` partecipa pienamente al Transaction Manager di Spring. Se metti `@Transactional` sul metodo del Service che chiama il DAO JDBC, tutte le operazioni (anche i batch) saranno atomiche. Se il batch fallisce a metà, viene fatto il rollback di tutto.

Capitolo 45

Spring Data JPA: The Abstraction Layer

Se JDBC Template è il bisturi, **Spring Data JPA** è il pilota automatico. Riduce il codice di accesso ai dati del 90%, eliminando completamente l'implementazione dei DAO.

Ma attenzione: Spring Data JPA **non è** un ORM. È un livello di astrazione aggiuntivo che si posiziona sopra Hibernate (che è l'ORM), il quale a sua volta sta sopra JDBC.

45.1 La Gerarchia: Capire lo Stack

Ai colloqui chiedono spesso la differenza tra JPA, Hibernate e Spring Data.

1. **JPA (Java Persistence API):** È solo una specifica (un documento PDF e interfacce). Non fa nulla da sola.
2. **Hibernate:** È l'implementazione della specifica JPA. È il motore che genera l'SQL.
3. **Spring Data JPA:** È un framework che genera automaticamente i Repository (le classi DAO) basandosi sulle interfacce che definisci.

45.2 Il Repository Pattern

Invece di scrivere classi `UserDaoImpl`, definisci solo un'interfaccia.

```
// Estendendo JpaRepository erediti GRATIS:
// save, saveAll, findById, findAll, delete, count, existsById...
public interface UserRepository extends JpaRepository<UserEntity, Long> {
    // Il corpo è vuoto! Spring crea un proxy a runtime che implementa tutto.
}
```

Il potere di JpaRepository

45.3 Query Creation Strategies

Spring Data offre tre modi per definire le query. Un Senior sa quando usarne uno e quando passare all'altro.

45.3.1 1. Derived Queries (Query Methods)

Spring analizza il **nome del metodo** e genera l'SQL.

```java
// SELECT * FROM users WHERE email = ? AND active = true
Optional<UserEntity> findByEmailAndActiveTrue(String email);

// SELECT * FROM users WHERE created_at > ? ORDER BY username DESC
List<UserEntity> findByCreatedAtAfterOrderByUsernameDesc(LocalDateTime date);
```

Pro: Velocissimo da scrivere. **Contro:** Se la query è complessa, il nome del metodo diventa illeggibile (`findByUserAddressCityAndActiveTrueAnd...`).

45.3.2 2. @Query (JPQL e Native)

Quando il nome del metodo diventa troppo lungo, usa `@Query`. Qui scrivi in **JPQL** (Java Persistence Query Language), che lavora sulle *Entity*, non sulle tabelle.

```java
@Query("SELECT u FROM UserEntity u WHERE u.email = :email AND u.active = true")
Optional<UserEntity> cercaUtenteAttivo(@Param("email") String email);
```

Deep Dive: Native Queries

Se devi usare funzionalità specifiche del DB (es. JSONB di Postgres o funzioni proprietarie), puoi usare SQL nativo: `@Query(value = "SELECT * FROM users u ...", nativeQuery = true)`. Tuttavia, perdi la portabilità del codice tra database diversi.

45.4 Paginazione e Ordinamento

Non fare mai `findAll()` su una tabella con un milione di righe. Spring Data ha il supporto nativo per la paginazione.

```java
// Nel Repository
Page<UserEntity> findByActiveTrue(Pageable pageable);

// Nel Service (Pagina 0, 10 elementi, ordinati per ID decrescente)
public Page<UserDTO> getUsers(int page, int size) {
    Pageable pageRequest = PageRequest.of(page, size, Sort.by("id").descending());

    Page<UserEntity> userPage = repository.findByActiveTrue(pageRequest);

    // map() converte automaticamente il contenuto della pagina
    return userPage.map(mapper::toDTO);
}
```

Pagination Repository

Colloquio: List vs Page vs Slice

Domanda: Che differenza c'è nel ritornare `Page<T>` o `Slice<T>`? **Risposta:**

- **`Page<T>`:** Esegue la query dei dati **PIÙ** una query `COUNT(*)` separata per sapere il numero totale di pagine. È costoso su grandi dataset.
- **`Slice<T>`:** Esegue solo la query dei dati (spesso chiedendo N+1 righe per sapere se c'è una pagina successiva). Non sa quante pagine ci sono in totale. Ideale per "Infinite Scroll" (Mobile/Social).

45.5 Performance: Il problema N+1

Il killer silenzioso delle applicazioni Spring Data JPA.

Scenario: Hai `User` (1) → `Orders` (N). La relazione è `LAZY` (Best Practice).

1. Chiami `findAll()` sugli utenti (1 Query).
2. Cicli sugli utenti e chiami `user.getOrders().size()`.
3. Hibernate esegue **una query aggiuntiva per OGNI utente** per caricare gli ordini.
4. Se hai 100 utenti, fai 101 Query.

45.5.1 Soluzione 1: @EntityGraph

Dici a Spring di caricare la relazione LAZY in una singola query (EAGER fetch) solo per quel metodo.

```
// Override del findAll per caricare anche gli ordini in un colpo solo
@EntityGraph(attributePaths = {"orders"})
List<UserEntity> findAll();
```

45.5.2 Soluzione 2: JPQL Fetch Join

```
@Query("SELECT u FROM UserEntity u JOIN FETCH u.orders")
List<UserEntity> findAllWithOrders();
```

45.6 Modifying Queries (Update/Delete massivi)

Di default, le query JPQL sono solo di lettura (SELECT). Se vuoi fare UPDATE o DELETE massivi senza caricare le entità in memoria:

```
@Modifying // Obbligatorio per INSERT/UPDATE/DELETE
@Query("UPDATE UserEntity u SET u.active = false WHERE u.lastLogin < :date")
int disattivaUtentiInattivi(@Param("date") LocalDateTime date);
```

Nota: Questo metodo scavalca il contesto di persistenza. Le entity in memoria non saranno aggiornate. Spesso si usa insieme a `@Modifying(clearAutomatically = true)` per pulire la cache di primo livello.

45.7 JPA Auditing

Non gestire manualmente `createdAt` e `updatedAt`.

1. Aggiungi `@EnableJpaAuditing` sulla classe main.
2. Usa le annotazioni sulle Entity (o su una classe base `BaseEntity`).

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class BaseEntity {

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdAt;

    @LastModifiedDate
```

```
    private LocalDateTime updatedAt;

    // Se usi Spring Security, popola automaticamente l'utente!
    @CreatedBy
    private String createdBy;
}
```

45.8 Riepilogo Best Practices

Feature	Consiglio Senior
Derived Queries	Solo per query semplici (max 2 parametri).
Relazioni	Sempre `LAZY` di default. Usa `@EntityGraph` o `JOIN FETCH` quando servono i dati.
Loop su relazioni	Mai farlo dentro un ciclo for senza fetch join (N+1 Problem).
Paginazione	Non ritornare mai `List` su tabelle potenzialmente grandi. Usa `Pageable`.
Projections	Usa Interfacce o DTO Records nelle query per evitare di caricare Entity intere in sola lettura.

Capitolo 46

Spring Transaction Management

Gestire le transazioni manualmente (aprire connessione, `con.setAutoCommit(false)`, commit, rollback, close) è verboso e incline a errori. Spring offre un approccio dichiarativo tramite l'annotazione `@Transactional`.

Tuttavia, usare questa annotazione senza capire cosa succede "sotto il cofano" è la causa principale di bug silenziosi e dati corrotti in produzione.

46.1 Under the Hood: Il Proxy Pattern

Questa è la prima domanda tecnica ai colloqui. **Come funziona @Transactional?** Spring non riscrive il bytecode della tua classe "magicamente". Usa il pattern **AOP (Aspect Oriented Programming)** tramite **Proxy**.

Quando annoti un metodo (o una classe) con `@Transactional`, Spring crea un oggetto wrapper (Proxy) attorno alla tua classe.

Deep Dive: Il flusso di chiamata del Proxy

Immagina di avere un `UserService`.

1. Il Controller chiama `userService.save(user)`.
2. In realtà, sta chiamando il **Proxy** generato da Spring.
3. Il Proxy apre la transazione DB.
4. Il Proxy chiama il metodo reale `save()` della tua classe.
5. Se il metodo reale finisce con successo, il Proxy esegue il **COMMIT**.
6. Se il metodo lancia un'eccezione (Runtime), il Proxy esegue il **ROLLBACK**.

Colloquio: La Trappola della Self-Invocation

Domanda: Ho un metodo A non transazionale che chiama un metodo B annotato con `@Transactional` *nella stessa classe*. La transazione partirà?

```
public void metodoA() {
    metodoB(); // Chiamata interna
}

@Transactional
public void metodoB() { ... }
```

Risposta: NO. La chiamata `metodoB()` avviene tramite `this.metodoB()`. Stai chiamando il metodo direttamente sull'istanza, scavalcando il Proxy. Spring non può intercet-

tare la chiamata e non aprirà nessuna transazione. **Soluzione:** Spostare `metodoB` in un altro Service (Bean diverso) o iniettare il Service in se stesso (brutto ma funzionante).

46.2 Propagation Levels (Propagazione)

Il parametro `propagation` definisce come la transazione deve comportarsi se ne esiste già un'altra attiva.

46.2.1 1. REQUIRED (Default)

Il più usato.

- **Se c'è una transazione attiva:** Il metodo si aggancia a quella esistente.
- **Se NON c'è:** Ne crea una nuova.
- **Conseguenza:** Se il metodo interno fallisce, fa rollback di *tutta* la transazione (anche quella del chiamante).

46.2.2 2. REQUIRES_NEW (Transazione Indipendente)

Il metodo sospende la transazione corrente (se esiste) e ne apre una **nuova e indipendente**.

- **Use Case:** Log di Audit o salvataggio errori.
- Esempio: Sto salvando un ordine (TX 1). L'ordine fallisce. Voglio salvare l'errore su tabella `error_logs` (TX 2). Se usassi REQUIRED, il rollback dell'ordine cancellerebbe anche il log dell'errore. Con REQUIRES_NEW, il log viene salvato (committato) anche se l'ordine principale fa rollback.

46.2.3 3. MANDATORY

Esige che ci sia già una transazione aperta. Se non c'è, lancia un'eccezione. Utile per metodi che non devono mai essere chiamati da soli ma solo come parte di un processo più grande.

46.2.4 4. SUPPORTS / NOT_SUPPORTED

- **SUPPORTS:** Se c'è una TX la usa, altrimenti esegue senza TX (modalità lettura non critica).
- **NOT_SUPPORTED:** Sospende la TX corrente ed esegue senza TX (es. invio email lento o chiamata a sistema esterno che non deve tenere bloccata la connessione DB).

46.3 Rollback Rules (Eccezioni)

Spring, di default, fa rollback **SOLO** per le `RuntimeException` (Unchecked) e per gli `Error`. **NON fa rollback** per le `Checked Exception` (es. `IOException`, `SQLException` custom).

```
// Sbagliato (Default): Non fa rollback se lancia IOException
@Transactional
public void salvaFile() throws IOException { ... }

// Corretto: Forza il rollback anche per le Checked Exception
@Transactional(rollbackFor = Exception.class)
public void salvaFileSicuro() throws IOException { ... }

```

```
// Caso opposto: NON fare rollback per una Runtime specifica
@Transactional(noRollbackFor = IllegalArgumentException.class)
public void ignoraErroriMinori() { ... }
```

Configurare il Rollback

46.4 Isolation Levels (Concorrenza)

Cosa succede se due transazioni toccano gli stessi dati contemporaneamente? L'isolamento definisce "quanto" una transazione vede delle modifiche dell'altra.

1. **READ_UNCOMMITTED:** Leggi anche i dati non ancora committati (Dirty Read). Pericolosissimo, quasi mai usato.
2. **READ_COMMITTED (Default in Postgres/Oracle):** Leggi solo i dati committati. Previene Dirty Read.
3. **REPEATABLE_READ (Default in MySQL):** Assicura che se leggi una riga due volte nella stessa TX, otterrai lo stesso valore (previene *Non-Repeatable Read*).
4. **SERIALIZABLE:** Il livello massimo. Esegue le transazioni come se fossero sequenziali. Lento (uso pesante di lock), ma garantisce consistenza totale (previene *Phantom Read*).

46.5 Read-Only Optimization

```
@Transactional(readOnly = true)
public List<User> findAll() { ... }
```

Perché metterlo? Non è solo documentazione.

- **JPA/Hibernate:** Disabilita il *Dirty Checking*. Hibernate non perderà tempo a controllare se hai modificato le entity caricate, risparmiando memoria e CPU.
- **Database (es. MySQL Replicas):** In architetture Master-Slave, il driver JDBC potrebbe instradare le query `readOnly` verso le Repliche (Slave) di lettura, alleggerendo il Master.

46.6 Riepilogo Best Practices

Situazione	Soluzione
Chiamata interna (Self-invocation)	Non usare `@Transactional` (o auto-inject del service).
Metodi privati	`@Transactional` non funziona sui metodi private (il proxy non li vede).
Salvataggio Log/Audit indipendenti	Usa `propagation = REQUIRES_NEW`.
Checked Exception custom	Usa `rollbackFor = MyException.class`.
Operazioni di sola lettura	Sempre `readOnly = true` per performance.
Transazioni lunghe (es. report)	Evitale. Tengono la connessione al DB occupata. Spostale in Job asincroni.

Capitolo 47

Spring MVC & REST API

47.1 Architettura e il Front Controller

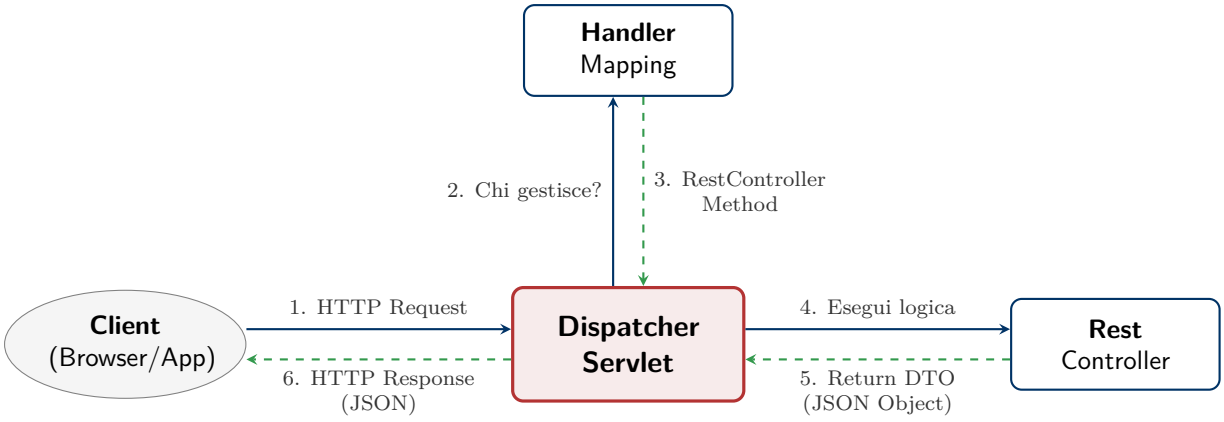
Spring MVC (Model-View-Controller) è il modulo web originale di Spring. Anche se oggi costruiamo principalmente API REST (che non hanno una "View" nel senso di HTML, ma restituiscono JSON), l'architettura sottostante rimane la stessa.

Tutto ruota attorno a un design pattern fondamentale: il **Front Controller Pattern**.

47.1.1 Il DispatcherServlet

In una classica applicazione Java Servlet (pre-Spring), ogni URL doveva essere mappato a una specifica Servlet nel `web.xml`. Questo diventava ingestibile rapidamente.

Spring risolve il problema con un'unica, onnipotente Servlet chiamata **DispatcherServlet**. Questa riceve **tutte** le richieste HTTP in ingresso e ha il compito di smistarle al Controller giusto.



47.1.2 Il Flusso della Richiesta

1. La richiesta arriva al `DispatcherServlet`.
2. Il Dispatcher consulta l'`HandlerMapping` per sapere quale metodo di quale Controller deve invocare (basandosi su URL e verbo HTTP).
3. Il Dispatcher invoca il **Controller**.
4. Il Controller esegue la logica di business (spesso chiamando un *Service*).
5. Il Controller restituisce un oggetto (es. `UserDto`).

6. Poiché è una REST API (`@RestController`), Spring usa un `HttpMessageConverter` (di solito **Jackson**) per trasformare l'oggetto in JSON.
7. La risposta JSON torna al Client.

Colloquio: DispatcherServlet e ApplicationContext

Domanda: "Come fa il DispatcherServlet a conoscere i tuoi Bean?"

Risposta: All'avvio di Spring Boot, il DispatcherServlet viene inizializzato e collegato all'`WebApplicationContext`. Questo contesto contiene tutti i bean definiti con `@Controller`, `@Service`, `@Repository`, rendendoli disponibili per l'iniezione delle dipendenze e per il mapping delle richieste.

47.2 Controller e Endpoint

Il Controller è il componente che espone i punti di accesso (Endpoint) della tua applicazione al mondo esterno. In un'architettura REST, ogni metodo del controller corrisponde solitamente a una combinazione di URL + Verbo HTTP.

47.2.1 @Controller vs @RestController

Spesso si crea confusione su quale annotazione usare.

- **@Controller:** È l'annotazione originale di Spring MVC. I metodi di questa classe, di default, restituiscono una `String` che rappresenta il **nome della vista** (es. "index.html" o "home.jsp"). Per restituire dati grezzi, bisognava aggiungere `@ResponseBody` su ogni metodo.
- **@RestController:** Introdotta in Spring 4.0, è un'annotazione di convenienza che combina `@Controller` + `@ResponseBody`. I metodi restituiscono oggetti che vengono automaticamente serializzati in JSON (o XML) nel corpo della risposta HTTP.

Deep Dive: Dietro le quinte

`@RestController` è definita letteralmente così:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Controller
@ResponseBody
public @interface RestController {}
```

Questo dimostra che è solo "zucchero sintattico".

47.2.2 Definizione degli Endpoint

Per mappare le richieste HTTP ai metodi Java, usiamo le annotazioni di mapping. È buona norma definire il percorso base a livello di classe e i percorsi specifici a livello di metodo.

```
1 @RestController
2 @RequestMapping("/api/v1/users") // Base path per tutti i metodi
3 public class UserController {
4
5     private final UserService userService;
```

```
6
7       // Injection via costruttore (Best Practice)
8       public UserController(UserService userService) {
9           this.userService = userService;
10      }
11
12      // GET /api/v1/users
13      @GetMapping
14      public List<UserDto> getAllUsers() {
15          return userService.findAll();
16      }
17
18      // GET /api/v1/users/{id}
19      @GetMapping("/{id}")
20      public UserDto getUserById(@PathVariable Long id) {
21          return userService.findById(id);
22      }
23
24      // POST /api/v1/users
25      @PostMapping
26      @ResponseStatus(HttpStatus.CREATED) // Restituisce 201 invece di 200
27      public UserDto createUser(@RequestBody UserDto userDto) {
28          return userService.save(userDto);
29      }
30
31      // DELETE /api/v1/users/{id}
32      @DeleteMapping("/{id}")
33      @ResponseStatus(HttpStatus.NO_CONTENT) // Restituisce 204 (Body vuoto)
34      public void deleteUser(@PathVariable Long id) {
35          userService.delete(id);
36      }
37  }
```

Listing 47.1: Struttura tipica di un Controller REST

47.2.3 Gli Stereotipi di Spring

Ai colloqui chiedono spesso la differenza tra le varie annotazioni di componente.

Colloquio: Component - Service - Repository - Controller

Domanda: "Tecnicamente `@Service` e `@Repository` sono alias di `@Component`. Perché li distinguiamo?"

Risposta: È vero che per il Container di Spring sono tutti Bean gestiti allo stesso modo (Singleton, Lazy/Eager, ecc.), ma hanno semantiche e funzionalità extra diverse:

1. **@Component:** Annotazione generica per bean non categorizzabili (es. una classe di Utility, un Mapper).
2. **@Controller / @RestController:** Indica che la classe gestisce richieste Web. Il `DispatcherServlet` scansiona specificamente queste classi per mappare gli URL.
3. **@Service:** Semanticamente indica che la classe contiene "Business Logic". Attualmente non aggiunge funzionalità tecniche extra rispetto a Component, ma serve a chiarire l'architettura.
4. **@Repository:** Indica che la classe interagisce con il DB (DAO pattern). **Funzionalità Extra:** Abilita la *PersistenceExceptionTranslation*. Se il driver JDBC lancia

un'eccezione SQL nativa (es. `SQLException`), Spring la intercetta e la traduce in una `DataAccessException` (unchecked e comune a tutti i DB), disaccoppiando il codice dalla tecnologia sottostante.

47.3 Gestione dei Dati (Request & Response)

Un'API REST è, essenzialmente, un meccanismo per scambiare dati. Spring MVC offre diversi modi per estrarre informazioni dalla richiesta HTTP in ingresso e formattare la risposta in uscita.

47.3.1 Input: PathVariable vs RequestParam

Questa è una distinzione fondamentale nel design delle API REST.

- **@PathVariable:** Si usa quando il dato è parte integrante dell'identità della risorsa (RESTful style). *Esempio:* `/users/123` (123 identifica l'utente).
- **@RequestParam:** Si usa per filtrare, ordinare o configurare la richiesta. I dati viaggiano nella "Query String". *Esempio:* `/users?role=ADMIN&active=true`.

```
// 1. PathVariable (Parte dell'URI)
// Chiamata: GET /products/42
@GetMapping("/products/{id}")
public Product getProduct(@PathVariable Long id) { ... }

// 2. RequestParam (Query String)
// Chiamata: GET /products?category=BOOKS&sort=price
@GetMapping("/products")
public List<Product> searchProducts(
    @RequestParam String category, // Obbligatorio di default
    @RequestParam(required = false, defaultValue = "name") String sort // Opzionale
) { ... }
```

Listing 47.2: Differenza di utilizzo

47.3.2 Input: @RequestBody e Deserializzazione

Per le operazioni di scrittura (POST, PUT), i dati complessi viaggiano nel corpo (Body) della richiesta, solitamente in formato JSON. L'annotazione `@RequestBody` dice a Spring: "Prendi il JSON nel body e trasformalo in questo oggetto Java".

Deep Dive: Il ruolo di Jackson (ObjectMapper)

Spring non fa la conversione da solo. Delega il lavoro a una libreria esterna chiamata **Jackson** (inclusa di default in Spring Boot).

Quando usi `@RequestBody`, dietro le quinte agisce un componente chiamato `MappingJackson2HttpMessageConverter`.

Requisiti affinché funzioni:

1. La classe Java (DTO) deve avere un **Costruttore di Default** (vuoto), affinché Jackson possa istanziarla.
2. I campi devono avere i **Getter/Setter** (o essere pubblici), oppure Jackson non potrà scriverci dentro (a meno di configurazioni specifiche sulla visibilità).

47.3.3 Controllare il JSON: @JsonProperty e @JsonIgnore

Spesso il nome del campo nel database o nella classe Java non deve corrispondere a quello esposto nell'API.

```java
public class UserDto {

    // Nel JSON sarà "user_email" invece di "email"
    @JsonProperty("user_email")
    private String email;

    // Questo campo NON verrà mai inviato al client (Sicurezza)
    @JsonIgnore
    private String password;
}
```

Listing 47.3: Personalizzazione JSON

47.3.4 Output: Content Negotiation

Come fa Spring a sapere se restituire JSON o XML?

Colloquio: Come funziona la Content Negotiation?

Domanda: "La stessa API può restituire i dati sia in JSON che in XML. Come decide Spring quale formato usare?"

Risposta: Spring guarda l'header HTTP **Accept** inviato dal client.

- Se il client invia `Accept: application/json` → Restituisce JSON (Default).
- Se il client invia `Accept: application/xml` → Restituisce XML (se la libreria *Jackson XML* è presente nel classpath).
- Se l'header manca, Spring usa il formato di default (JSON).

Questo meccanismo è gestito dall'interfaccia `HttpMessageConverter`. Spring cicla su tutti i converter disponibili e sceglie il primo capace di scrivere il tipo di risposta richiesto.

47.4 DTO Pattern (Data Transfer Object)

Una regola d'oro nello sviluppo di API REST Enterprise è: **Non esporre mai le tue Entity JPA (@Entity) direttamente come risposta del Controller.**

Le Entity rappresentano il tuo Database (struttura interna). I DTO rappresentano il contratto della tua API (struttura pubblica). Questi due mondi devono rimanere disaccoppiati.

47.4.1 I 3 Grandi Problemi delle Entity nei Controller

Se restituisci 'return userRepository.findAll()' direttamente, incorri in tre problemi fatali:

1. **Sicurezza (Data Leakage):** La tua Entity 'User' contiene probabilmente campi come 'password', 'salt', 'created_at' o flag interni. Se la serializzi in JSON, Jackson invierà tutto al client, inclusa la password hashata. Usare '@JsonIgnore' sull'Entity è una pezza, non una soluzione (sporchi il modello di dominio con logica di presentazione).
2. **Accoppiamento (Tight Coupling):** Se domani rinomini una colonna nel DB o spacchi una tabella in due per performance, spacchi automaticamente l'API usata dai client Front-end. Col DTO, puoi cambiare il DB quanto vuoi, basta rimappare il DTO correttamente.
3. **Eccezioni di Serializzazione (Infinite Recursion):** Questo è un classico.

Deep Dive: Deep Dive: Infinite Recursion e Lazy Loading

Le Entity hanno spesso relazioni bidirezionali (es. Utente ↔ Ordini).

- **Infinite Recursion:** Jackson inizia a serializzare l'Utente, trova la lista di Ordini, entra nell'Ordine, trova il riferimento all'Utente, torna all'Utente... Boom: `StackOverflowError`.
- **LazyInitializationException:** Se hai una relazione `FetchType.LAZY`, i dati non sono caricati in memoria. Quando Jackson prova a leggerli nel Controller (fuori dalla transazione del Service), Hibernate lancia l'eccezione perché la sessione col DB è chiusa.

Soluzione: Il DTO è un POJO piatto (senza logica JPA), quindi Jackson lo serializza senza problemi.

47.4.2 Implementazione: Entity vs DTO

```java
// --- ENTITY (Modello DB) ---
@Entity
public class User {
    @Id
    private Long id;
    private String username;
    private String password; // PERICOLO!

    @OneToMany(mappedBy = "user")
    private List<Order> orders; // Rischio Lazy/Recursion
}

// --- DTO (Modello API) ---
public class UserResponseDto {
    private Long id;
    private String username;
    // Niente password!
    private int ordersCount; // Magari al client serve solo il numero, non la lista
}
```

Listing 47.4: Il problema (Entity) vs La soluzione (DTO)

47.4.3 Strategie di Mapping

Come copiamo i dati dall'Entity al DTO?

Colloquio: MapStruct vs ModelMapper vs BeanUtils

Domanda: "Come gestisci la conversione tra Entity e DTO?"

Risposta: Evito `BeanUtils` (copia shallow, pericolosa) e `ModelMapper` (usa Reflection a runtime, lento e difficile da debuggare).

Preferisco **MapStruct**.

- Lavora a **Compile Time** (genera codice Java reale durante la build).
- È **Type-Safe**: se cambi un campo e dimentichi di aggiornare il mapper, la build fallisce (ottimo per il refactoring).
- È estremamente **veloce** (zero overhead di reflection).

```java
@Mapper(componentModel = "spring")
public interface UserMapper {

    UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);

    // Mapping automatico per nomi uguali
    @Mapping(target = "userId", source = "id")
    UserDto toDto(User entity);

    User toEntity(UserDto dto);
}
```

Listing 47.5: Esempio MapStruct

47.5 Validazione dei Dati

Uno dei principi cardine della sicurezza e della stabilità è: **Mai fidarsi dell'input utente**. Invece di riempire il codice di controlli manuali (‘if (dto.getEmail() == null)...‘), Spring Boot supporta lo standard **Jakarta Validation** (implementato da Hibernate Validator).

Questo permette di definire le regole di validazione in modo *dichiarativo* direttamente sui campi del DTO tramite annotazioni.

47.5.1 Le Annotazioni Principali

Per attivare la validazione, è necessario aggiungere la dipendenza `spring-boot-starter-validation`. Ecco le annotazioni più comuni:

```java
public class UserRegistrationDto {

    @NotBlank(message = "Il nome non può essere vuoto")
    private String firstName;

    @Email(message = "Formato email non valido")
    @NotBlank
    private String email;

    @Size(min = 8, max = 20, message = "La password deve essere tra 8 e 20 caratteri")
    private String password;

    @Min(value = 18, message = "Devi essere maggiorenne")
    private int age;

    // @Pattern per Regex complesse (es. Codice Fiscale)
    @Pattern(regexp = "^[A-Z]{6}[0-9]{2}...", message = "Codice Fiscale invalido")
    private String fiscalCode;
}
```

Listing 47.6: DTO con Validazione

Colloquio: NotNull vs NotEmpty vs NotBlank

Domanda: "Qual è la differenza tra queste tre annotazioni per le Stringhe? Quale useresti per un campo 'username'?"

Risposta: È una trappola classica.

- **@NotNull:** Impedisce solo il valore `null`. Accetta stringhe vuote `""` o stringhe di soli spazi `" "`. Spesso inutile da solo per le stringhe.
- **@NotEmpty:** Impedisce `null` e stringhe di lunghezza 0 (`""`). Accetta però stringhe di soli spazi `" "`.
- **@NotBlank:** È la più restrittiva. Impedisce `null`, vuote e stringhe composte **solo da spazi**. Inoltre **trimma** la stringa prima del controllo.

Conclusione: Per i campi testuali obbligatori (nome, email, username), si usa quasi sempre **@NotBlank**.

47.5.2 Attivare la Validazione nel Controller

Aver annotato il DTO non basta. Bisogna istruire Spring di eseguire i controlli nel momento in cui riceve la richiesta. Si usa l'annotazione **@Valid** (o `@Validated`) prima del parametro `@RequestBody`.

```
@PostMapping
public UserDto register(@Valid @RequestBody UserRegistrationDto dto) {
    // Se arrivo qui, SONO SICURO che il dto è valido.
    // Non serve fare if(dto.getEmail() == null).
    return userService.register(dto);
}
```

Listing 47.7: Trigger della Validazione

47.5.3 Cosa succede se la validazione fallisce?

Se un campo non rispetta le regole:

1. Il metodo del controller **NON** viene eseguito.
2. Spring lancia automaticamente un'eccezione di tipo **MethodArgumentNotValidException**.
3. Di default, Spring Boot risponde con un HTTP 400 Bad Request e un JSON standard (che però è spesso troppo verboso e poco leggibile per il frontend).

Deep Dive: Gestione Manuale con BindingResult

Esiste un modo per non far lanciare l'eccezione e gestire l'errore manualmente dentro il metodo. Basta aggiungere un oggetto `BindingResult` **immediatamente dopo** il DTO validato.

```
public ResponseEntity<?> create(@Valid @RequestBody UserDto dto,
                                BindingResult result) {
    if (result.hasErrors()) {
        return ResponseEntity.badRequest().body("Errore nei dati!");
    }
    // ...
}
```

Nota: Sebbene utile a fini didattici, nei progetti reali si preferisce lasciar lanciare l'eccezione e gestirla centralmente (come vedremo nella prossima sezione), per tenere pulito il codice del controller.

47.6 Gestione Globale degli Errori (Exception Handling)

Un cattivo esempio di Controller è quello pieno di blocchi `try-catch`. Questo approccio porta a duplicazione del codice e risposte di errore incoerenti (un endpoint restituisce un messaggio stringa, l'altro un JSON, l'altro nulla).

Spring MVC risolve il problema intercettando le eccezioni a livello globale tramite il pattern AOP (Aspect Oriented Programming), usando l'annotazione **@ControllerAdvice** (o la specializzazione **@RestControllerAdvice**).

47.6.1 L'architettura Centralizzata

L'idea è semplice:

1. Il Controller esegue solo l'Happy Path (il caso in cui tutto va bene).
2. Se qualcosa va storto (es. ID non trovato), il Service lancia un'eccezione (es. `UserNotFoundException`).
3. L'eccezione risale la pila fino a uscire dal Controller.
4. Un componente globale cattura l'eccezione e la trasforma in una `ResponseEntity` formattata correttamente.

47.6.2 Implementazione: @RestControllerAdvice

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    // 1. Gestione eccezioni custom (es. Risorsa non trovata)
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleNotFound(ResourceNotFoundException ex)
    ↪ {
        ErrorResponse error = new ErrorResponse(
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            LocalDateTime.now()
        );
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    // 2. Gestione errori di Validazione (@Valid fallito)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleValidationErrors(
            MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();

        // Estrae i messaggi di errore dai campi del DTO
        ex.getBindingResult().getFieldErrors().forEach(error -> {
            errors.put(error.getField(), error.getDefaultMessage());
        });

        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }

    // 3. Catch-all per errori imprevisti (NullPointer, DB down...)
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGenericException(Exception ex) {
```

```
33          // Loggare sempre l'errore reale sui server log!
34          return new ResponseEntity<>("Errore interno del server",
35                                      HttpStatus.INTERNAL_SERVER_ERROR);
36      }
37  }
```

Listing 47.8: Global Exception Handler

47.6.3 Standardizzazione: RFC 7807 (Problem Details)

Inventarsi un formato JSON per gli errori (come l'oggetto `ErrorResponse` sopra) va bene, ma esiste uno standard IETF internazionale chiamato **Problem Details for HTTP APIs** (RFC 7807).

Deep Dive: Novità Spring Boot 3: ProblemDetail

Dalla versione 6 di Spring Framework (Spring Boot 3), c'è il supporto nativo per questo standard. Non serve più creare DTO custom per gli errori.

Puoi restituire direttamente l'oggetto `ProblemDetail`:

```
@ExceptionHandler(UserNotFoundException.class)
public ProblemDetail handleNotFound(UserNotFoundException ex) {
    return ProblemDetail.forStatusAndDetail(
        HttpStatus.NOT_FOUND,
        ex.getMessage()
    );
}
```

Il JSON risultante sarà standard e automaticamente arricchito (con campi come `type`, `title`, `status`, `detail`, `instance`).

Colloquio: Gestione 404 vs 500

Domanda: "Se cerco un utente con ID 99 e non esiste, il server deve restituire 200 con body vuoto, 404 o 500?"

Risposta:

- **Mai 500:** Il 500 è per errori del server (bug, db down). L'assenza di un dato è una situazione gestita.
- **Mai 200 con body vuoto:** È ambiguo. Le REST API usano i codici di stato come protocollo di comunicazione.
- **Corretto: 404 Not Found.**

Eccezione: Per una ricerca su una lista (es. `/users?name=X`), se non trovo nulla restituisco **200 OK** con una lista vuota `[]`, non 404. Il 404 si usa quando la risorsa specifica (identificata dall'URL) non esiste.

47.7 Teoria REST e Best Practices

Scrivere un'API REST non significa solo restituire JSON invece di HTML. REST (Representational State Transfer) è uno stile architetturale basato su principi precisi. Ignorare questi principi porta a creare API "RPC over HTTP" (una collezione di URL disordinati) invece di vere risorse RESTful.

47.7.1 Il Modello di Maturità di Richardson

È una scala da 0 a 3 usata per classificare quanto un'API è aderente ai principi REST.

- **Livello 0 (The Swamp of POX):** Si usa HTTP solo come tunnel. Esiste un solo URL (es. `/apiService`) e si usa sempre POST per tutto (stile SOAP/RPC).
- **Livello 1 (Resources):** Si introducono URL distinti per risorse diverse (es. `/users`, `/products`).
- **Livello 2 (HTTP Verbs):** Si usano i verbi corretti per le azioni (GET per leggere, POST per creare, DELETE per rimuovere). Questo è il livello della maggior parte delle API commerciali.
- **Livello 3 (HATEOAS):** *Hypermedia As The Engine Of Application State.* La risposta API contiene non solo i dati, ma anche i link per le possibili azioni successive (es. "next_page", "delete_user"). È il "Santo Graal" del REST, ma complesso da implementare.

47.7.2 PUT vs PATCH: La differenza semantica

Questa è forse la domanda numero 1 sui verbi HTTP.

Colloquio: Differenza tra PUT e PATCH

Domanda: "Voglio modificare l'email di un utente. Uso PUT o PATCH?"

Risposta: Dipende da come implementi l'aggiornamento.

- **PUT (Sostituzione Completa):** Il client invia l'intera risorsa modificata. Se l'oggetto ha 10 campi e ne invii solo 1 (l'email), gli altri 9 dovrebbero essere resettati a `null` (o cancellati). È idempotente.
- **PATCH (Aggiornamento Parziale):** Il client invia solo i campi che cambiano. Se invii solo l'email, il server aggiorna l'email e mantiene inalterati gli altri 9 campi. Non è necessariamente idempotente.

Best Practice: Per modifiche puntuali (es. cambio password o email), si preferisce **PATCH**.

47.7.3 Codici di Stato HTTP (Status Codes)

Non restituire sempre `200 OK`! Il codice di stato è parte integrante della risposta.

- **2xx Success**
  - **200 OK:** Operazione riuscita (generico).
  - **201 Created:** Risorsa creata con successo (Risposta a una POST).
  - **204 No Content:** Operazione riuscita, ma non c'è nulla da restituire (tipico della DELETE).
- **4xx Client Error**
  - **400 Bad Request:** Errore generico di sintassi.
  - **401 Unauthorized:** Non sei loggato (manca il token).
  - **403 Forbidden:** Sei loggato ma non hai i permessi (manca il ruolo).
  - **404 Not Found:** Risorsa inesistente.
  - **422 Unprocessable Entity:** I dati sono sintatticamente corretti (è un JSON valido), ma semanticamente errati (es. validazione fallita). Molto usato in Spring.
- **5xx Server Error**
  - **500 Internal Server Error:** Bug nel codice, NullPointer non gestita, DB irraggiungibile.

47.7.4 Idempotenza

Un concetto avanzato che distingue i Senior Developer.

Deep Dive: Cos'è l'Idempotenza?

Un'operazione è **idempotente** se eseguirla più volte produce lo stesso effetto di eseguirla una volta sola.

$$f(f(x)) = f(x)$$

Esempi:

- **DELETE (Idempotente):** Se cancello l'utente ID 5, sparisce. Se riprovo a cancellarlo, non esiste già più (mi darà 404), ma lo stato del server è identico (l'utente 5 non c'è). Non faccio danni.
- **GET (Idempotente):** Leggere 100 volte non cambia i dati.
- **PUT (Idempotente):** "Imposta il nome a 'Mario'". Se lo dico 10 volte, il nome resta 'Mario'.
- **POST (NON Idempotente):** "Crea un ordine". Se la rete cade e il client ritenta l'invio (Retry), potrei creare **due** ordini identici e addebitare due volte il cliente.

Implicazione: Se usi POST, devi gestire manualmente la duplicazione (es. tramite un "Idempotency Key" nell'header).

47.8 Documentazione e Client

Un'API non documentata è un'API che non esiste. Inoltre, un backend developer non si limita a *creare* API, spesso deve *consumare* API esterne (es. Gateway di pagamento, servizi meteo, microservizi interni).

47.8.1 Documentazione: Swagger / OpenAPI

Fino a qualche anno fa si usava la libreria *Springfox* (ormai abbandonata). Oggi lo standard è **Springdoc OpenAPI**. Questa libreria analizza i tuoi Controller al bootstrap e genera automaticamente una pagina web interattiva dove testare le API.

Basta aggiungere la dipendenza `springdoc-openapi-starter-webmvc-ui`. L'interfaccia sarà disponibile su: `http://localhost:8080/swagger-ui/index.html`.

```java
@RestController
@RequestMapping("/api/users")
// Descrizione generale del Controller
@Tag(name = "User API", description = "Gestione anagrafica utenti")
public class UserController {

    @Operation(summary = "Cerca utente per ID",
               description = "Restituisce 404 se l'utente non esiste")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Trovato"),
        @ApiResponse(responseCode = "404", description = "Non trovato")
    })
    @GetMapping("/{id}")
    public UserDto getUser(@PathVariable Long id) { ... }
}
```

Listing 47.9: Arricchire la documentazione

47.8.2 Consumare API Esterne: L'evoluzione dei Client

Spring ha cambiato strategia diverse volte negli anni. È importante conoscere la storia per scegliere lo strumento giusto.

Colloquio: RestTemplate vs WebClient vs RestClient

Domanda: "Devo chiamare un servizio esterno da un'app Spring Boot 3. Cosa uso?"

Risposta:

1. **RestTemplate:** È il client storico (bloccante). È in *Maintenance Mode*. Non verranno aggiunte nuove feature. Si usa solo su progetti legacy.
2. **WebClient:** È il client moderno e Reattivo (non-bloccante). È potente, ma richiede la dipendenza `spring-boot-starter-webflux`. Usarlo in un'app MVC classica è spesso "overkill" (porta dentro Netty e tutto lo stack reattivo inutilmente).
3. **RestClient (La Scelta Giusta):** Introdotto in **Spring Boot 3.2**. Offre la stessa sintassi fluente e moderna di WebClient, ma funziona sullo stack Servlet classico (bloccante). È il successore spirituale di RestTemplate.

47.8.3 Esempio: Usare il nuovo RestClient

Ecco come fare una chiamata HTTP in modo moderno, elegante e tipizzato, senza importare WebFlux.

```
@Service
public class WeatherService {

    private final RestClient restClient;

    public WeatherService(RestClient.Builder builder) {
        // Configurazione base (URL, Timeout, Auth)
        this.restClient = builder
                .baseUrl("https://api.weather.com")
                .defaultHeader("Authorization", "Bearer my-token")
                .build();
    }

    public WeatherDto getWeather(String city) {
        return restClient.get()
                .uri("/forecast/{city}", city)
                .retrieve()
                // Gestione errori specifica per status code
                .onStatus(HttpStatusCode::is4xxClientError, (req, resp) -> {
                    throw new CityNotFoundException();
                })
                .body(WeatherDto.class); // Deserializzazione automatica JSON ->
    ↪ Oggetto
    }
}
```

Listing 47.10: Chiamata GET con RestClient

Capitolo 48

Spring Cache, H2 & Redis: Ottimizzazione Performance

In un'applicazione enterprise, il collo di bottiglia è quasi sempre il Database o la Rete. Recuperare un dato dalla RAM richiede nanosecondi. Recuperarlo dal DB (su disco o via rete) richiede millisecondi. C'è un fattore di differenza di $100.000x$.

Il **Caching** è l'arte di memorizzare risultati costosi in una memoria temporanea veloce per riutilizzarli.

48.1 Spring Cache Abstraction

Spring non implementa una cache. Fornisce un'**astrazione**. Funziona esattamente come JDBC (che astrae i driver DB) o Slf4j (che astrae i log).

Tu usi le annotazioni standard (`@Cacheable`), e "sotto il cofano" puoi cambiare il motore di cache (HashMap, Redis, Caffeine, EhCache) cambiando solo una riga nel file di configurazione.

48.1.1 Le Annotazioni Magiche

Per attivare il sistema, devi aggiungere `@EnableCaching` su una classe di configurazione o sulla Main class.

Ecco le tre annotazioni che devi conoscere:

1. @Cacheable (Lettura - Look-aside Pattern)

È la più usata. Intercetta la chiamata al metodo.

- **Prima di eseguire il metodo:** Spring controlla se nella cache esiste già un valore per quella chiave.
- **Se esiste (Cache Hit):** Restituisce il valore immediato. Il metodo NON viene eseguito (risparmiando la query al DB).
- **Se non esiste (Cache Miss):** Esegue il metodo reale, salva il risultato in cache e lo restituisce.

```
@Service
public class ProductService {

    // Cache Name: "prodotti"
    // Key: generata automaticamente basandosi sui parametri (es. id)
    @Cacheable(value = "prodotti", key = "#id")
    public ProductDTO getProductById(Long id) {
```

```
        // Simuliamo operazione lenta (DB call)
        simulateSlowService();
        return repository.findById(id).map(mapper::toDto).orElse(null);
    }
}
```

2. @CacheEvict (Pulizia)

Le cache devono essere invalidare quando i dati cambiano, altrimenti l'utente vedrà dati vecchi (*Stale Data*). `@CacheEvict` rimuove elementi dalla cache.

```
    // Quando aggiorno un prodotto, DEVO cancellarlo dalla cache
    // altrimenti le prossime getProductById ritorneranno il vecchio oggetto!
    @CacheEvict(value = "prodotti", key = "#dto.id")
    public ProductDTO updateProduct(ProductDTO dto) {
        return repository.save(mapper.toEntity(dto));
    }

    // Svuota INTERA cache (es. dopo un import massivo notturno)
    @CacheEvict(value = "prodotti", allEntries = true)
    public void clearCache() { ... }
```

3. @CachePut (Aggiornamento)

Meno usata. Esegue *sempre* il metodo e aggiorna la cache con il nuovo risultato. Utile se vuoi evitare il "buco" temporaneo causato da una Evict.

48.2 H2 Database: In-Memory Relational DB

Spesso confuso con una cache, **H2** è un vero RDBMS SQL, ma risiede interamente nella RAM (o su file locale).

48.2.1 Quando usarlo?

- **Test:** Per i test di integrazione (‘@DataJpaTest‘) è perfetto perché parte in millisecondi e si pulisce ad ogni riavvio.
- **Sviluppo Locale:** Evita di dover installare Postgres/MySQL sul laptop per fare due prove.

48.2.2 Configurazione e Console

Spring Boot autoconfigura H2 se trova la dipendenza.

```
# Abilita la console web su /h2-console
spring.h2.console.enabled=true
# URL per connettersi (mem = in memoria, perde dati al riavvio)
spring.datasource.url=jdbc:h2:mem:testdb
```

application.properties

Deep Dive: H2 vs Cache

Perché usare `@Cacheable` se uso già H2 (che è in RAM)? Anche se H2 è veloce, è comunque un database SQL. Ogni query richiede parsing dell'SQL, ottimizzazione del piano di esecuzione e trasformazione dei ResultSet. Una Cache (come Redis o HashMap) è un accesso Key-Value diretto ($O(1)$). È ordini di grandezza più veloce di una query SQL su H2.

48.3 Redis: The Production Cache

In produzione (Kubernetes, Microservizi), non puoi usare la cache in memoria semplice (HashMap). Se hai 10 istanze del tuo backend, e l'istanza A mette in cache un dato, l'istanza B non lo sa. Serve una **Cache Distribuita**.

Redis (Remote Dictionary Server) è lo standard industriale. È un "Data Structure Store" in-memory.

48.3.1 Perché Redis?

1. **Centralizzato:** Tutte le istanze dei microservizi leggono/scrivono dallo stesso Redis.
2. **TTL (Time To Live):** Puoi dire "tieni questo dato solo per 10 minuti". Fondamentale per non riempire la RAM.
3. **Persistenza:** A differenza di Memcached, Redis può salvare su disco, quindi se il server si riavvia, la cache calda non è persa (opzionale).

48.3.2 Integrazione Spring Boot + Redis

Basta aggiungere `spring-boot-starter-data-redis`. Spring rileva Redis e configura automaticamente il `CacheManager` per usare Redis invece della HashMap.

```
spring.cache.type=redis
spring.data.redis.host=localhost
spring.data.redis.port=6379
# Default TTL (es. 60000ms = 1 minuto)
spring.cache.redis.time-to-live=60000
```

application.properties

48.3.3 Serializzazione: Il problema dei binari

Di default, Spring usa la serializzazione Java nativa (ObjectInputStream), che produce blob binari illeggibili in Redis e fragili. Un Senior Developer configura sempre un **Serializer JSON**.

```
@Bean
public RedisCacheConfiguration cacheConfiguration() {
    return RedisCacheConfiguration.defaultCacheConfig()
        .entryTtl(Duration.ofMinutes(10)) // TTL Globale
        .disableCachingNullValues()
        // Usa JSON leggibile invece di binario Java
        .serializeValuesWith(SerializationPair.fromSerializer(
            new GenericJackson2JsonRedisSerializer()
        ));
}
```

Configurazione Redis Template

48.4 Interview Questions & Best Practices

Colloquio: Cache Consistency Problem

Domanda: Qual è il rischio maggiore nell'usare la Cache?

Risposta: L'inconsistenza dei dati (**Stale Data**). Se modifichi un dato sul DB (es. cambi il prezzo di un prodotto) ma dimentichi di chiamare `@CacheEvict`, la cache continuerà a servire il prezzo vecchio finché non scade il TTL. **Regola d'oro:** Ci sono solo due cose difficili in informatica: invalidare la cache e dare i nomi alle cose.

Colloquio: The Thundering Herd Problem

Domanda: Cosa succede se il TTL di un dato molto richiesto scade improvvisamente?

Risposta: Migliaia di richieste arrivano contemporaneamente. Trovano la cache vuota (Miss). Tutte e mille partono verso il Database simultaneamente per ricalcolare lo stesso dato. Il DB crolla per il picco di carico. **Soluzioni:**

- **Soft TTL:** Rinfrescare la cache in background prima che scada davvero.
- **Locking:** Solo il primo thread va al DB, gli altri aspettano.

48.5 Riepilogo Strategie

Tecnologia	Ruolo
Spring Cache	L'interfaccia. Usi `@Cacheable` per non sporcare il codice con logica di infrastruttura.
ConcurrentMap	Cache di default (dev). Veloce ma locale (non condivisa tra istanze).
H2	Database SQL in-memory. Utile per test, non è propriamente una cache layer.
Redis	Cache distribuita di produzione. Supporta TTL, eviction policies e condivisione dati tra microservizi.

Capitolo 49

Spring Security

49.1 Concetti Fondamentali

Spring Security è il framework standard *de-facto* per la sicurezza delle applicazioni Java. È potente, altamente personalizzabile, ma noto per avere una curva di apprendimento ripida. Non si limita al semplice login: gestisce la protezione contro attacchi comuni (CSRF, Session Fixation), la crittografia delle password e la gestione granulare degli accessi.

49.1.1 Autenticazione vs Autorizzazione

Questi due termini vengono spesso confusi, ma rappresentano due fasi distinte e sequenziali.

- **Authentication (Autenticazione) - "Chi sei?"**
  È il processo di verifica dell'identità di un utente. *Esempio:* Inserire username e password, fornire l'impronta digitale o inviare un token JWT. Se il sistema riconosce le credenziali, l'utente è *autenticato*.
- **Authorization (Autorizzazione) - "Cosa puoi fare?"**
  È il processo di verifica dei permessi di un utente già autenticato. *Esempio:* Un utente loggato può leggere i dati, ma solo un "Admin" può cancellarli. Se l'utente ha il ruolo giusto, è *autorizzato*.

49.1.2 Il Glossario Essenziale

Per capire il codice di Spring Security, bisogna conoscere i nomi degli oggetti chiave che circolano nel framework.

1. **Principal:** Rappresenta l'utente corrente (o il servizio/dispositivo). Solitamente è lo *username* o l'oggetto `User` completo.
2. **Credentials:** La prova dell'identità. Solitamente la *password* (prima di essere verificata) o il token. Vengono spesso cancellate dalla memoria subito dopo l'autenticazione per sicurezza.
3. **Authorities (GrantedAuthority):** I permessi assegnati all'utente. Possono essere ruoli ("ROLE_ADMIN") o permessi fini ("READ_PRIVILEGE").

Deep Dive: Il prefisso "ROLE_"

In Spring Security, c'è una differenza tecnica sottile tra **Role** e **Authority**. Per convenzione, i ruoli hanno il prefisso `ROLE_` (es. `ROLE_USER`). Quando nel codice usi `.hasRole("USER")`, Spring cerca automaticamente `ROLE_USER` nel database. Se usi `.hasAuthority("USER")`, cercherà esattamente la stringa "USER". È una trappola

comune in cui cadono molti junior.

49.1.3 Stateful vs Stateless

La scelta architetturale più importante quando si configura la sicurezza riguarda la gestione della sessione.

Colloquio: Autenticazione Stateful (Session) vs Stateless (Token)

Domanda: "Stiamo costruendo un backend per un'App Mobile e una Single Page Application (React). Che strategia di sicurezza usiamo?"

Risposta: Dobbiamo usare un approccio **Stateless** (tipicamente JWT).

- **Stateful (Classico/Monolite):** Il server crea un oggetto `HttpSession` in memoria e invia al client un cookie `JSESSIONID`.
  - *Pro:* Facile da implementare, invalidazione sessione immediata.
  - *Contro:* Non scala bene orizzontalmente (serve sticky session o session replication tra server), non ideale per App Mobile.
- **Stateless (REST/Microservizi):** Il server non salva nulla in memoria dopo la risposta. Invia un **Token** (es. JWT) firmato al client. Il client deve inviare il token ad ogni richiesta successiva.
  - *Pro:* Scalabilità infinita (qualsiasi server può validare il token), standard per Mobile/Frontend moderni.
  - *Contro:* Difficile revocare un token prima della scadenza (necessità di Blacklist o tempi brevi).

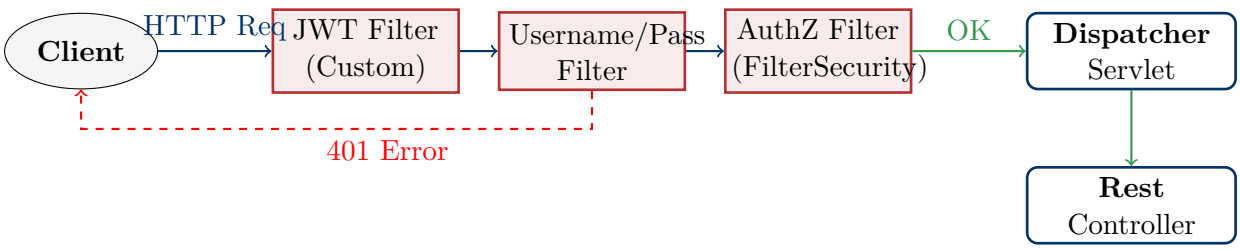
49.2 Architettura Interna (Dietro le quinte)

Per capire come risolvere i problemi di sicurezza (es. CORS errors, 403 Forbidden inaspettati), bisogna comprendere il flusso della richiesta HTTP. Spring Security non è magico: è basato interamente sullo standard dei **Servlet Filters**.

49.2.1 La Security Filter Chain

Prima che una richiesta HTTP raggiunga il tuo `DispatcherServlet` (e quindi i tuoi `@RestController`), deve attraversare una serie di filtri di sicurezza. Se uno di questi filtri rifiuta la richiesta (lancia un'eccezione o restituisce un errore 401/403), il tuo Controller non verrà mai eseguito.

Ecco una visualizzazione del flusso:



I componenti chiave sono:

1. **DelegatingFilterProxy:** È il ponte tra il container Servlet (Tomcat) e il contesto di Spring. Intercetta le richieste e le passa ai bean di Spring.
2. **FilterChainProxy:** Gestisce la lista dei filtri di sicurezza.
3. **SecurityFilterChain:** La catena effettiva dei filtri che corrispondono alla richiesta. L'ordine è vitale (es. controlli il token JWT *prima* di controllare se l'utente è un Admin).

49.2.2 SecurityContextHolder e ThreadLocal

Una volta che un utente è autenticato con successo (es. password corretta o token valido), dove vengono salvati i suoi dati? Spring Security usa il `SecurityContextHolder`.

Deep Dive: Deep Dive: ThreadLocal

Il `SecurityContextHolder`, di default, usa una strategia **ThreadLocal**.

Cosa significa? In un'applicazione Web Java, ogni richiesta HTTP viene gestita da un singolo thread dedicato (dal pool di thread di Tomcat). Il `ThreadLocal` è come una "variabile globale", ma visibile **solo** all'interno di quel specifico thread.

Vantaggio: Non devi passare l'oggetto `User` come parametro in ogni metodo del Service o del Repository. Puoi recuperarlo staticamente ovunque, perché Spring sa che quel thread appartiene a quell'utente.

Attenzione con @Async: Se lanci un metodo `@Async` (nuovo thread), il contesto di sicurezza viene perso (l'utente risulta null), a meno che non configuri la strategia `MODE_INHERITABLETHREADLOCAL`.

49.2.3 Come recuperare l'utente loggato

Grazie al meccanismo sopra descritto, ecco come accedere ai dati dell'utente corrente in qualsiasi punto del codice (Service, Utility, ecc.):

```
public String getCurrentUsername() {
    // 1. Accedi al contesto statico
    SecurityContext context = SecurityContextHolder.getContext();

    // 2. Recupera l'Authentication (se esiste)
    Authentication authentication = context.getAuthentication();

    // 3. Verifica se è autenticato e non è un utente anonimo
    if (authentication != null && authentication.isAuthenticated()
        && !(authentication instanceof AnonymousAuthenticationToken)) {

        // Il Principal è solitamente lo username o un oggetto UserDetails
        return authentication.getName();
    }

    return null;
}
```

Listing 49.1: Recuperare il Principal corrente

Colloquio: Filtri Custom: Dove posizionarli?

Domanda: "Ho creato un filtro per validare un Token JWT proprietario. Dove lo inserisco nella catena?"

Risposta: Va inserito **prima** del filtro di autenticazione username/password standard.

In Spring Boot 3 si usa il metodo `addFilterBefore`:

```
http.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);
```

Se lo mettessi dopo, Spring potrebbe aver già bloccato l'utente perché "non autenticato", rendendo inutile il tuo controllo del token.

49.3 Configurazione Moderna (Spring Boot 3)

La versione 5.7 di Spring Security (inclusa in Spring Boot 3) ha segnato un punto di svolta epocale, introducendo cambiamenti che rompono la retrocompatibilità. Se provi a copiare un tutorial del 2021, il codice non compilerà.

49.3.1 Addio WebSecurityConfigurerAdapter

Fino a poco tempo fa, la prassi era estendere la classe `WebSecurityConfigurerAdapter` e fare override del metodo `configure(HttpSecurity http)`. Questa classe è stata **rimossa**.

L'approccio moderno è **basato sui componenti (Component-based)**: invece di estendere una classe e sovrascrivere metodi, si definisce un **@Bean** di tipo `SecurityFilterChain`.

Colloquio: Perché hanno rimosso l'Adapter?

Domanda: "Perché Spring ha complicato le cose rimuovendo una classe che funzionava bene?"

Risposta: In realtà, l'hanno semplificata. L'ereditarietà (`extends`) creava un forte accoppiamento con il framework e nascondeva cosa succedeva "dietro le quinte" (es. quale `AuthenticationManager` veniva esposto). L'approccio a **Bean** è più trasparente, favorisce la Dependency Injection e permette di avere multiple catene di sicurezza (es. una per le API /api/** e una per la parte Web /admin/**) semplicemente definendo più Bean.

49.3.2 La sintassi Lambda DSL

Oltre al cambio di struttura, è cambiata la sintassi interna. Il vecchio stile "concatenato" con i metodi `.and()` è deprecato e sostituito dalle espressioni Lambda. Questo rende la configurazione più leggibile perché l'indentazione riflette la gerarchia delle impostazioni.

- **Vecchio Stile:** `http.csrf().disable().authorizeRequests()...`
- **Nuovo Stile:** `http.csrf(csrf -> csrf.disable())...`

49.3.3 Codice: Configurazione API REST Standard

Ecco il template "Gold Standard" per un backend REST in Spring Boot 3.

```
1 @Configuration
2 @EnableWebSecurity
3 public class SecurityConfig {
4
5     @Bean
6     public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
7         http
```

```
            // 1. Disabilita CSRF (non serve per API Stateless gestite da token)
            .csrf(csrf -> csrf.disable())

            // 2. Configura la sessione come STATELESS
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )

            // 3. Gestione Autorizzazioni (URL Matching)
            .authorizeHttpRequests(auth -> auth
                // Endpoint pubblici (Login, Registrazione)
                .requestMatchers("/api/auth/**").permitAll()
                // Endpoint Swagger/OpenAPI (opzionale)
                .requestMatchers("/v3/api-docs/**", "/swagger-ui/**").permitAll()
                // Endpoint Admin
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                // Tutto il resto richiede autenticazione
                .anyRequest().authenticated()
            );

            // Nota: Qui andrebbe aggiunto il filtro JWT (lo vedremo dopo)
            // .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.
    ↪ class);

        return http.build();
    }
}
```

Listing 49.2: SecurityFilterChain per API REST

Deep Dive: authorizeRequests vs authorizeHttpRequests

Un altro "gotcha" della migrazione:

- `authorizeRequests()`: È il vecchio metodo. Funziona ancora ma è deprecato.
- `authorizeHttpRequests()`: È il nuovo metodo introdotto in Spring Security 5.5+.

Differenza tecnica: Il nuovo metodo usa l'`AuthorizationManager` API (più efficiente e flessibile) invece del vecchio sistema basato su metadati e votatori. Quando scrivi nuovo codice, usa sempre `authorizeHttpRequests`.

49.3.4 Gestione CORS (Cross-Origin Resource Sharing)

Se il tuo Frontend (es. Angular su localhost:4200) chiama il Backend (localhost:8080), il browser bloccherà la richiesta se non configuri il CORS. In Spring Security, il CORS deve essere abilitato esplicitamente nella catena.

```
http.cors(cors -> cors.configurationSource(request -> {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(List.of("http://localhost:4200"));
    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE"));
    config.setAllowedHeaders(List.of("*"));
    return config;
}));
```

Listing 49.3: Abilitare CORS

49.4 Gestione Utenti e Password

Configurare i filtri serve a poco se il sistema non sa dove andare a pescare gli utenti per verificare le credenziali. Spring Security disaccoppia la logica di autenticazione dalla fonte dei dati tramite due interfacce chiave: `UserDetailsService` e `PasswordEncoder`.

49.4.1 UserDetailsService: Il ponte col Database

Questa interfaccia ha un unico metodo: `loadUserByUsername(String username)`. Il suo compito è accettare uno username (o email), cercare nel database (o LDAP, o memoria) e restituire un oggetto `UserDetails`.

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String email) throws
    ↪ UsernameNotFoundException {
        // 1. Cerca l'utente nel DB (Entity JPA)
        com.example.entity.User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new UsernameNotFoundException("Utente non trovato"));

        // 2. Converti l'Entity nel formato UserDetails di Spring
        return org.springframework.security.core.userdetails.User.builder()
            .username(user.getEmail())
            .password(user.getPassword()) // Password GIA' hashata nel DB
            .roles(user.getRole().name()) // Es. "ADMIN" -> diventa ROLE_ADMIN
            .build();
    }
}
```

Listing 49.4: Implementazione Custom UserDetailsService

49.4.2 Memorizzazione Password: Hashing vs Encryption

Questo è un argomento teorico imprescindibile. Mai salvare password in chiaro, e mai usare algoritmi reversibili.

Colloquio: Differenza tra Hashing e Cifratura (Encryption)

Domanda: "Per proteggere la password nel DB, la cifri o ne fai l'hash?"

Risposta: Ne faccio l'**Hash**.

- **Cifratura (Encryption):** È bidirezionale (Reversibile). Se ho la chiave segreta, posso decifrare il dato e ottenere la password originale. Questo è pericoloso: se un hacker ruba la chiave e il DB, ha tutte le password.
- **Hashing (es. BCrypt, SHA-256):** È monodirezionale (Irreversibile). È matematicamente impossibile risalire alla password originale partendo dall'hash. Per verificare il login, si fa l'hash dell'input utente e lo si confronta con quello nel DB.

49.4.3 PasswordEncoder e BCrypt

Spring Security raccomanda l'uso di `BCryptPasswordEncoder`.

Deep Dive: Perché BCrypt e cos'è il "Salting"?

Se usi MD5 o SHA-256 semplice, due utenti con la password "ciro123" avranno lo stesso hash nel DB. Questo li rende vulnerabili alle *Rainbow Tables* (tabelle precalcolate di hash).

BCrypt risolve il problema usando il **Salting** automatico:

1. Genera una stringa casuale (il "salt").
2. Unisce il salt alla password e fa l'hash.
3. Il risultato finale contiene sia l'hash che il salt in chiaro.

Risultato: Ogni volta che fai l'hash di "ciro123", ottieni una stringa diversa! `$2a$10$...stringa_diversa_ogni_volta...` Spring Security sa come estrarre il salt dalla stringa salvata per verificare la password durante il login.

49.4.4 Configurazione Finale

Per attivare tutto questo, dobbiamo definire i Bean nella classe di configurazione.

```
@Configuration
public class SecurityConfig {

    // 1. Definiamo l'algoritmo di hashing
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    // 2. Configuriamo l'AuthenticationProvider (Collega Service e Encoder)
    @Bean
    public AuthenticationProvider authenticationProvider(UserDetailsService
    ↪ userDetailsService) {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }

    // 3. Esponiamo l'AuthenticationManager (utile per il login JWT)
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
    ↪ config)
            throws Exception {
        return config.getAuthenticationManager();
    }
}
```

Listing 49.5: Bean AuthProvider e PasswordEncoder

49.5 Sicurezza per API REST (JWT)

Nelle architetture moderne a microservizi o Single Page Application (SPA), l'uso delle sessioni server-side (Cookie JSESSIONID) è sconsigliato. Si preferisce un approccio basato su **Token**.

Lo standard di mercato è il **JSON Web Token (JWT)**.

49.5.1 Anatomia di un JWT

Un token JWT è una stringa composta da tre parti separate da un punto (`.`), codificate in Base64Url: `aaaaaa.bbbbbb.cccccc`

1. **Header:** Contiene il tipo di token (JWT) e l'algoritmo di firma usato (es. HS256).
2. **Payload (Claims):** Contiene i dati utili (es. user ID, email, data di scadenza, ruoli).
3. **Signature:** È la firma digitale creata combinando Header, Payload e una **Chiave Segreta** posseduta solo dal server. Garantisce che il token non sia stato manomesso.

Colloquio: Posso mettere la password nel JWT?

Domanda: "Il payload del JWT è cifrato? Posso metterci dati sensibili come la password?"

Risposta: ASSOLUTAMENTE NO. Il payload è solo **codificato** in Base64, non cifrato. Chiunque intercetti il token può decodificarlo e leggere il contenuto (es. usando il sito *jwt.io*). La sicurezza del JWT sta nella **Firma**: se un hacker modifica il payload (es. cambia ruolo da "USER" a "ADMIN"), la firma non corrisponderà più e il server rifiuterà il token.

49.5.2 Il Flusso di Autenticazione (Stateless)

1. Il client invia una POST `/login` con username e password.
2. Il server verifica le credenziali. Se corrette, **genera** un JWT firmato con la sua chiave privata.
3. Il server restituisce il JWT al client.
4. Per ogni richiesta successiva, il client invia il JWT nell'header HTTP: `Authorization: Bearer <token>`
5. Un filtro custom nel server intercetta la richiesta, valida la firma del token e autentica l'utente nel contesto.

49.5.3 Il Filtro Custom: JwtAuthenticationFilter

Questo è il componente critico che va scritto manualmente. Estende `OncePerRequestFilter` per garantire un'esecuzione unica per richiesta.

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtService jwtService; // Classe utility per parsing token
    private final UserDetailsService userDetailsService;

    // Costruttore omesso per brevità (Lombok @RequiredArgsConstructor)

    @Override
    protected void doFilterInternal(
            @NonNull HttpServletRequest request,
            @NonNull HttpServletResponse response,
            @NonNull FilterChain filterChain
    ) throws ServletException, IOException {

        final String authHeader = request.getHeader("Authorization");
```

```
        final String jwt;
        final String userEmail;

        // 1. Controlla se l'header esiste e inizia con "Bearer "
        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        // 2. Estrai il token
        jwt = authHeader.substring(7);
        userEmail = jwtService.extractUsername(jwt); // Metodo utility

        // 3. Se abbiamo l'email e l'utente NON è ancora autenticato nel contesto
        if (userEmail != null && SecurityContextHolder.getContext().getAuthentication
    ↪ () == null) {

            // Carica i dettagli utente dal DB
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(
    ↪ userEmail);

            // 4. Valida il token (firma e scadenza)
            if (jwtService.isTokenValid(jwt, userDetails)) {

                // 5. Crea l'oggetto Authentication
                UsernamePasswordAuthenticationToken authToken = new
    ↪ UsernamePasswordAuthenticationToken(
                        userDetails,
                        null, // Non servono credenziali
                        userDetails.getAuthorities()
                );

                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails
    ↪ (request));

                // 6. INSERISCI L'UTENTE NEL CONTESTO DI SICUREZZA
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }

        // 7. Passa la palla al prossimo filtro
        filterChain.doFilter(request, response);
    }
}
```

Listing 49.6: JwtAuthenticationFilter

Deep Dive: Perché controllare se il Context è null?

La riga `SecurityContextHolder.getContext().getAuthentication() == null` è fondamentale per le performance. Se l'utente è già stato autenticato in questa richiesta (caso raro, ma possibile in catene complesse o forward), non vogliamo rieseguire query al database o calcoli crittografici pesanti.

49.5.4 Access Token vs Refresh Token

Nei sistemi enterprise, i token non durano per sempre.

- **Access Token:** Dura poco (es. 15 minuti). Se viene rubato, il danno è limitato nel tempo.
- **Refresh Token:** Dura molto (es. 7 giorni). Serve per ottenere un nuovo Access Token senza chiedere di nuovo la password all'utente. Viene conservato in modo più sicuro (es. Cookie HttpOnly) e può essere revocato dal database (Blacklisting).

49.6 Gestione delle Autorizzazioni (RBAC)

Una volta che l'utente è autenticato (sappiamo chi è) e abbiamo caricato i suoi ruoli (UserDetailsService), dobbiamo decidere cosa può fare. Spring Security offre due livelli di protezione:

1. **Request Level (URL):** Definita nella `SecurityFilterChain` (es. `.requestMatchers("/admin/**")`). È la prima barriera.
2. **Method Level (AOP):** Definita tramite annotazioni direttamente sui metodi dei Service o dei Controller. È più granulare e preferibile in architetture complesse.

49.6.1 Abilitare la Method Security

In Spring Boot 3 (Security 6), la vecchia annotazione `@EnableGlobalMethodSecurity` è deprecata. La nuova annotazione da usare sulla classe di configurazione è `@EnableMethodSecurity`.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true) // Abilita @PreAuthorize e @PostAuthorize
public class SecurityConfig {
    // ... configurazione dei bean
}
```

Listing 49.7: Abilitare le annotazioni di sicurezza

49.6.2 Le Annotazioni Principali

@PreAuthorize

È l'annotazione più potente e usata. Supporta il linguaggio **SpEL** (Spring Expression Language), permettendo di scrivere logica condizionale complessa. Viene valutata *prima* dell'esecuzione del metodo.

```
@Service
public class DocumentService {

    // Caso 1: Controllo di Ruolo semplice
    @PreAuthorize("hasRole('ADMIN')")
    public void deleteDocument(Long id) {
        // ...
    }

    // Caso 2: Logica complessa (AND/OR)
    @PreAuthorize("hasRole('MANAGER') or hasRole('ADMIN')")
    public void approveDocument(Long id) {
        // ...
```

```
    }

    // Caso 3: Controllo di Ownership (Il dato appartiene all'utente?)
    // #username è il nome del parametro del metodo
    // authentication.name è l'utente loggato nel SecurityContext
    @PreAuthorize("#username == authentication.name")
    public void changePassword(String username, String newPassword) {
        // ...
    }
}
```

Listing 49.8: Esempi di @PreAuthorize

Deep Dive: @Secured vs @PreAuthorize

Spesso si trova codice legacy con `@Secured("ROLE_ADMIN")`.

Differenza: `@Secured` è un'annotazione vecchia (stile Java 5), non supporta SpEL (niente `OR`, `AND`, niente accesso ai parametri). **Consiglio:** Usa sempre `@PreAuthorize`. È lo standard moderno.

49.6.3 Post Authorization

A volte non basta controllare prima. Immagina un metodo che cerca un documento nel DB: non sai di chi è il documento finché non lo carichi. `@PostAuthorize` permette di eseguire il metodo, e poi decidere se l'utente può vedere il risultato o se lanciare un'eccezione (AccessDeniedException).

```
// returnObject è l'oggetto restituito dal metodo
@PostAuthorize("returnObject.owner == authentication.name")
public Document getDocument(Long id) {
    return documentRepository.findById(id);
}
```

Listing 49.9: Esempio @PostAuthorize

49.6.4 La trappola del prefisso "ROLE_"

Torniamo su questo punto perché è la causa numero 1 di bug "Access Denied" inspiegabili.

Colloquio: hasRole() vs hasAuthority()

Domanda: "Nel DB ho salvato il ruolo `ADMIN`. Nel codice uso `.hasRole('ADMIN')`, ma non funziona. Perché?"

Risposta: Spring Security, quando usi `hasRole('XYZ')`, aggiunge automaticamente il prefisso `ROLE_` e cerca `ROLE_XYZ`.

Hai due soluzioni:

1. **Opzione A (Consigliata):** Salva nel DB direttamente `ROLE_ADMIN`.
2. **Opzione B:** Usa `hasAuthority('ADMIN')` invece di `hasRole`. `hasAuthority` non aggiunge nessun prefisso, cerca la stringa esatta.

49.7 Protezione dagli Attacchi Comuni

Spring Security non serve solo a gestire i login, ma protegge l'applicazione da vulnerabilità note del protocollo HTTP. Analizziamo le due più frequenti: CSRF (che spesso disabilitiamo) e CORS (che dobbiamo configurare con cura).

49.7.1 CSRF (Cross-Site Request Forgery)

Il CSRF è un attacco che forza un utente finale (già autenticato) a eseguire azioni indesiderate su un'applicazione web in cui è attualmente loggato.

Esempio: Sei loggato sul sito della tua banca. Apri una mail malevola che contiene un link nascosto che fa una richiesta `POST /bonifico?to=hacker`. Se il sito usa i **Cookie** per l'autenticazione, il browser allegherà automaticamente il cookie di sessione e la banca eseguirà il bonifico.

Colloquio: Perché disabilitiamo CSRF nelle API REST?

Domanda: "Vedo sempre `.csrf(csrf -> csrf.disable())`. Non è pericoloso?"

Risposta: Dipende dal meccanismo di autenticazione.

- **Se usi Sessioni/Cookie:** Il CSRF protection deve essere **ABILITATO**. Spring Security si aspetta un token CSRF extra in ogni form HTML per validare la richiesta.
- **Se usi JWT (Stateless):** Il CSRF protection può essere **DISABILITATO**.

Motivo: I browser allegano automaticamente i Cookie, ma **non** allegano automaticamente gli Header custom (es. `Authorization: Bearer ...`). Poiché il JWT viaggia nell'header, un sito attaccante non può forzare il browser a inviare il token.

49.7.2 CORS (Cross-Origin Resource Sharing)

Il CORS non è un attacco, ma un meccanismo di sicurezza dei browser. Per default, la *Same-Origin Policy* impedisce a uno script caricato da `http://domain-a.com` di fare chiamate API verso `http://domain-b.com`.

Nello sviluppo moderno (Frontend Angular/React su localhost:4200, Backend Spring su localhost:8080), le origini sono diverse (cambia la porta). Il browser blocca tutto se il server non acconsente esplicitamente.

Deep Dive: La richiesta Preflight (OPTIONS)

Prima di fare una richiesta "complessa" (es. una POST con JSON e header Authorization), il browser invia automaticamente una richiesta preliminare di tipo **OPTIONS**.

Questa richiesta chiede: "Ehi server, accetti chiamate da localhost:4200 con questi header?". Se Spring Security blocca la richiesta OPTIONS (perché non autenticata), il Frontend riceverà un errore CORS. **Soluzione:** Configurare il CORS a livello di Spring Security, non solo nel Controller.

Configurazione CORS Globale

Non usare l'annotazione `@CrossOrigin` sui singoli controller (diventa ingestibile). Definisci una configurazione centralizzata.

```
@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
```

```
4
5       // 1. Chi può chiamarmi? (In prod mettere domini specifici, non *)
6       configuration.setAllowedOrigins(Arrays.asList("http://localhost:4200", "https://
        ↪ mia-app.com"));
7
8       // 2. Quali metodi accetto?
9       configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "
        ↪ OPTIONS"));
10
11      // 3. Quali header accetto? (Authorization è fondamentale per JWT)
12      configuration.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"));
13
14      // 4. Registra la configurazione per tutti gli endpoint
15      UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
16      source.registerCorsConfiguration("/**", configuration);
17      return source;
18  }
```

Listing 49.10: Configurazione CORS Best Practice

Capitolo 50

Spring Batch

50.1 Introduzione

Spring Batch è un framework leggero e completo, progettato per consentire lo sviluppo di applicazioni *batch* robuste. Un'elaborazione batch è, per definizione, l'esecuzione di una serie di job (lavori) senza l'interazione manuale dell'utente, spesso gestendo grandi moli di dati in maniera sequenziale o parallela.

Nel mondo Enterprise Java, è lo standard *de-facto*. Si basa sui principi di Spring Framework (Dependency Injection, POJO) per permettere agli sviluppatori di concentrarsi sulla logica di business (es. calcolo stipendi) delegando al framework la gestione della complessità infrastrutturale (transazioni, gestione risorse, logging).

50.1.1 Caratteristiche Fondamentali

Non si tratta solo di eseguire un ciclo `for` su una lista di dati. Spring Batch offre funzionalità avanzate pronte all'uso:

- **Gestione delle Transazioni:** Automatica e granulare (per "chunk").
- **Restartability:** Capacità di riavviare un job fallito esattamente dal punto in cui si è interrotto.
- **Skip & Retry:** Logica configurabile per saltare record corrotti o riprovare operazioni fallite (es. timeout di rete).
- **Statistiche e Monitoraggio:** Tracciamento automatico di tempi di esecuzione, record letti, scritti e ignorati tramite metadati persistenti su DB.
- **Scalabilità:** Supporto per esecuzione multithread e partizionamento dei dati.

Colloquio: Spring Batch vs @Scheduled (Spring Task)

Questa è una classica domanda per capire se il candidato ha esperienza reale.

Domanda: "Perché dovrei usare Spring Batch se posso semplicemente annotare un metodo con `@Scheduled` e fare un ciclo su tutti i record?"

Risposta:

1. **Gestione della Memoria (OOM):** Un metodo `@Scheduled` che carica 1 milione di record in una `List` causerà un *Out Of Memory*. Spring Batch legge a "pezzi" (chunk), mantenendo la memoria costante.
2. **Transazionalità:** In un metodo semplice, se il record 99.999 fallisce, di solito si fa rollback di tutto (persi 99.998 record elaborati) o commit parziale ingestibile. Spring Batch committa ogni N record.

3. **Ripresa (Restartability):** Se lo script si blocca a metà notte, con `@Scheduled` non sai dove sei arrivato. Spring Batch salva lo stato nel DB: al riavvio riprende esattamente da dove si era fermato.

50.1.2 Architettura a Livelli

L'architettura di Spring Batch è strutturata in tre livelli distinti per separare le responsabilità:

1. **Application Layer:** Contiene i job personalizzati e il codice scritto dallo sviluppatore (Business Logic).
2. **Batch Core:** Le classi API principali per lanciare e controllare i job (`JobLauncher`, `Job`, `Step`).
3. **Batch Infrastructure:** I servizi comuni di basso livello (lettori e scrittori di file/DB, gestione dei retry, template).

Deep Dive: Batch Processing vs Real-Time

È importante distinguere i due paradigmi:

- **Real-Time (Online):** Elabora una richiesta alla volta (es. API REST). La risposta deve essere immediata (millisecondi). Se c'è un errore, l'utente viene notificato subito.
- **Batch (Offline):** Elabora milioni di record in background. La durata può essere di ore. Se c'è un errore, il sistema deve gestirlo (skip/retry) o notificare un amministratore, ma il processo non ha un utente "in attesa" davanti allo schermo.

50.1.3 Casi d'uso tipici

Spring Batch è ideale in scenari come:

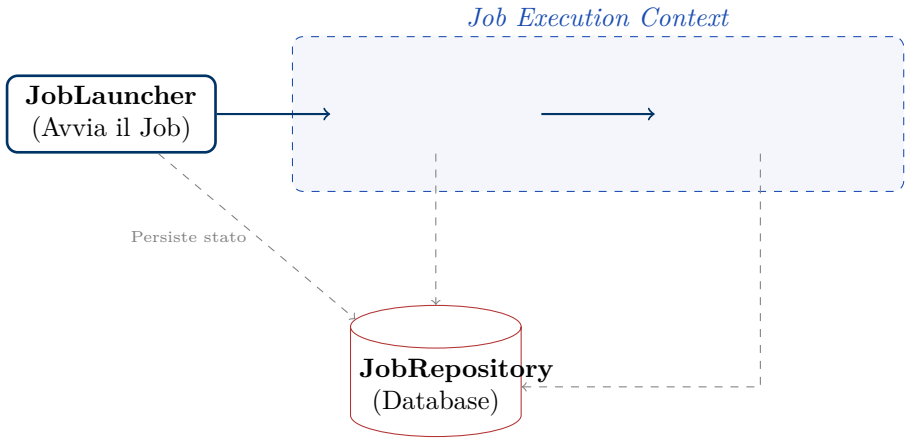
- **ETL (Extract, Transform, Load):** Spostare dati da un vecchio DB a uno nuovo, o caricare dati da file CSV/XML ricevuti da fornitori esterni.
- **Reportistica Finanziaria:** Calcolo di interessi, fatturazione mensile o riconciliazione bancaria a fine giornata.
- **Massive Emailing:** Invio di newsletter o notifiche a milioni di utenti (senza bloccare il server di posta).

50.2 Architettura e Componenti Fondamentali

Per padroneggiare Spring Batch, bisogna comprendere come i suoi componenti interagiscono. Non basta scrivere codice Java; bisogna capire il flusso di controllo.

50.2.1 Panoramica dei Componenti

Il flusso di esecuzione segue una gerarchia precisa. Ecco lo schema architetturale ad alto livello:



1. **JobLauncher:** È l'interfaccia semplice che permette di avviare un Job. Prende in input il `Job` e i `JobParameters`.
2. **Job:** Rappresenta l'intero processo batch. È un contenitore di uno o più Step.
3. **Step:** È l'unità di lavoro. Un Job è composto da una sequenza di Step. Uno Step può essere di due tipi:
   - **Chunk-Oriented:** Legge, elabora e scrive dati (il classico ETL).
   - **Tasklet:** Esegue un singolo compito semplice (es. cancellare un file, eseguire una query di pulizia, inviare una mail di fine lavoro).
4. **JobRepository:** Il meccanismo di persistenza. Si occupa di salvare lo stato attuale del Job e degli Step nel database.

50.2.2 JobInstance vs JobExecution

Questa è una distinzione terminologica che crea confusione ma è essenziale per comprendere il meccanismo di *Restart*.

Colloquio: Differenza tra JobInstance e JobExecution

Domanda: "Se lancio un Job oggi che fallisce, e lo rilancio domani, creo una nuova istanza?"

Risposta: Dipende dai **JobParameters**.

- **JobInstance:** È il concetto logico di "Esecuzione del Job per quei specifici parametri". Esempio: `EndDayJob` per la data `2023-10-25`.
- **JobExecution:** È il tentativo tecnico di eseguire quell'istanza.

Esempio Pratico:

1. Lancio il Job con parametro `date=2023-10-25`. Viene creata la **JobInstance A** e la **JobExecution 1**. **Fallisce**.
2. Correggo il bug e rilancio con `date=2023-10-25`. Spring Batch riconosce che l'Istanza A esiste già. Crea la **JobExecution 2** collegata alla **JobInstance A**. Riprende da dove si era fermato.
3. Lancio il Job con `date=2023-10-26`. Nuovi parametri → Nuova **JobInstance B** → Nuova **JobExecution 3**.

50.2.3 Il Metamodello (Tabelle di Spring Batch)

Spring Batch non funziona "in memoria" (di default). Ha bisogno di un set di tabelle dedicate sul database per tracciare lo stato. Senza queste tabelle, le funzionalità di riavvio e monitoraggio non funzionano.

Deep Dive: Le Tabelle BATCH_*

Durante un colloquio per posizioni Senior, sapere dove guardare quando un batch si blocca è vitale.

- `BATCH_JOB_INSTANCE`: Contiene la chiave univoca del job (Nome Job + Hash dei Parametri).
- `BATCH_JOB_EXECUTION`: Contiene lo stato (COMPLETED, FAILED, STARTED), data inizio/fine.
- `BATCH_JOB_EXECUTION_CONTEXT`: Memorizza dati serializzati da passare tra gli step o da recuperare al riavvio (es. "riga corrente: 450").
- `BATCH_STEP_EXECUTION`: Dettagli su ogni step (record letti, scritti, commit effettuati, rollback).

Nota: In Spring Boot, queste tabelle possono essere create automaticamente all'avvio impostando `spring.batch.jdbc.initialize-schema=always`, ma in produzione si usa solitamente uno script DDL manuale fornito da Spring.

50.2.4 Tasklet vs Chunk: Quando usare cosa?

- Usa un **Tasklet** quando l'operazione è "atomica" e non iterativa.

```
@Bean
public Step cleanUpStep(JobRepository jobRepository, PlatformTransactionManager
    ↪ txManager) {
    return new StepBuilder("cleanUpStep", jobRepository)
        .tasklet((contribution, chunkContext) -> {
            System.out.println("Pulizia file temporanei...");
            return RepeatStatus.FINISHED;
        }, txManager)
        .build();
}

```

Esempio Tasklet Semplice

- Usa un **Chunk** quando devi processare una lista di item. Spring Batch gestirà per te l'apertura e chiusura delle transazioni in modo efficiente.

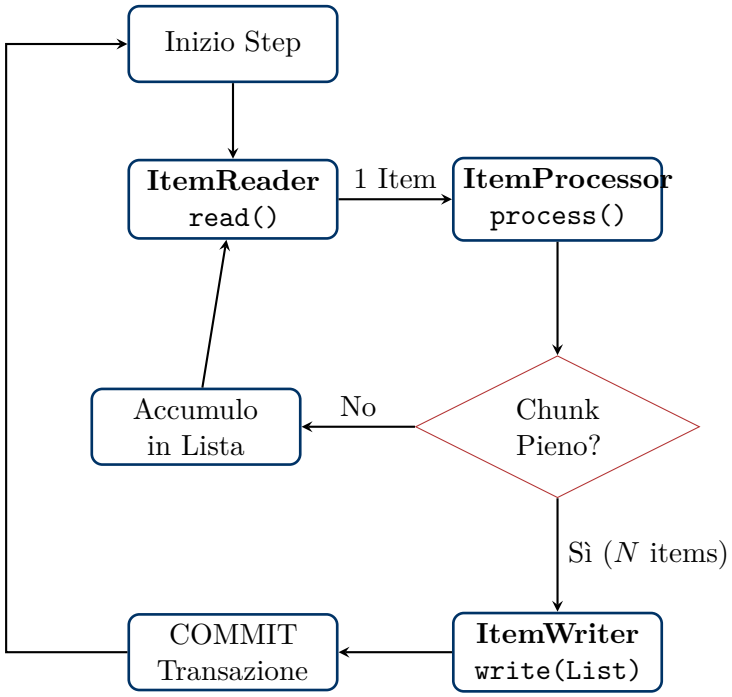
50.3 Chunk-Oriented Processing

Il "Chunk-Oriented Processing" è il pattern architetturale distintivo di Spring Batch. A differenza della semplice iterazione, questo approccio ottimizza le performance raggruppando le operazioni di scrittura.

L'idea base è: leggere un dato alla volta, elaborarlo, e accumulare il risultato in memoria. Quando il numero di elementi accumulati raggiunge una soglia predefinita (il **Commit Interval**), l'intero blocco viene scritto in un colpo solo e la transazione viene committata.

50.3.1 Il Ciclo di Vita del Chunk

Ecco come avviene il flusso di dati all'interno di uno Step configurato a chunk:



50.3.2 Le Interfacce Chiave

È fondamentale conoscere la firma dei metodi delle tre interfacce principali.

- **ItemReader<T>:** Restituisce un oggetto alla volta. Restituisce `null` per indicare che i dati sono finiti.
- **ItemProcessor<I, O>:** Riceve un oggetto di input `I`, lo trasforma e restituisce `O`.
  - **Nota Importante:** Se restituisce `null`, l'elemento viene **filtrato** (scartato) e non arriverà al Writer.
- **ItemWriter<T>:** Riceve una `Chunk<? extends T>` (o una `List` nelle versioni precedenti), non un singolo oggetto.

```
// 1. READER
public interface ItemReader<T> {
    T read() throws Exception; // Ritorna null alla fine
}

// 2. PROCESSOR
public interface ItemProcessor<I, O> {
    O process(I item) throws Exception; // Ritorna null per scartare
}

// 3. WRITER
public interface ItemWriter<T> {
    // Nota: riceve una LISTA, non un singolo oggetto
    void write(Chunk<? extends T> chunk) throws Exception;
}
```

Listing 50.1: Le interfacce funzionali

50.3.3 Gestione delle Transazioni

La potenza di Spring Batch risiede nel confine transazionale.

Deep Dive: Commit Interval e Rollback

In uno scenario con `chunk-size = 10`:

1. Spring apre una transazione DB.
2. Legge 10 record, li processa 10 volte.
3. Passa una lista di 10 elementi al Writer.
4. Se tutto va bene → **COMMIT** (i dati sono persistiti).
5. Se avviene un'eccezione al record 9 (durante la lettura o processazione) o durante la scrittura:
   - L'intera transazione viene annullata (**ROLLBACK**).
   - Nessuno dei 10 record viene salvato (nemmeno i primi 8 validi).

Questo garantisce l'integrità dei dati: o tutto il blocco è valido, o nulla viene scritto.

Colloquio: Come scelgo la dimensione del Chunk?

Domanda: "Perché non impostare il chunk size a 1 o a 1.000.000?"

Risposta: È un trade-off tra overhead transazionale e memoria.

- **Chunk = 1:** Troppo lento. Per 10.000 record fai 10.000 commit al database. L'overhead della gestione della transazione uccide le performance.
- **Chunk = 1.000.000:** Rischio `OutOfMemoryError` (tieni troppi oggetti in RAM prima di scrivere) e rischio di *Transaction Log* del DB pieno. Inoltre, se fallisce l'ultimo record, devi riprocessarne un milione (spreco di tempo).
- **Valori tipici:** Tra 50 e 1000, a seconda della grandezza degli oggetti e della latenza del DB.

50.4 Implementazione dei Componenti

Spring Batch fornisce molte implementazioni pronte all'uso ("Out of the box") per leggere e scrivere dalle fonti dati più disparate (File piatti, XML, JSON, Database JDBC/JPA, MongoDB, Kafka, ecc.). Analizziamo le più frequenti in ambito Enterprise.

50.4.1 ItemReader: Lettura da File (CSV)

Il `FlatFileItemReader` è il componente standard per leggere file di testo. È altamente configurabile per gestire delimitatori, intestazioni e righe di commento.

```
@Bean
public FlatFileItemReader<Person> reader() {
    return new FlatFileItemReaderBuilder<Person>()
        .name("personItemReader")
        .resource(new ClassPathResource("dati_input.csv"))
        .delimited() // Indica che è un file delimitato (default virgola)
        .names("firstName", "lastName") // Mappa le colonne ai campi del POJO
        .targetType(Person.class) // La classe di destinazione
        .linesToSkip(1) // Salta l'header
        .build();
}
```

Listing 50.2: Configurazione FlatFileItemReader

50.4.2 ItemReader: Lettura da Database (JDBC)

Quando si legge da database, ci sono due approcci filosofici diversi. Capire la differenza è vitale per le performance e la concorrenza.

1. Cursor Based (JdbcCursorItemReader)

Funziona come una `ResultSet` standard di JDBC. Apre una connessione, esegue la query e mantiene il cursore aperto mentre scorre i risultati uno a uno.

- **Pro:** Molto performante per stream di dati sequenziali.
- **Contro:** Mantiene la connessione al DB occupata per tutta la durata dello step. **Non è Thread-Safe** (non usabile in step multithread).

2. Paging Based (JdbcPagingItemReader)

Invece di una query unica, esegue molteplici query per recuperare "pagine" di dati (es. `LIMIT 1000 OFFSET 0`, poi `OFFSET 1000`...).

- **Pro:** Non tiene la connessione bloccata a lungo. **È Thread-Safe** (ogni thread può leggere una pagina diversa).
- **Contro:** Richiede una chiave di ordinamento (Sort Key) stabile per garantire che le pagine non si sovrappongano.

Deep Dive: Deep Dive: Cursor vs Paging in Multithreading

Se in un colloquio ti chiedono: *"Come velocizzi un job che legge da DB?"*, la risposta "Attivo il multithreading" è parziale.

Devi specificare: "Attivo il multithreading e cambio il reader da `Cursor` a `Paging`". Se usi un `JdbcCursorItemReader` in un ambiente multithread, avrai eccezioni di concorrenza perché più thread proveranno a spostare lo stesso cursore JDBC contemporaneamente. Il `JdbcPagingItemReader` è progettato proprio per risolvere questo problema.

```
@Bean
public JdbcPagingItemReader<Person> pagingReader(DataSource dataSource) {
    Map<String, Order> sortKeys = new HashMap<>();
    sortKeys.put("id", Order.ASCENDING); // Fondamentale per la paginazione

    return new JdbcPagingItemReaderBuilder<Person>()
        .name("dbReader")
        .dataSource(dataSource)
        .selectClause("SELECT id, first_name, last_name")
        .fromClause("FROM person")
        .sortKeys(sortKeys)
        .pageSize(1000) // Dimensione della pagina (non del chunk!)
        .rowMapper(new BeanPropertyRowMapper<>(Person.class))
        .build();
}
```

Listing 50.3: Configurazione JdbcPagingItemReader

50.4.3 ItemProcessor: Logica di Business

Il Processor è il luogo dove risiede la logica. È opzionale (puoi passare direttamente da Reader a Writer), ma consigliato per trasformazioni o validazioni.

```java
public class PersonProcessor implements ItemProcessor<Person, Person> {

    @Override
    public Person process(Person item) throws Exception {
        // Logica di trasformazione
        String upperName = item.getFirstName().toUpperCase();
        item.setFirstName(upperName);

        // Logica di FILTRAGGIO
        // Se il cognome è "Test", scarta il record
        if ("Test".equalsIgnoreCase(item.getLastName())) {
            return null; // Restituire null significa "SKIP"
        }

        return item;
    }
}
```

Listing 50.4: Processor con filtraggio

50.4.4 ItemWriter: Scrittura su DB

Il `JdbcBatchItemWriter` è molto efficiente perché usa il `batchUpdate` di JDBC (invia un blocco di comandi SQL in una sola chiamata di rete).

```java
@Bean
public JdbcBatchItemWriter<Person> writer(DataSource dataSource) {
    return new JdbcBatchItemWriterBuilder<Person>()
        .itemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSourceProvider
    ↪ <>())
        .sql("INSERT INTO people (first_name, last_name) VALUES (:firstName, :lastName
    ↪ )")
        .dataSource(dataSource)
        .build();
}
```

Listing 50.5: Writer JDBC

50.5 Gestione degli Errori e Resilienza

In un ambiente di produzione reale, i job batch non lavorano mai in condizioni ideali. File corrotti, record duplicati, timeout di rete o database momentaneamente irraggiungibili sono eventi comuni. Spring Batch offre meccanismi robusti per gestire questi scenari senza far fallire l'intero job.

50.5.1 Skip Logic (Tolleranza ai guasti)

La logica di **Skip** permette di ignorare specifici record che causano eccezioni, continuando l'elaborazione dei successivi. È utile per errori *deterministici* legati al dato (es. un CSV con una riga malformata o una stringa al posto di un numero).

Configuriamo lo Step per tollerare l'eccezione ma fissiamo un limite:

- **skip(Class):** Specifica quale eccezione ignorare.

- **skipLimit(int):** Specifica quanti errori tollerare prima di far fallire il Job.

Deep Dive: Deep Dive: Come funziona lo Skip (Scan Mode)

Questo è un dettaglio architetturale avanzato. Quando avviene un'eccezione in un Chunk durante la scrittura:

1. Spring Batch esegue il **Rollback** dell'intera transazione (tutto il chunk viene annullato).
2. Entra in una modalità chiamata "Scan Mode" (o item-by-item processing).
3. Ripassa gli item del chunk **uno alla volta**, committando transazioni singole per ogni item.
4. Quando incontra l'item che genera l'errore, lo "skippa" (segnalandolo nei log/tabelle) e prosegue col successivo.

Conclusione: Lo skip ha un costo prestazionale elevato (moltiplicazione delle transazioni). Va usato per eccezioni rare, non come logica di flusso normale.

50.5.2 Retry Logic (Riprovare l'operazione)

La logica di **Retry** serve per errori *transienti* (temporanei), come un timeout di connessione o un deadlock sul DB. Non ha senso scartare il dato se il problema è la rete; bisogna riprovare.

Colloquio: Skip vs Retry: Quando usare quale?

Domanda: "Ho un'eccezione `DataIntegrityViolationException` e una `SocketTimeoutException`. Come le gestisco?"

Risposta:

- **DataIntegrityViolationException** (es. Vincolo Unique violato): È un errore permanente. Riprovare 100 volte darà sempre errore. → Uso lo **SKIP**.
- **SocketTimeoutException**: È un errore temporaneo. Tra 1 secondo la rete potrebbe tornare. → Uso il **RETRY**.

50.5.3 Esempio di Configurazione Fault-Tolerant

Ecco come configurare uno Step resiliente che tollera file CSV sporchi e brevi down di rete.

```
@Bean
public Step resilientStep(JobRepository jobRepository,
                          PlatformTransactionManager txManager,
                          ItemReader<User> reader,
                          ItemWriter<User> writer) {

    return new StepBuilder("resilientStep", jobRepository)
        .<User, User>chunk(10, txManager)
        .reader(reader)
        .writer(writer)
        .faultTolerant() // Abilita le funzionalità di resilienza

        // --- SKIP CONFIG ---
        .skip(FlatFileParseException.class) // Errore di parsing file
        .skip(ValidationException.class)    // Errore logico
        .noSkip(NullPointerException.class) // Bug del codice: non skippare, deve
    ↪ fallire!
        .skipLimit(10) // Max 10 record ignorati
```

```
18
19          // --- RETRY CONFIG ---
20          .retry(ConnectTimeoutException.class) // Errore di rete
21          .retry(DeadlockLoserDataAccessException.class) // Deadlock DB
22          .retryLimit(3) // Riprova max 3 volte per ogni item
23
24          .build();
25  }
```

Listing 50.6: Configurazione Skip e Retry

50.5.4 Listeners (Intercettare gli Eventi)

I **Listeners** permettono di eseguire logica custom (logging, notifiche email, pulizia risorse) in momenti specifici del ciclo di vita. Interfacce comuni: `JobExecutionListener`, `StepExecutionListener`, `ItemReadListener`, `ItemWriteListener`.

```
1  @Component
2  public class JobNotificationListener implements JobExecutionListener {
3
4      private static final Logger log = LoggerFactory.getLogger(JobNotificationListener.
       ↪ class);
5
6      @Override
7      public void afterJob(JobExecution jobExecution) {
8          if (jobExecution.getStatus() == BatchStatus.COMPLETED) {
9              log.info("!!! JOB COMPLETATO CON SUCCESSO !!!");
10             // Qui potrei inviare una mail all'amministratore
11         } else if (jobExecution.getStatus() == BatchStatus.FAILED) {
12             log.error("!!! JOB FALLITO con eccezioni: " + jobExecution.
       ↪ getAllFailureExceptions());
13         }
14     }
15 }
```

Listing 50.7: Esempio JobExecutionListener

50.6 Scalabilità e Performance

Di default, Spring Batch esegue il Job in un **singolo thread**. Questo garantisce la massima sicurezza e semplicità (niente race conditions), ma su moli di dati enormi (milioni di record) potrebbe non essere sufficiente per rispettare le finestre temporali di esecuzione (SLA).

Esistono due strategie principali per scalare: il *Multi-threaded Step* (Vertical Scaling) e il *Partitioning* (Horizontal Scaling).

50.6.1 Multi-threaded Step

È il metodo più veloce da implementare. Si configura lo Step per utilizzare un `TaskExecutor` (pool di thread). In questo modo, più chunk vengono letti, processati e scritti in parallelo.

```
1  @Bean
2  public Step multiThreadStep(JobRepository jobRepository,
3                              PlatformTransactionManager txManager,
4                              ThreadPoolTaskExecutor taskExecutor) {
```

```
    return new StepBuilder("multiThreadStep", jobRepository)
        .<User, User>chunk(100, txManager)
        .reader(pagingReader) // ATTENZIONE: Deve essere Thread-Safe!
        .writer(writer)
        .taskExecutor(taskExecutor) // Abilita il parallelismo
        .throttleLimit(4) // Max 4 thread attivi contemporaneamente
        .build();
}
```

Listing 50.8: Abilitare il Multithreading

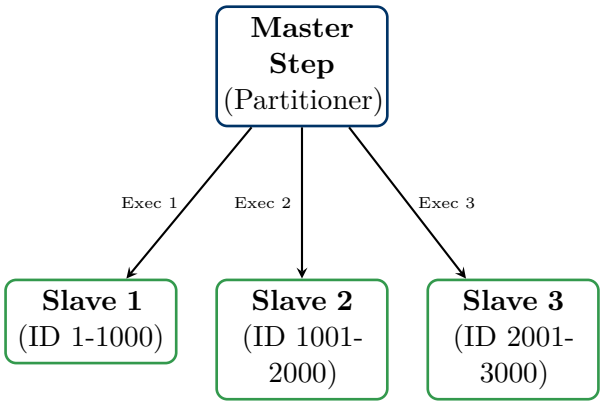
Deep Dive: Attenzione alla Thread Safety del Reader

Questa è la trappola numero uno. Se usi un **Multi-threaded Step**, il tuo `ItemReader` deve essere Thread-Safe.

- **JdbcCursorItemReader:** **NON** è thread-safe. Se lo usi qui, i thread si ruberanno i dati a vicenda o lanceranno eccezioni.
- **JdbcPagingItemReader:** **È** thread-safe. Ogni thread legge una pagina diversa.
- **FlatFileItemReader:** **NON** è thread-safe di default. Va wrappato in un `SynchronizedItemStreamReader`, ma questo crea un collo di bottiglia (i thread si mettono in coda per leggere).

50.6.2 Partitioning (Partizionamento)

Il partitioning è una tecnica più avanzata e robusta. Invece di far competere i thread per leggere dallo stesso Reader, si divide il lavoro a monte. Uno step "Master" (Manager) analizza i dati, crea delle partizioni (es. per range di ID o per file in una cartella) e assegna ogni partizione a uno step "Slave" (Worker) separato.



- **Partitioner:** L'interfaccia che decide come dividere i dati (es. crea una mappa `minValue=1, maxValue=1000`).
- **PartitionHandler:** Gestisce l'esecuzione dei worker (può essere locale coi thread o remoto su altri server).

Colloquio: Quando usare Multi-thread e quando Partitioning?

Domanda: "Come decidi quale strategia di scaling applicare?"

Risposta:

- Uso **Multi-threaded Step** quando ho un collo di bottiglia sulla I/O (es. scrittura lenta su DB) e ho un Reader thread-safe (Paging Reader). È facile da configurare.

- Uso **Partitioning** quando ho bisogno di controllo totale sulle partizioni (es. "Voglio che il Thread A processi solo il File A e il Thread B il File B") o quando voglio scalare su più macchine fisiche (*Remote Partitioning*).

50.6.3 Parallel Steps

Se un Job ha due flussi indipendenti (es. scaricare dati da due fornitori diversi), non serve aspettare che finisca il primo per iniziare il secondo.

```
@Bean
public Job parallelJob(JobRepository repo, Flow flow1, Flow flow2) {
    return new JobBuilder("parallelJob", repo)
        .start(flow1)
        .split(new SimpleAsyncTaskExecutor()) // Esegue in parallelo
        .add(flow2)
        .end()
        .build();
}
```

Listing 50.9: Flussi Paralleli (Split)

50.7 Spring Batch 5 e Spring Boot 3

Con il rilascio di Spring Boot 3 (e di conseguenza Spring Batch 5), il framework ha subito una modernizzazione significativa. Questo aggiornamento ha introdotto dei *Breaking Changes* che rendono il codice scritto per Spring Boot 2 non compilabile senza modifiche.

50.7.1 Principali Novità

1. **Java 17 Obbligatorio:** Spring Batch 5 richiede almeno Java 17 come base.
2. **Rimozione delle Factory:** `JobBuilderFactory` e `StepBuilderFactory` sono state rimosse (non solo deprecate). Non possono più essere iniettate.
3. **Gestione Transazioni Esplicita:** Ora è obbligatorio specificare il `PlatformTransactionManager` in ogni Step (prima veniva spesso dedotto automaticamente dalle factory).
4. **Supporto GraalVM:** Pieno supporto per compilare i batch come immagini native (avvio istantaneo e minore uso di memoria), ideale per job "usa e getta" in ambiente Cloud/Kubernetes.

50.7.2 Migrazione: Dalle Factory ai Builder

Questa è la modifica che impatta il 90% del codice esistente.

Colloquio: Come si configura un Job in Spring Batch 5?

Domanda: "Nel tuo codice vedo che usi `StepBuilderFactory`. Ma in Spring Boot 3 non esiste più. Come lo riscrivi?"

Risposta: Bisogna istanziare manualmente `JobBuilder` e `StepBuilder`, passando esplicitamente il `JobRepository`. Inoltre, nel metodo `.chunk()`, è ora obbligatorio passare il `TransactionManager`.

Confrontiamo i due approcci:

```java
// --- STILE VECCHIO (Spring Boot 2) - NON FUNZIONA PIU' ---
@Autowired
private StepBuilderFactory stepBuilderFactory; // RIMOSSO

@Bean
public Step oldStep() {
    return stepBuilderFactory.get("step1")
            .<User, User>chunk(10) // TxManager era implicito
            .reader(reader)
            .writer(writer)
            .build();
}

// --- STILE NUOVO (Spring Boot 3 / Batch 5) ---
@Bean
public Step newStep(JobRepository jobRepository,
                    PlatformTransactionManager txManager, // Va iniettato
    ↪ esplicitamente
                    ItemReader<User> reader,
                    ItemWriter<User> writer) {

    return new StepBuilder("step1", jobRepository) // Passaggio esplicito repo
            .<User, User>chunk(10, txManager)      // Passaggio esplicito txManager
            .reader(reader)
            .writer(writer)
            .build();
}
```

Listing 50.10: Vecchio Stile vs Nuovo Stile (Batch 5)

50.7.3 Modifiche al Database

Anche lo schema delle tabelle di metadati (`BATCH_*`) è cambiato leggermente (es. gestione delle sequenze su alcuni DB). Se si migra un'applicazione esistente, non basta aggiornare il JAR; bisogna lanciare gli script SQL di migrazione forniti dalla documentazione ufficiale di Spring.

Deep Dive: Perché questo cambiamento?

La rimozione delle Factory favorisce un codice meno "magico". Prima, le factory nascondevano la complessità del `JobRepository` e del `TransactionManager`. Ora, richiedendoli esplicitamente nel builder, si evitano ambiguità in applicazioni complesse con più database o più transaction manager (scenario frequente nelle grandi aziende).

Capitolo 51

Testing in Spring Boot

Nel mondo Enterprise, "funziona sulla mia macchina" non basta. Il testing in Spring non riguarda solo l'uso di JUnit, ma la capacità di caricare il **Context** in modo intelligente.

Spring Boot fornisce uno starter dedicato: `spring-boot-starter-test`, che include già le librerie standard de-facto: **JUnit 5** (il motore), **Mockito** (per i mock), **AssertJ** (per asserzioni leggibili) e **Hamcrest**.

51.1 La Piramide dei Test

Prima di scrivere codice, devi scegliere la strategia. Non tutti i test devono caricare Spring!

1. **Unit Test (Solitari):** Testano la logica di una singola classe. **Non avviano Spring**. Velocissimi (ms).
2. **Slice Test (Integrazione Parziale):** Caricano solo una "fetta" del contesto Spring (es. solo il Web Layer o solo il DB Layer).
3. **Integration Test (Full):** Caricano l'intero `ApplicationContext`. Lenti (secondi/minuti).

51.2 1. Unit Testing (Senza Spring)

Se devi testare un algoritmo dentro un Service, non serve Spring. Usa Mockito puro. È la forma di test più veloce e preferibile.

```
// @ExtendWith abilita le annotazioni di Mockito (@Mock, @InjectMocks)
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository userRepository; // Mock del dipendenza

    @InjectMocks
    private UserService userService; // Classe da testare

    @Test
    void shouldCreateUser() {
        // 1. Arrange (Preparo i dati e il comportamento del mock)
        User input = new User("Mario");
        when(userRepository.save(any())).thenReturn(new User(1L, "Mario"));

        // 2. Act (Eseguo)
```

```
        User result = userService.createUser(input);

        // 3. Assert (Verifico con AssertJ)
        assertThat(result.getId()).isNotNull();
        verify(userRepository).save(any()); // Verifico che il mock sia stato chiamato
    }
}
```

Unit Test Puro con Mockito

51.3 2. Slice Testing (Il bisturi di Spring)

A volte devi testare se il tuo Controller risponde correttamente in JSON, o se la tua Query JPA funziona. Caricare tutta l'app è eccessivo. Spring offre le "Annotation Slices".

51.3.1 @WebMvcTest (Controller Layer)

Carica **SOLO** i Controller, i filtri di sicurezza e la gestione JSON. Non carica Service, Repository o connessioni al DB.

```
@WebMvcTest(UserController.class) // Carica solo questo controller
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc; // Simula le chiamate HTTP senza server reale

    @MockBean // Fondamentale: Sostituisce il Service vero con un Mock nel contesto
    ↪ Spring
    private UserService userService;

    @Test
    void shouldReturnUser() throws Exception {
        when(userService.findById(1L)).thenReturn(new UserDTO("Mario"));

        mockMvc.perform(get("/api/users/1")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.username").value("Mario"));
    }
}
```

51.3.2 @DataJpaTest (Persistence Layer)

Carica solo JPA, Hibernate e il DataSource. Di default configura un database in-memory (**H2**) per velocità.

```
@DataJpaTest
class UserRepositoryTest {

    @Autowired
    private UserRepository repository;

    @Test
    void shouldFindActiveUsers() {
```

```
        repository.save(new User("Mario", true));
        repository.save(new User("Luigi", false));

        List<User> active = repository.findByActiveTrue();

        assertThat(active).hasSize(1);
        assertThat(active.get(0).getUsername()).isEqualTo("Mario");
    }
}
```

51.4 3. Full Integration Test (@SpringBootTest)

Qui si alza tutto: Database, Service, Controller, Security. Simula l'avvio reale dell'applicazione.

```
// webEnvironment = RANDOM_PORT avvia un vero Tomcat su una porta casuale
// per evitare conflitti (es. porta 8080 occupata).
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class FullApplicationTest {

    @Autowired
    private TestRestTemplate restTemplate; // Client HTTP per fare chiamate reali

    @Test
    void flowCompleto() {
        ResponseEntity<String> response = restTemplate.getForEntity("/api/health",
    ↪ String.class);
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    }
}
```

51.5 Mocking: @Mock vs @MockBean

Questa è la domanda da colloquio per eccellenza sui test Spring.

Colloquio: Differenza tra @Mock e @MockBean

Domanda: Quando uso l'uno e quando l'altro?

Risposta:

- `@Mock` **(org.mockito.Mock):** Fa parte di Mockito. Crea un oggetto finto. È scollegato da Spring. Si usa negli **Unit Test** puri.
- `@MockBean` **(org.springframework...):** Fa parte di Spring Boot.
  1. Cerca quel Bean nell'`ApplicationContext`.
  2. Lo **rimuove** (o non lo carica).
  3. Lo **sostituisce** con un Mock.

  Si usa negli **Integration/Slice Test** (es. in `@WebMvcTest` devi "mockare" il Service perché non viene caricato).

Attenzione: Ogni volta che usi `@MockBean`, Spring "sporca" il contesto (Dirty Context). Se hai 100 test e ognuno usa un `@MockBean` diverso, Spring dovrà ricaricare l'applicazione 100 volte. Lento!

51.6 Modern Best Practice: Testcontainers

Usare H2 (DB in memory) per i test e PostgreSQL in produzione è rischioso. H2 potrebbe accettare sintassi SQL che Postgres rifiuta (e viceversa).

Testcontainers è una libreria che avvia un vero Database Dockerizzato per la durata del test.

```java
@SpringBootTest
@Testcontainers // Richiede Docker installato sulla macchina
class PostgresIntegrationTest {

    // Avvia un container Postgres 15 reale
    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:15");

    // Sovrascrive le proprietà di connessione di Spring al volo
    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Autowired
    private UserRepository repository;

    @Test
    void testSuVeroDB() {
        // Questo test gira su un vero Postgres pulito!
        repository.save(new User("Test"));
        assertThat(repository.count()).isEqualTo(1);
    }
}
```

Integrazione con Testcontainers

51.7 Riepilogo Strategia di Testing

Tipo Test	Velocità	Cosa usare
Logica di Business	Altissima	Unit Test puro + Mockito (`@ExtendWith`)
Controller / API	Alta	`@WebMvcTest` + `@MockBean` (Service)
Repository / Query	Media	`@DataJpaTest` + Testcontainers
End-to-End	Bassa	`@SpringBootTest` + `TestRestTemplate`

Capitolo 52

Spring Profiles: Gestione degli Ambienti

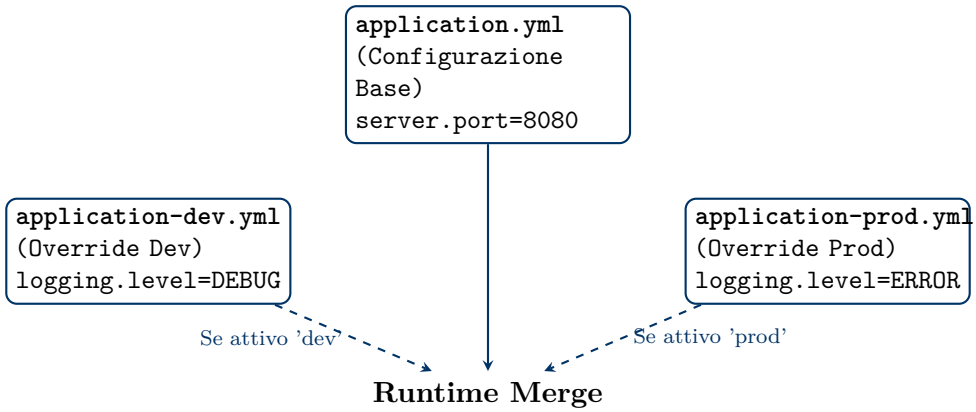
"Funziona sulla mia macchina". Quante volte abbiamo sentito questa frase? Il codice è lo stesso, ma l'ambiente cambia:

- **Dev:** Database H2 in memoria, log a livello DEBUG, mock dei servizi di pagamento.
- **Prod:** Database Oracle clusterizzato, log a livello ERROR, servizi di pagamento reali.

I **Spring Profiles** sono il meccanismo nativo per segregare parti della configurazione (e dei Bean) e attivarle solo in specifici ambienti.

52.1 Come definire un Profilo

Spring Boot usa una convenzione di nomenclatura basata sui suffissi dei file di configurazione.



52.1.1 La Regola dell'Override

Spring carica **sempre** `application.properties` (o `.yml`). Se un profilo è attivo (es. `dev`), carica **anche** `application-dev.properties`. Le proprietà nel file specifico **sovrascrivono** quelle del file base.

52.2 Attivazione dei Profili

Come dico a Spring: "Oggi sei in Produzione"? Esistono 3 modi principali, in ordine di priorità.

52.2.1 1. File di Configurazione (Hardcoded - Sconsigliato)

Nel file `application.properties`: `spring.profiles.active=dev` Va bene solo per lo sviluppo locale. Se lo committi su Git, forzi tutti a usare quel profilo.

52.2.2 2. JVM Argument (Build/CI)

Passato al lancio del JAR. Vince su quello scritto nel file.

```
java -jar -Dspring.profiles.active=prod my-app.jar
```

52.2.3 3. Environment Variable (Cloud Native)

La scelta obbligatoria per Docker e Kubernetes. Definisci la variabile d'ambiente `SPRING_PROFILES_ACTIVE` nel container.

```
export SPRING_PROFILES_ACTIVE=prod
```

Colloquio: Il Profilo "Default"

Domanda: Cosa succede se non attivo nessun profilo? **Risposta:** Spring attiva automaticamente un profilo chiamato **"default"**. Se hai un file `application-default.properties`, verrà caricato. È una best practice usare questo meccanismo per la configurazione di sviluppo locale "out-of-the-box", così il nuovo sviluppatore scarica il repo, preme "Run" e tutto funziona senza configurare nulla.

52.3 Bean Condizionali (@Profile)

Non solo le proprietà, ma interi pezzi di codice (Bean) possono essere caricati o meno a seconda del profilo.

Scenario: In Dev voglio un Database in memoria (H2) che si popola con dati finti. In Prod voglio collegarmi a un vero PostgreSQL.

```
@Configuration
public class DatabaseConfig {

    @Bean
    @Profile("dev") // Caricato SOLO se active=dev
    public DataSource h2DataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .addScript("schema.sql")
            .addScript("mock-data.sql")
            .build();
    }

    @Bean
    @Profile("prod") // Caricato SOLO se active=prod
    public DataSource postgresDataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl(System.getenv("DB_URL")); // Mai hardcodare URL prod!
        return new HikariDataSource(config);
    }
```

```
}
```

Configurazione Datasource Condizionale

52.3.1 Logica Negativa (NOT)

Puoi anche dire "Carica questo bean se NON siamo in produzione".

```
@Service
@Profile("!prod") // Carica in dev, test, default... basta che non sia prod
public class MockEmailService implements EmailService { ... }
```

52.4 Advanced: Profile Groups (Spring Boot 2.4+)

Nei sistemi complessi, un solo profilo non basta. Potresti avere:

- `db-postgres` vs `db-mysql`
- `msg-kafka` vs `msg-rabbit`
- `metrics-prometheus` vs `metrics-datadog`

Attivare l'app con `-Dspring.profiles.active=prod,db-postgres,msg-kafka,metrics-datadog` è scomodo e prono a errori.

Spring Boot 2.4 ha introdotto i **Gruppi di Profili**. Nel file base `application.yml`:

```
spring:
  profiles:
    group:
      # Se attivo 'prod', attiva automaticamente anche questi sottoprofili
      prod:
        - "db-postgres"
        - "msg-kafka"
        - "metrics-datadog"
      # Se attivo 'dev', attiva questi
      dev:
        - "db-h2"
        - "msg-mock"
```

Ora basta lanciare con `active=prod` e Spring si porta dietro tutto il pacchetto.

52.5 Multi-Document YAML

Un trucco "da Senior" per vedere tutte le configurazioni in un colpo d'occhio. Invece di avere 5 file separati, puoi usare i separatori YAML `---` per definire profili diversi nello stesso file.

```
server:
  port: 8080 # Configurazione Default
spring:
  application:
    name: my-app
---
spring:
  config:
    activate:
      on-profile: dev # Sezione valida solo per DEV
server:
```

```yaml
  port: 9090
logging:
  level:
    root: DEBUG
---
spring:
  config:
    activate:
      on-profile: prod # Sezione valida solo per PROD
server:
  port: 80
logging:
  level:
    root: WARN
```

application.yml (Tutto in uno)

Deep Dive: Sicurezza e Profili

Mai committare le password di produzione nel file application-prod.yml! Anche se il file è "per la produzione", finisce nel Git. **Best Practice:** Nel file `application-prod.yml` metti dei segnaposto o riferimenti a variabili d'ambiente: `spring.datasource.password=${DB_PASSWORD}` E inietta la password reale solo al momento del deploy (tramite Kubernetes Secrets o Vault).

Parte VIII

Dev Ops

Capitolo 53

Git e Version Control per Team

Saper usare `git add` e `git commit` è il minimo sindacale. In un colloquio per posizioni Backend, si aspettano che tu sappia come mantenere una history pulita, come risolvere conflitti complessi e quale strategia di branching adottare per la CI/CD.

53.1 Merge vs Rebase: L'eterno dilemma

Questa è la domanda numero uno su Git. Entrambi i comandi servono a integrare le modifiche di un branch in un altro, ma la filosofia cambia radicalmente.

Colloquio: Qual è la differenza tra Merge e Rebase e quando usarli?

1. **Git Merge (`git merge feature`):**
   - Crea un nuovo "Merge Commit" che unisce le due storie.
   - **Pro:** È "non distruttivo". Preserva la storia esatta (chi ha fatto cosa e quando), anche se disordinata.
   - **Contro:** Se il team è grande, la history diventa un groviglio illeggibile ("spaghetti history") pieno di merge commit inutili.
2. **Git Rebase (`git rebase main`):**
   - Prende i tuoi commit dal branch feature e li "riapplica" uno alla volta in cima al branch main, come se li avessi scritti adesso.
   - **Pro:** Crea una history lineare e pulitissima. Facilita il debug (git bisect).
   - **Contro:** Riscrive la storia (cambia gli hash dei commit).

La Regola Aurea (The Golden Rule): Mai usare `rebase` su branch pubblici condivisi (es. `develop` o `main`). Se riscrivi la storia di un branch che altri hanno scaricato, creerai conflitti disastrosi per tutto il team. Usa rebase solo sul tuo branch locale *prima* di fare push.

53.2 Gestione della Storia: Reset vs Revert

Capire come annullare le modifiche distingue chi capisce Git da chi va nel panico.

- **git reset (Hard/Soft):** Sposta il puntatore `HEAD` indietro nel tempo. È come se i commit successivi non fossero mai esistiti.
- **git revert:** Crea un **nuovo commit** che fa l'esatto opposto del commit che vuoi annullare.

Deep Dive: Scenario: Hai pushato un bug in produzione. Cosa usi?

Devi usare `git revert`. Perché? Perché la history di produzione è sacra e condivisa. Se usassi `git reset` e forzassi il push (`push -force`), spaccheresti la history a tutti gli altri sviluppatori che hanno già scaricato quell'aggiornamento. `Revert` è sicuro perché "aggiunge storia" invece di cancellarla.

53.3 Comandi Tattici: Stash, Cherry-Pick e Squash

Strumenti essenziali per la vita quotidiana.

53.3.1 Git Stash

Sei a metà di una feature complessa ("lavoro sporco") e il capo ti chiede di fixare un bug urgente su un altro branch. Non puoi cambiare branch se hai modifiche non committate.

- `git stash`: Salva le modifiche "in tasca" e pulisce la working directory.
- Cambi branch, fixi il bug, torni indietro.
- `git stash pop`: Recuperi le modifiche dalla tasca e continui a lavorare.

53.3.2 Git Cherry-Pick

Ti serve *solo* un commit specifico da un altro branch, non tutto il branch. Es: C'è un fix nel branch `release-2.0` che serve anche in `main`. `git cherry-pick <commit-hash>` copia quel singolo commit nel tuo branch attuale.

53.3.3 Squashing (Interactive Rebase)

Prima di aprire una Pull Request, è buona norma unire i tuoi 10 commit di "wip", "fix typo", "fix again" in un unico commit pulito e significativo. `git rebase -i HEAD~5`.

53.4 Workflow Aziendali: Git Flow vs Trunk-Based

Non decidi tu come usare Git, lo decide l'architetto o il team leader. Devi conoscere i due modelli principali.

Git Flow (Classico)	**Trunk-Based Development (Moderno)**
Struttura rigida con branch lunghevi: `main`, `develop`, `feature/*`, `release/*`, `hotfix/*`.	Un unico branch principale (`main` o `trunk`).
Le feature vivono per giorni o settimane su branch separati prima del merge.	I developer pushano direttamente su main (o su short-lived branches) più volte al giorno.
Ottimo per software "a pacchetto" con versioni precise (v1.0, v1.1).	Essenziale per la **CI/CD** e i rilasci continui (Web App, SaaS).
Rischio: "Merge Hell" alla fine dello sprint.	**Requisito:** Test automatici forti e **Feature Flags** (per nascondere codice non finito in prod).

Colloquio: Perché le aziende tech preferiscono Trunk-Based?

Git Flow ritarda l'integrazione. Se due sviluppatori lavorano su feature separate per 2 settimane, quando proveranno a unire il codice (Merge) avranno conflitti enormi. Con Trunk-Based, l'integrazione è continua ("Continuous Integration"). I conflitti si risolvono subito quando sono piccoli.

Capitolo 54

Docker per Java Developers

Per vent'anni abbiamo spedito JAR o WAR. Oggi spediamo **Container**. Docker risolve l'eterno problema: *"Sulla mia macchina funzionava"*. Invece di chiedere al sistemista di installare Java 17, Tomcat e configurare le variabili d'ambiente, impacchettiamo tutto (OS ridotto + JVM + App) in una scatola standardizzata.

In questo capitolo vedremo come dockerizzare un'app Spring Boot seguendo le best practices di sicurezza e performance.

54.1 Concetti Fondamentali: L'Analogia Java

Se conosci Java, conosci già Docker. I concetti sono paralleli.

Concetto Docker	Concetto Java	Spiegazione
Dockerfile	`MyClass.java`	Il codice sorgente/ricetta che descrive come costruire l'immagine.
Image	`MyClass.class`	Il template compilato, immutabile e statico.
Container	`new MyClass()`	L'istanza in esecuzione dell'immagine (Oggetto). Puoi averne N identiche.
Registry	Maven Repo	Dove carichi e scarichi le immagini (es. Docker Hub).

54.2 Il Primo Dockerfile (Naive Approach)

Il `Dockerfile` è un file di testo (senza estensione) nella root del progetto. Ecco l'approccio base (funzionante, ma non ottimizzato).

```
# 1. Partiamo da una base con Linux + Java 17
FROM eclipse-temurin:17-jdk-alpine

# 2. Creiamo una directory di lavoro
WORKDIR /app

# 3. Copiamo il JAR compilato (dal target di Maven) dentro l'immagine
# Nota: Devi aver fatto 'mvn package' prima sulla tua macchina!
COPY target/my-app-1.0.0.jar app.jar

# 4. Esponiamo la porta (solo documentazione)
EXPOSE 8080

```

```
# 5. Comando di avvio
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Dockerfile Base

Colloquio: JDK vs JRE in Docker

Domanda: Perché in produzione dovresti usare una JRE invece del JDK? **Risposta:** Il JDK contiene compilatore (`javac`) e tool di debug. In produzione non servono. Usare una JRE (o un'immagine *distroless*) riduce la dimensione dell'immagine (risparmio banda/disco) e riduce la superficie di attacco per gli hacker (meno tool disponibili nel container).

54.3 Multi-Stage Build: L'Approccio Senior

Il Dockerfile sopra ha un difetto: ti obbliga ad avere Maven installato sulla tua macchina per generare il JAR. L'approccio **Multi-Stage** usa Docker anche per compilare.


```
# --- STAGE 1: Build ---
FROM maven:3.9-eclipse-temurin-17 AS build
WORKDIR /app

# Copia solo il pom per scaricare le dipendenze (Layer Caching!)
COPY pom.xml .
RUN mvn dependency:go-offline

# Copia il codice e compila
COPY src ./src
RUN mvn clean package -DskipTests

# --- STAGE 2: Runtime ---
FROM eclipse-temurin:17-jre-alpine
WORKDIR /app

# Copia SOLO il JAR dallo Stage 1
COPY --from=build /app/target/*.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Dockerfile Ottimizzato (Multi-Stage)

Deep Dive: Il Layer Caching

Docker costruisce l'immagine a strati. Se non cambi una riga, Docker riusa lo strato cachato. Perché copiamo prima `pom.xml` e poi `src`? Perché il codice sorgente cambia spesso, le dipendenze Maven raramente. Se cambi una riga di codice Java, Docker invalida

la cache dal `COPY src` in poi, ma riusa il layer dove ha scaricato "Mezzo Internet" (le dipendenze Maven). Build ultra-veloci!

54.4 Docker Compose: Orchestrazione Locale

Un'app Java raramente gira da sola. Ha bisogno di un Database (Postgres) o di Redis. Installare Postgres sul tuo PC è noioso. Usiamo **Docker Compose**.

File: `docker-compose.yml`

```
version: '3.8'
services:
  # La nostra App Spring Boot
  backend:
    build: . # Usa il Dockerfile nella cartella corrente
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/mydb
      - SPRING_DATASOURCE_USERNAME=user
      - SPRING_DATASOURCE_PASSWORD=pass
    depends_on:
      - db

  # Il Database
  db:
    image: postgres:15-alpine
    environment:
      - POSTGRES_DB=mydb
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    ports:
      - "5432:5432"
```

Colloquio: Networking in Docker Compose

Domanda: Come fa l'app Java a connettersi al DB? Cosa metto al posto di "localhost"? **Risposta:** In Docker Compose, ogni servizio è raggiungibile tramite il suo **nome del servizio** come hostname. Nel `SPRING_DATASOURCE_URL`, usiamo `jdbc:postgresql://db:5432/...` perché il servizio nel file yaml si chiama `db`. "Localhost" dentro un container si riferisce al container stesso, non al tuo PC!

54.5 JVM e Container: La Memoria

Fino a Java 8 (update 191), la JVM non sapeva di essere in un container. Vedeva la RAM totale del Server (es. 64GB) e cercava di prenderne gran parte, venendo uccisa dal Docker OOM Killer (Out Of Memory) se il container aveva un limite di 1GB.

Da Java 10+ (e backportato su 8), la JVM è **Container Aware**.

Best Practice: Non impostare `-Xmx` fisso (es. `-Xmx512m`) nel Dockerfile. Usa le percentuali.

```
# Nel Dockerfile o Runtime
ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75.0", "-jar", "app.jar"]
```

Questo dice alla JVM: "Usa al massimo il 75% della RAM assegnata al Container". Se alzi il limite del container in Kubernetes, la JVM si adatta automaticamente.

54.6 Comandi Essenziali (Cheat Sheet)

Comando	Azione
`docker build -t my-app .`	Costruisce l'immagine dal Dockerfile corrente.
`docker run -p 8080:8080 my-app`	Avvia il container mappando la porta (Host:Container).
`docker ps`	Lista i container attivi.
`docker logs -f <container_id>`	Guarda i log (il System.out.println) in tempo reale.
`docker-compose up -d`	Avvia tutto lo stack (DB + App) in background.
`docker-compose down`	Ferma e rimuove tutto (Network e Container).

Capitolo 55

Kubernetes per Java Developers

Docker ha risolto il problema del packaging ("Funziona sulla mia macchina"). Ma in produzione non hai un container, ne hai cento. Chi li riavvia se crashano? Chi scala se il traffico aumenta? Chi gestisce le password?

Kubernetes (K8s) è l'orchestratore. Immaginalo come il sistema operativo del Data Center. Per un Java Developer, K8s non è solo "infrastruttura": cambia il modo in cui gestiamo la configurazione, i log e lo stato dell'applicazione.

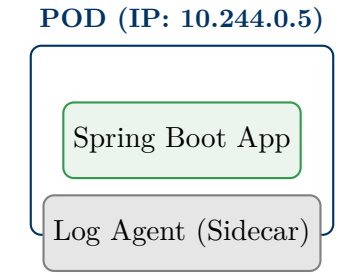
55.1 Architettura: L'Analogia Java

Per capire K8s, usiamo i concetti che già conosciamo.

Oggetto K8s	Concetto Java	Spiegazione
Pod	`Thread`	L'unità minima di esecuzione. Effimero, mortale.
Deployment	`ExecutorService`	Gestisce il ciclo di vita dei Pod (quanti ne voglio, come aggiornarli).
Service	`Interface/Proxy`	Un indirizzo stabile (DNS) per parlare con un gruppo di Pod dinamici.
ConfigMap	`Properties File`	Configurazione iniettata dall'esterno.
Ingress	`DispatcherServlet`	Il punto di ingresso HTTP che smista il traffico.

55.2 Il Pod: L'Atomo di K8s

Docker esegue container. Kubernetes esegue **Pod**. Un Pod è un "guscio" che contiene uno o più container (solitamente uno, la tua app Spring Boot).



Condividono: Localhost, Volumi, IP

Colloquio: Pod vs Container

Domanda: Perché K8s usa i Pod e non direttamente i Container? **Risposta:** Per permettere il pattern **Sidecar**. A volte l'applicazione principale ha bisogno di un "aiutante" (es. un proxy per i log, o un gestore di certificati). In un Pod, questi due container vivono insieme, condividono lo stesso indirizzo IP e possono parlarsi su `localhost`.

55.3 Il Deployment: Scalabilità e Resilienza

Tu non crei mai un Pod manualmente. Crei un **Deployment**. Il Deployment è una dichiarazione di stato desiderato: *"Voglio 3 repliche della mia app"*.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-java-app
spec:
  replicas: 3 # Desired State
  selector:
    matchLabels:
      app: backend
  template: # Il modello del Pod
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: java-app
        image: my-registry/app:v1
        ports:
        - containerPort: 8080
```

deployment.yaml

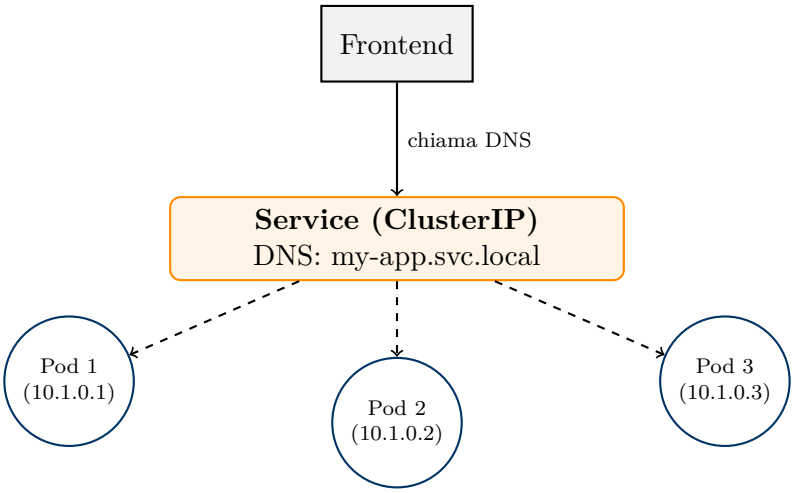
55.3.1 Self-Healing (Auto-Guarigione)

Se la tua app Java crasha (OutOfMemoryError) e il processo termina, K8s se ne accorge. Il Deployment vede che ci sono solo 2 repliche invece di 3 e ne avvia istantaneamente una nuova.

55.4 Service: Networking Stabile

I Pod sono mortali. Ogni volta che un Pod muore e rinasce, **il suo indirizzo IP cambia**. Come fa il Frontend a chiamare il Backend se l'IP cambia sempre?

Il **Service** è un'astrazione che fornisce un IP e un DNS stabili.



Nota: Il Service fa anche da **Load Balancer** interno, distribuendo le richieste tra i Pod disponibili.

55.5 Configurazione: ConfigMap e Secrets

Mai mettere il file `application.properties` dentro l'immagine Docker! K8s ci permette di iniettare la configurazione dall'esterno.

1. **ConfigMap:** Per dati non sensibili (URL DB, Log Level).
2. **Secret:** Per password e certificati (codificati in Base64).

```
env:
  - name: SPRING_DATASOURCE_URL
    valueFrom:
      configMapKeyRef:
        name: db-config
        key: db.url
  - name: SPRING_DATASOURCE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: db.password
```

Iniettare Env Vars in Spring

Spring Boot converte automaticamente le variabili d'ambiente maiuscole (`SPRING_DATASOURCE_URL`) nelle proprietà Java (`spring.datasource.url`).

55.6 JVM in Kubernetes: Memory Limits & OOM

Questa è la causa #1 dei crash in produzione.

In K8s definisci due parametri per la memoria:

- **Request:** La memoria garantita all'avvio.
- **Limit:** Il tetto massimo. Se lo superi, K8s uccide il container (**OOMKilled**).

Deep Dive: Il problema della Heap non visibile

Se imposti il limite del container a 1GB, ma non configuri la JVM, Java potrebbe cercare di allocare più memoria (vedendo la RAM totale del nodo). **Soluzione:** Devi dire alla

JVM di guardare i limiti del container.

```
# Nel Dockerfile o nel Deployment
java -XX:MaxRAMPercentage=75.0 -jar app.jar
```

Questo lascia il 25% di spazio per il Metaspace e l'overhead del container, evitando il temuto OOMKilled.

55.7 Liveness e Readiness Probes (Spring Actuator)

K8s deve sapere se la tua app è viva. Non basta che il processo PID esista (potrebbe essere in deadlock).

Spring Boot Actuator fornisce gli endpoint perfetti per questo.

```
livenessProbe:
  httpGet:
    path: /actuator/health/liveness
    port: 8080
  initialDelaySeconds: 15 # Dai tempo a Spring di partire!

readinessProbe:
  httpGet:
    path: /actuator/health/readiness
    port: 8080
```

Deployment configuration

Colloquio: Liveness vs Readiness

Domanda: Qual è la differenza? **Risposta:**

- **Liveness (Riavvia):** "Sono vivo?". Se fallisce, K8s riavvia il Pod. (Es. Deadlock, stato corrotto irrecuperabile).
- **Readiness (Non mandare traffico):** "Sono pronto a lavorare?". Se fallisce, K8s smette di mandare traffico a questo Pod, ma **non** lo riavvia. (Es. il DB è temporaneamente lento, o l'app si sta ancora avviando).

55.8 Graceful Shutdown

Quando K8s spegne un Pod (scaling down o rolling update), invia un segnale `SIGTERM`. Se l'app si spegne di colpo, le transazioni in corso falliscono.

In `application.properties`:

```
server.shutdown=graceful
spring.lifecycle.timeout-per-shutdown-phase=20s
```

Spring Boot smetterà di accettare nuove richieste, finirà quelle in corso e poi si spegnerà.

55.9 Riepilogo Best Practices

Area	Best Practice
Memoria	Usa sempre `-XX:MaxRAMPercentage`. Imposta Request e Limit nel YAML.
Observability	Abilita **Spring Actuator** e configura Liveness/Readiness Probes.
Config	Usa ConfigMap e Secrets. Non buildare configurazione nell'immagine.
Logs	Scrivi solo su `System.out` (Console). K8s raccoglierà i log (non usare file su disco).
Startup	Spring può essere lento. Configura `initialDelaySeconds` abbondante nei Probe per evitare "Kill loop" all'avvio.

Parte IX

Microservizi e Cloud

Capitolo 56

Monolite vs Microservizi: Scelte Architetturali

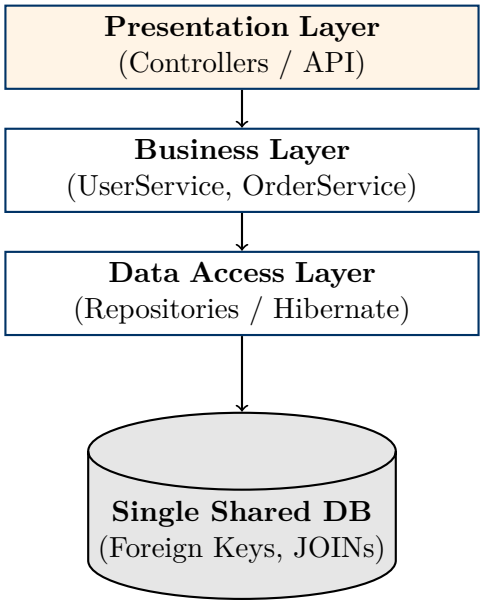
"Dobbiamo riscrivere tutto in Microservizi". Questa frase ha ucciso più progetti aziendali di qualsiasi bug.

Passare da un'architettura Monolitica a una a Microservizi non è un aggiornamento tecnico (come passare da Java 11 a 17). È un cambio di paradigma che sposta la complessità dal **Codice** all'**Infrastruttura**.

In questo capitolo analizzeremo le differenze, i costi nascosti e quando ha senso fare il grande salto.

56.1 Il Monolite Modulare (The Majestic Monolith)

Un'applicazione monolitica è un'unica unità di deployment (un singolo WAR/JAR). Tutta la logica (Utenti, Ordini, Pagamenti) vive nello stesso processo di memoria.



56.1.1 I Vantaggi del Monolite

Non sottovalutarlo. Stack Overflow e Shopify sono partiti (e in gran parte rimasti) monoliti.

- **Semplicità:** Un solo repo git, una sola pipeline CI/CD.

- **Transazioni ACID:** Puoi salvare l'Ordine e scalare i Soldi in un'unica transazione atomica `@Transactional`. O tutto o niente.
- **Performance:** Le chiamate tra servizi sono chiamate a metodo in memoria (nanosecondi), non chiamate di rete (millisecondi).
- **Refactoring:** Spostare una classe da un package all'altro è banale.

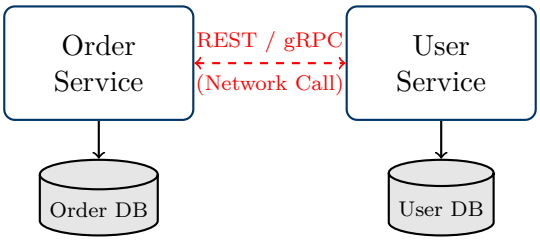
56.1.2 I Problemi (The Big Ball of Mud)

Quando il team cresce da 5 a 50 sviluppatori:

- **Coupling:** Tutto è collegato. Cambiare una riga nel modulo Fatturazione può rompere il modulo Utenti.
- **Scaling:** Se il modulo "Reportistica" consuma tutta la RAM, devi replicare l'intero server (anche la parte login che è leggera), sprecando risorse.
- **Technology Lock-in:** Sei bloccato su Java 8? Aggiornare l'intero monolite è un rischio enorme.

56.2 Microservizi: Sistemi Distribuiti

Nei microservizi, l'applicazione è divisa in piccoli servizi indipendenti, ognuno con il proprio processo, il proprio database e il proprio ciclo di vita.



56.2.1 Database per Service

Questa è la regola d'oro (e la più difficile). **Ogni microservizio deve possedere il proprio database privato.** Nessuno può leggere le tabelle degli ordini tranne l'`OrderService`.

Colloquio: Perché non condividere il DB?

Domanda: Perché non posso collegare tutti i microservizi a un unico Database Oracle gigante? **Risposta:** Creeresti un **Monolite Distribuito**. Se due servizi condividono le tabelle, sono accoppiati. Se l'`OrderService` cambia lo schema della tabella, il `ReportService` si rompe. L'obiettivo dei microservizi è l'**Independent Deployability**: poter rilasciare una nuova versione di un servizio senza chiedere il permesso agli altri team.

56.3 Il Prezzo da Pagare (Trade-offs)

Non è tutto oro quel che luccica.

1. **Transazioni Distribuite:** Non esiste più `@Transactional` tra servizi. Se l'ordine viene creato ma il pagamento fallisce (su un altro servizio), devi gestire il rollback manualmente (Pattern SAGA / Compensazione).
2. **Latenza di Rete:** Una chiamata locale richiede nanosecondi. Una chiamata HTTP richiede millisecondi ed è fallibile (timeout, rete giù).

3. **Eventual Consistency:** I dati non sono aggiornati istantaneamente ovunque. L'utente potrebbe vedere dati vecchi per qualche secondo.
4. **Operational Complexity:** Hai bisogno di Kubernetes, Docker, Distributed Tracing (Zipkin/Micrometer), API Gateway, Service Mesh.

56.4 Quando migrare? (The Complexity Graph)

Martin Fowler, uno dei padri dell'architettura software, ha disegnato un grafico famoso.

Deep Dive: Produttività vs Complessità

- Per sistemi semplici/medi, il **Monolite** è infinitamente più produttivo.
- Solo quando la complessità del sistema diventa ingestibile per un singolo team, i **Microservizi** (nonostante il loro overhead iniziale) permettono di continuare a scalare la produttività.

Senior Tip: Se sei una startup o un team piccolo, parti col Monolite Modulare. Non partire coi Microservizi.

56.5 Strategie di Migrazione: Strangler Fig Pattern

Come si mangia un elefante (Monolite)? Un morso alla volta. Mai riscrivere tutto da zero (The Big Bang Rewrite è un suicidio).

Il pattern **Strangler Fig** (Fico Strangolatore) suggerisce di:

1. Mettere un API Gateway davanti al monolite.
2. Identificare una funzionalità isolata (es. "Notifiche").
3. Scrivere un nuovo Microservizio per le Notifiche.
4. Configurare l'API Gateway per girare le chiamate `/api/notifications` al nuovo servizio e tutto il resto al vecchio monolite.
5. Ripetere finché il monolite non scompare (o diventa molto piccolo).

56.6 Tabella Riassuntiva Definitiva

Aspetto	Monolite	Microservizi
Deployment	Semplice (1 file), ma rischioso (tutto o niente).	Complesso (Orchestrazione K8s), ma granulare.
Database	Condiviso, ACID garantito.	Separato per servizio, Eventual Consistency.
Scalabilità	Verticale (Server più potente) o Orizzontale (Replica intera app).	Granulare (Scalo solo il servizio che serve).
Debug	Facile (Stacktrace locale).	Difficile (Log distribuiti, Tracing ID).
Comunicazione	Metodi Java (Veloce).	REST/gRPC/Kafka (Lento, Fallibile).

Capitolo 57

Advanced Messaging: Apache Kafka

"Se RabbitMQ è una cassetta delle lettere (il messaggio sparisce dopo che lo leggi), Kafka è una biblioteca (il libro resta lì, tu tieni il segno della pagina)."

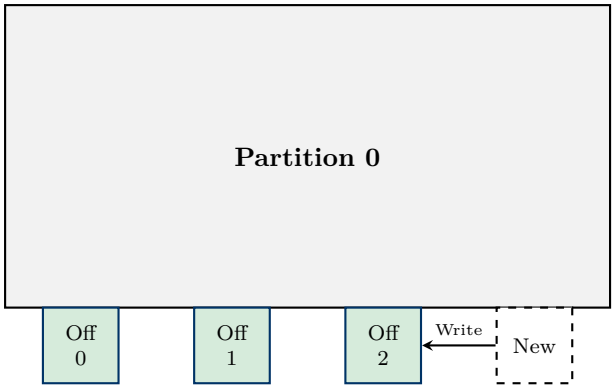
Apache Kafka non è un Message Broker tradizionale. È una **Piattaforma di Streaming Eventi**. È progettato per gestire milioni di messaggi al secondo, persistendoli su disco in modo duraturo.

57.1 Architettura: Log, non Code

In una coda JMS classica, il messaggio viene rimosso dopo il consumo. In Kafka, il messaggio è **immutabile** e viene appeso alla fine di un **Log**. Resta lì per giorni (retention policy), permettendo a diversi consumer di leggerlo in momenti diversi.

57.1.1 Anatomia di un Topic

Un **Topic** è una categoria logica (es. "ordini-creati"). Per scalare, un Topic è diviso in **Partizioni**.



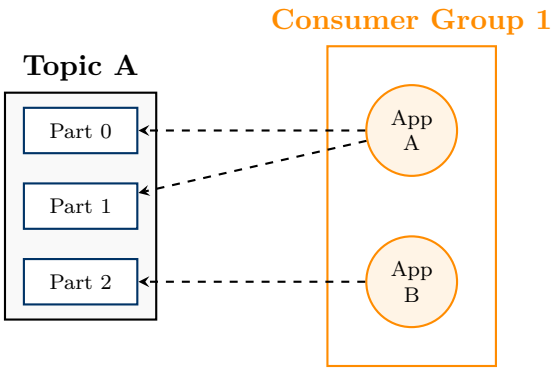
Il Producer scrive sempre in coda

- **Partition:** L'unità di parallelismo. I dati dentro una partizione sono **Ordinati**.
- **Offset:** Un numero sequenziale (ID) che identifica univocamente un messaggio nella partizione.
- **Immutabilità:** Una volta scritto, un messaggio non cambia e non si cancella (fino alla scadenza della retention).

57.2 Consumer Groups: La Magia dello Scaling

Come facciamo a leggere velocemente se arrivano 1000 messaggi al secondo? Non possiamo avere un solo consumer.

Kafka introduce il **Consumer Group**. Un gruppo di applicazioni che collaborano per leggere un Topic. Kafka assegna automaticamente le partizioni ai membri del gruppo.



Deep Dive: La Regola d'Oro dello Scaling

1 Partizione = Max 1 Consumer (dello stesso gruppo). Se hai 3 partizioni e avvii 4 istanze della tua applicazione (Consumer), la quarta istanza rimarrà **Idle** (inattiva). Non riceverà messaggi. Se vuoi più parallelismo, devi aumentare il numero di partizioni del Topic (non solo il numero di consumer).

57.3 Spring Boot e Kafka

Spring fornisce `spring-kafka`, un'astrazione potente che gestisce la connessione e la deserializzazione.

57.3.1 Il Producer (KafkaTemplate)

Inviare messaggi è semplice. Ricorda che è un'operazione asincrona.

```
@Service
@RequiredArgsConstructor
public class OrderProducer {

    private final KafkaTemplate<String, OrderDTO> kafkaTemplate;

    public void sendOrder(OrderDTO order) {
        // Topic, Key (per ordinamento), Value
        // La Key è fondamentale: messaggi con la stessa Key finiscono
        // nella stessa partizione (garantendo l'ordine).
        kafkaTemplate.send("ordini-topic", order.getOrderId(), order);
    }
}
```

57.3.2 Il Consumer (@KafkaListener)

Spring gestisce il polling e il commit degli offset per te.

```java
@Service
public class OrderConsumer {

    @KafkaListener(
        topics = "ordini-topic",
        groupId = "gestione-magazzino-group",
        concurrency = "3" // Avvia 3 thread (come avere 3 consumer)
    )
    public void processOrder(OrderDTO order, Acknowledgment ack) {
        try {
            log.info("Ricevuto ordine: {}", order);
            warehouseService.reserveStock(order);

            // Commit manuale (se configurato)
            ack.acknowledge();
        } catch (Exception e) {
            log.error("Errore processamento", e);
            // Non fare ack -> il messaggio verrà riletto (o andrà in DLQ)
        }
    }
}
```

57.4 Reliability: Non perdere dati

Kafka è configurabile. Puoi scegliere tra velocità e sicurezza.

57.4.1 Producer Acks

Quando il producer considera il messaggio "inviato"?

- **acks=0 (Fire & Forget):** Invia e non aspetta risposta. Velocissimo, rischio perdita dati altissimo.
- **acks=1 (Leader):** Aspetta che il Broker Leader l'abbia scritto su disco. Compromesso standard.
- **acks=all (Strongest):** Aspetta che il Leader E tutte le Repliche l'abbiano scritto. Lento ma sicurissimo.

57.4.2 Consumer Semantics

- **At-most-once:** Leggi, fai commit dell'offset, poi processi. Se crashi durante il processamento, il messaggio è perso.
- **At-least-once (Default):** Leggi, processi, poi fai commit. Se crashi prima del commit, al riavvio rileggi lo stesso messaggio. **Implicazione:** Il tuo codice deve essere **Idempotente** (gestire duplicati).

57.5 Interview Questions

Colloquio: Ordering Guarantee

Domanda: Kafka garantisce l'ordine totale dei messaggi? **Risposta: NO.** Kafka garantisce l'ordine **solo all'interno di una partizione**. Se invii l'ordine A e l'ordine B su partizioni diverse, il consumer potrebbe leggere B prima di A. Se l'ordine è vitale (es. "Crea Utente" prima di "Attiva Utente"), devi assicurarti che entrambi i messaggi abbiano la stessa **Message Key** (es. l'ID utente). Stessa chiave = Stessa partizione.

Colloquio: Kafka vs RabbitMQ

Domanda: Quando scegliere l'uno o l'altro? **Risposta:**

- **RabbitMQ:** Smart Broker, Dumb Consumer. Ottimo per routing complesso, priority queues, e quando vuoi cancellare il messaggio dopo l'uso.
- **Kafka:** Dumb Broker, Smart Consumer. Ottimo per throughput enormi, replay dei dati (riavvolgere il nastro), event sourcing e analytics.

57.6 Riepilogo

Concetto	Dettaglio
Topic	Flusso di dati logico. Diviso in Partizioni fisiche.
Offset	Puntatore di lettura. Il Consumer Group ricorda "dove era arrivato".
Retention	I dati non si cancellano dopo la lettura. Si cancellano per tempo (es. 7 giorni) o dimensione.
Key	Fondamentale per l'ordine. Messaggi con stessa chiave vanno nella stessa partizione.

Capitolo 58

Cloud Native Development e Patterns

"Cloud Native non riguarda *dove* gira la tua applicazione, ma *come* gira."

Un'applicazione Java tradizionale (Legacy) è come un animale domestico (**Pet**): le diamo un nome (server-01), la curiamo quando sta male e se muore è una tragedia. Un'applicazione Cloud Native è bestiame (**Cattle**): sono tutte uguali, numerate, e se una muore viene sostituita immediatamente da una nuova senza che nessuno pianga.

Per scrivere codice "Cattle-ready", dobbiamo seguire la metodologia **12-Factor App** e implementare pattern di resilienza.

58.1 I 12-Factor App (Java Edition)

Di 12 fattori, 4 sono critici per uno sviluppatore Java. Se ne violi uno, la tua app non scalerà su Kubernetes.

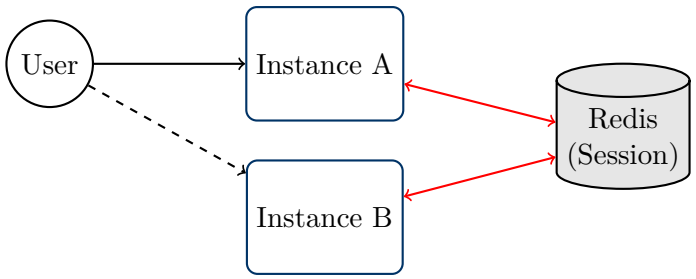
58.1.1 1. Configurazione (Factor III)

Regola: Mai salvare la configurazione nel codice o nel pacchetto (WAR). La configurazione deve venire dall'**Ambiente**.

- **Legacy:** File `config.properties` dentro il JAR. Per cambiare DB, ricompili.
- **Cloud Native:** Spring Boot legge `System.getenv()`. Kubernetes inietta ConfigMaps e Secrets come variabili d'ambiente.

58.1.2 2. Processi Stateless (Factor VI)

Regola: Il processo deve essere senza stato e "Shared-Nothing".



La RAM delle istanze è effimera.
Lo stato va su Redis.

Deep Dive: Sticky Session: L'Anti-Pattern

Se salvi l'oggetto `HttpSession` nella RAM di Tomcat, l'utente è "incollato" a quel server. Se quel server muore (o se K8s scala giù), l'utente viene buttato fuori. **Soluzione:** Usa **Spring Session Data Redis**. La sessione viene salvata su Redis. Qualsiasi istanza può servire qualsiasi utente.

58.1.3 3. Disposability (Factor IX)

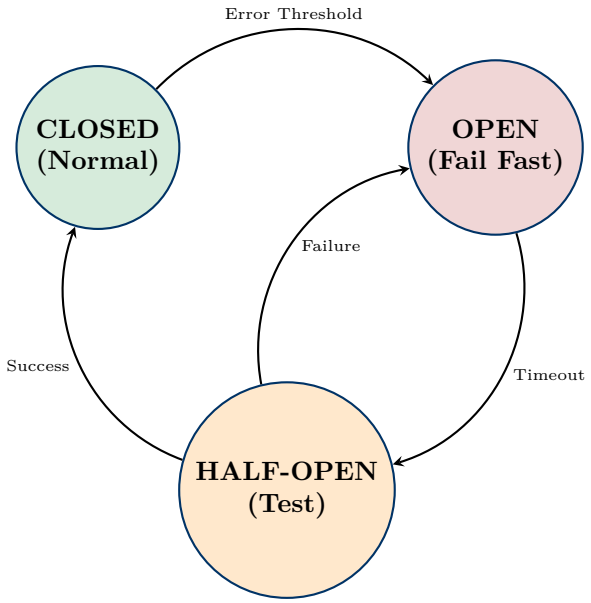
Regola: Avvio rapido e spegnimento grazioso (*Graceful Shutdown*). I container sono mortali. K8s può uccidere il tuo Pod in qualsiasi momento (scaling, update, crash nodo). L'app deve intercettare il `SIGTERM`, smettere di accettare traffico, finire le query in corso e spegnersi entro 30 secondi.

58.2 Design for Failure: Resilienza

Nel Cloud, le cose si rompono. La rete è inaffidabile. Se il Microservizio A chiama B, e B è lento, A non deve bloccarsi aspettando in eterno (esaurendo i Thread).

58.2.1 Il Circuit Breaker Pattern

È un interruttore di protezione. Se le chiamate verso un servizio falliscono ripetutamente, il circuito si "Apre" e le chiamate successive vengono bloccate istantaneamente (Fail Fast) senza intasare la rete, permettendo al sistema remoto di riprendersi.



58.2.2 Implementazione con Resilience4j

Netflix Hystrix è deprecato. Lo standard oggi è **Resilience4j**.

```
@Service
public class PaymentService {

    @Autowired
```

```
    private RestTemplate restTemplate;

    // Configura il Circuit Breaker "pagamenti"
    @CircuitBreaker(name = "pagamenti", fallbackMethod = "fallbackPay")
    @Retry(name = "pagamenti") // Riprova 3 volte prima di arrendersi
    public String processPayment() {
        return restTemplate.getForObject("http://billing-service/pay", String.class);
    }

    // Metodo eseguito se il circuito è APERTO o se c'è eccezione
    public String fallbackPay(Exception e) {
        return "Pagamento differito (Sistema non disponibile)";
    }
}
```

58.3 Spring Cloud vs Kubernetes Native

Questa è una distinzione architetturale fondamentale che un Senior deve conoscere. Ci sono due modi per fare "Cloud Native":

58.3.1 1. La "Old School" (Netflix OSS / Spring Cloud)

L'applicazione gestisce tutto.

- **Discovery:** L'app chiama Eureka Server per sapere gli IP.
- **Config:** L'app chiama Spring Config Server.
- **Load Balancing:** L'app usa Ribbon/Client-Side LB.

Pro: Funziona ovunque (anche su metallo nudo). *Contro:* L'app è pesante e piena di librerie infrastrutturali.

58.3.2 2. La "Modern Way" (Kubernetes Native)

Deleghiamo l'infrastruttura a Kubernetes.

- **Discovery:** K8s Service (DNS).
- **Config:** K8s ConfigMaps (volumi/env vars).
- **Load Balancing:** K8s Service (ClusterIP).

Pro: L'app è agnostica. Il codice è pulito.

Colloquio: Cosa usare oggi?

Domanda: Ha senso usare Eureka se sono su Kubernetes? **Risposta: Generalmente No.** È ridondante. K8s fa già Service Discovery (DNS) e Load Balancing in modo eccellente a livello di rete. Usa le librerie di Spring Cloud solo per logica applicativa interna (es. OpenFeign, Circuit Breaker, Gateway), ma delega il "wiring" infrastrutturale a Kubernetes.

58.4 Distributed Tracing (Observability)

In un monolite, se c'è un errore, guardi lo stacktrace. Nei microservizi, una richiesta attraversa 5 servizi. Dove si è bloccata?

Devi usare uno standard di **Tracing** (come OpenTelemetry / Micrometer Tracing). Ogni richiesta HTTP deve avere due header propagati:

- **Trace ID:** Identificativo unico della transazione globale.
- **Span ID:** Identificativo del singolo passaggio.

```
# Log Service A
[INFO] [TraceID: abc-123] [SpanID: 1] Chiamata a Service B avviata...

# Log Service B
[INFO] [TraceID: abc-123] [SpanID: 2] Ricevuta richiesta da A...
[ERROR] [TraceID: abc-123] [SpanID: 2] DB Timeout!
```

Esempio Log Cloud Native

Cercando `abc-123` su Kibana/Grafana, ricostruisci l'intera storia.

58.5 Riepilogo

Concetto	Cloud Native Way
Stato (Session)	Mai in RAM. Sempre su Redis (Stateless).
Filesystem	Effimero. Se scrivi un file su disco, al riavvio è perso. Usa S3/Blob Storage.
Log	Stream su `System.out`. Non scrivere su file. Ci pensa l'infrastruttura a raccoglierli.
Configurazione	Variabili d'ambiente (Environment Variables).
Resilienza	Fail Fast. Usa Circuit Breakers per non bloccare i thread.

Capitolo 59

AWS per Sviluppatori Java

Il Cloud non è solo "il computer di qualcun altro". È un set di primitive programmabili. Per uno sviluppatore Java, passare ad AWS significa smettere di preoccuparsi dell'hardware e iniziare a comporre servizi gestiti.

L'obiettivo è trasformare l'infrastruttura in codice.

59.1 AWS SDK for Java 2.x

La prima cosa da sapere: esiste una nuova versione dell'SDK.

- **V1 (Legacy):** Bloccante, monolitica, usa classi come `AmazonS3Client`.
- **V2 (Modern):** Non bloccante (basata su Netty), modulare, usa il pattern Builder e Client immutabili (es. `S3Client`, `S3AsyncClient`).

Best Practice: Nei nuovi progetti usa sempre la V2.

```
@Configuration
public class AwsConfig {

    @Bean
    public S3Client s3Client() {
        // Le credenziali vengono lette automaticamente da:
        // 1. Variabili d'ambiente (AWS_ACCESS_KEY_ID)
        // 2. Ruolo IAM (se siamo su EC2/EKS/Lambda)
        // 3. File ~/.aws/credentials (in locale)
        return S3Client.builder()
                .region(Region.EU_SOUTH_1) // Milano
                .build();
    }
}
```

Configurazione S3 Client V2

59.2 Compute: Dove gira il mio JAR?

Hai compilato la tua app Spring Boot. Ora dove la metti? Esistono 4 opzioni principali, con diversi gradi di controllo.

Servizio	Descrizione	Use Case
EC2	Virtual Machine pura (Linux/Windows). Hai il controllo totale (e la responsabilità totale).	Legacy, DB proprietari.
Elastic Beanstalk	PaaS. Carichi il JAR e lui crea l'EC2, il Load Balancer e l'Auto Scaling.	Startup, MVP veloce.
ECS / Fargate	Docker nativo. Tu definisci CPU/RAM e l'immagine Docker. Lui la esegue senza gestire server.	**Microservizi Spring Boot**.
Lambda	Function-as-a-Service. Paghi per millisecondo. Scala a zero.	Event-driven, Task asincroni.

Deep Dive: Java su Lambda: Il problema del Cold Start

Java è famoso per essere lento a partire (caricamento classi, JIT warm-up). Su Lambda, questo è un problema: se la funzione non viene chiamata per un po', AWS spegne l'ambiente. La chiamata successiva potrebbe impiegare 5-10 secondi (Cold Start). **Soluzione:** Usare **AWS Lambda SnapStart** (che fa uno snapshot della memoria dopo l'init) o passare a GraalVM Native Image.

59.3 Storage: S3 (Simple Storage Service)

S3 non è un filesystem. Non puoi fare `new File("/s3/miofile.txt")`. È un Object Store: una mappa gigante `Key` → `Blob` accessibile via HTTP.

Scenario Classico: Upload immagine profilo utente.

```
public void uploadProfilePic(String userId, byte[] imageBytes) {
    PutObjectRequest request = PutObjectRequest.builder()
            .bucket("my-app-user-images")
            .key(userId + "/profile.jpg") // La chiave è il percorso
            .contentType("image/jpeg")
            .build();

    s3Client.putObject(request, RequestBody.fromBytes(imageBytes));
}
```

Colloquio: Database vs S3

Domanda: Perché non salvare le immagini (BLOB) nel Database dentro una colonna `byte[]`? **Risposta:**

- **Costo:** Lo storage DB (EBS/SSD) costa 10x più di S3.
- **Performance:** Caricare blob pesanti intasa la memoria del DB e la rete.
- **CDN:** S3 può essere collegato a CloudFront per servire i file statici globalmente con latenza minima.

59.4 Database: RDS vs DynamoDB

59.4.1 RDS (Relational Database Service)

È il classico SQL (Postgres, MySQL, Oracle) ma gestito. AWS fa i backup, le patch di sicurezza e la replica. Per l'applicazione Java, è trasparente: cambi solo la stringa JDBC.

59.4.2 DynamoDB (NoSQL Serverless)

Qui cambia tutto. Non ci sono Join, non ci sono schemi fissi. È una **Hash Map distribuita** infinitamente scalabile.

Quando usarlo? Alta mole di dati, accessi per chiave primaria, carichi di picco imprevedibili (es. Black Friday).

```
@DynamoDbBean // Simile a @Entity di JPA
public class Order {
    private String orderId; // Partition Key
    private String userId;  // Sort Key

    @DynamoDbPartitionKey
    public String getOrderId() { return orderId; }
    // ... getter e setter
}

// Salvataggio
dynamoDbEnhancedClient.save(order);
```

DynamoDB Enhanced Client (Mappatura Oggetti)

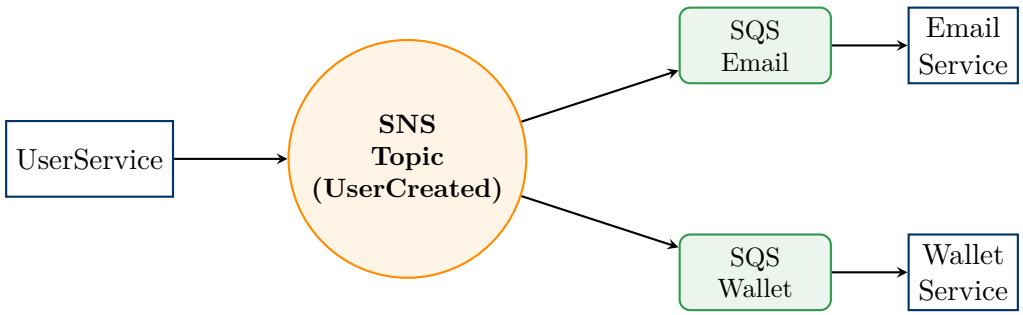
59.5 Messaging: SQS e SNS (Fanout Pattern)

Disaccoppiare i servizi è vitale.

- **SQS (Simple Queue Service):** Coda 1-a-1. Pull-based (il consumer chiede i messaggi).
- **SNS (Simple Notification Service):** Topic 1-a-Molti. Push-based.

59.5.1 Architectural Pattern: Fanout

Vogliamo che quando un utente si registra, succedano due cose: invio mail e creazione wallet. Usiamo SNS per "moltiplicare" il messaggio verso più code SQS.



59.6 Secrets Management: Parameter Store

Mai committare password nel codice o nei file properties. AWS fornisce **SSM Parameter Store** (gratuito per standard throughput) o **Secrets Manager** (a pagamento, rotazione automatica).

Con **Spring Cloud AWS**, puoi importare le proprietà all'avvio automaticamente:

```
spring:
  config:
    import: "aws-parameterstore:/config/myservice/"
```

application.yml (bootstrap)

Se in AWS crei il parametro `/config/myservice/spring.datasource.password`, Spring Boot lo inietterà automaticamente nel DataSource. Magia.

59.7 Riepilogo Scelte Architetturali

Problema	Soluzione AWS Java
Gestione File	**S3**. Non usare il disco locale del container.
Code Asincrone	**SQS**. Garanzia di consegna "At-least-once".
Database Relazionale	**Aurora (RDS)**. Postgres/MySQL compatibile ma cloud-native.
Password	**SSM Parameter Store**. Iniettato a runtime.
Hosting App	**Fargate (ECS)**. Container serverless, ideale per Spring Boot.

Capitolo 60

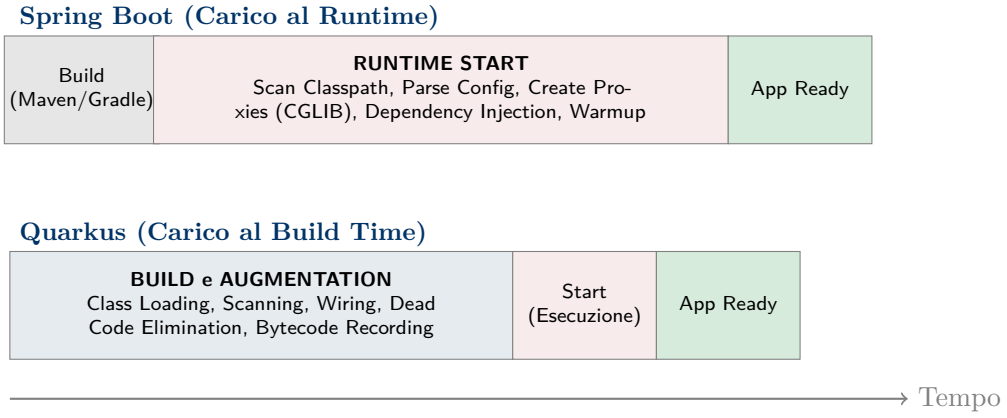
Quarkus: Supersonic Subatomic Java

"Java è lento. Consuma troppa memoria. Non va bene per il Serverless." Queste erano le critiche che stavano spingendo le aziende verso Go e Node.js. Poi è arrivato **Quarkus**.

Quarkus è un framework "Kubernetes Native", progettato per GraalVM e HotSpot. La sua rivoluzione non sta nelle API (che sono standard: JAX-RS, JPA, Hibernate), ma nel **ciclo di vita dell'applicazione**.

60.1 La Filosofia: Build Time First

La differenza fondamentale tra Spring Boot e Quarkus sta in *quando* vengono fatte le cose pesanti.



- **Spring (Runtime):** Quando lanci il JAR, Spring scansiona il classpath, legge le annotazioni, crea i proxy dinamici e inietta i bean. Questo costa CPU e RAM all'avvio.
- **Quarkus (Build Time):** Sposta queste operazioni alla fase di compilazione (`mvn package`). Il bytecode generato è già "cablato". All'avvio deve solo eseguire, non "capire" cosa fare.

60.2 Developer Experience: Live Coding

Dimentica il tasto "Riavvia Server". Quarkus offre la migliore Developer Experience (DX) sul mercato.

Comando magico:

```
mvn quarkus:dev
```

Deep Dive: Come funziona il Dev Mode?

Quando modifichi un file Java e salvi, Quarkus non ricompila tutto. Mantiene l'app in uno stato "sospeso". Appena arriva una richiesta HTTP: 1. Ferma il mondo. 2. Ricompila solo le classi cambiate (millisecondi). 3. Ricarica il contesto. 4. Serve la richiesta. Tutto questo avviene in meno di 0.5 secondi. È trasparente.

60.2.1 Dev Services (Zero Config)

Se aggiungi l'estensione PostgreSQL ma non configuri l'URL nel `application.properties`, Quarkus (in Dev Mode) usa **Testcontainers** per avviare automaticamente un container Postgres Docker, collegarlo all'app e spegnerlo quando chiudi. Niente più `docker-compose` per lo sviluppo locale!

60.3 Hibernate con Panache

Quarkus usa Hibernate, ma offre un layer opzionale chiamato **Panache** che semplifica drasticamente il codice. Supporta due pattern:

60.3.1 1. Repository Pattern (Classico)

Simile a Spring Data.

```
@ApplicationScoped
public class UserRepository implements PanacheRepository<User> {
    // findById, persist, delete... sono già qui.
    // Puoi aggiungere metodi custom:
    public User findByUsername(String name) {
        return find("username", name).firstResult();
    }
}
```

60.3.2 2. Active Record Pattern (Il marchio di fabbrica)

Qui l'Entity sa come salvarsi da sola. Molto amato per la sua sinteticità, ma controverso per i puristi dell'architettura (mescola dati e accesso ai dati).

```
@Entity
public class User extends PanacheEntity {
    // PanacheEntity aggiunge il campo 'id' autogenerato

    public String username; // Campi pubblici! (Panache riscrive il bytecode per
    ↪ renderli getter/setter)
    public LocalDate birthDate;

    // Metodi statici sull'Entity
    public static User findByName(String name) {
        return find("username", name).firstResult();
    }
}

// Utilizzo nel Service
User u = new User();
```

```
u.username = "Mario";
u.persist(); // L'entità si salva da sola!
```

Entity Active Record

60.4 Reactive Core: Mutiny

Spring ha aggiunto la reattività (WebFlux) dopo anni. Quarkus è nato reattivo. Il motore sottostante è **Vert.x** (non-blocking I/O).

Anche se scrivi codice imperativo (bloccante), Quarkus lo esegue in modo efficiente. Ma se vuoi la massima scalabilità, usi le API reattive **Mutiny**.

```
// Uni = 0 o 1 elemento (come Mono)
// Multi = 0 o N elementi (come Flux)

@GET
@Path("/{id}")
public Uni<User> getUser(Long id) {
    return User.findById(id)
        .onItem().ifNull().failWith(new NotFoundException())
        .onItem().transform(u -> {
            log.info("Utente trovato: " + u.username);
            return u;
        });
}
```

60.5 Native Compilation (GraalVM)

Quarkus definisce le estensioni come "Native Ready". Se usi librerie supportate (l'ecosistema Quarkus Extension), compilare in nativo è banale:

```
mvn package -Pnative
```

Metrica	JVM (OpenJDK)	Native (GraalVM)
Startup Time	2.0s	0.05s
RSS Memory (Idle)	140MB	12MB
Throughput	Molto Alto (JIT)	Alto (AOT)
Use Case	Monoliti, Long Running	Serverless, K8s Scaling

60.6 Spring vs Quarkus: Guida alla Scelta

Colloquio: Quando migrare a Quarkus?

Domanda: Ho un monolite Spring Boot. Devo riscriverlo in Quarkus?

Risposta:

- **Resta su Spring Boot se:** Hai un team grande che conosce bene Spring, usi molte librerie legacy non supportate da Quarkus, o l'applicazione è un servizio "always-on" dove 200MB di RAM non sono un problema.
- **Passa a Quarkus se:** Stai facendo **Serverless** (AWS Lambda) dove il cold-start è critico. Stai costruendo microservizi ad alta densità su Kubernetes (vuoi far girare

50 pod su un nodo piccolo). Vuoi ridurre la bolletta del Cloud (meno RAM = istanze più piccole).

60.7 Spring Compatibility API

Quarkus è intelligente. Sa che gli sviluppatori conoscono Spring. Offre estensioni di compatibilità: puoi usare le annotazioni di Spring (`@RestController`, `@Autowired`, `@Service`) dentro Quarkus!

```
// Questo è codice Quarkus, ma sembra Spring!
@RestController
@RequestMapping("/api")
public class SpringGreeterController {

    @GetMapping("/hello")
    public String hello() {
        return "Funziona su Quarkus!";
    }
}
```

Nota: Sotto il cofano vengono tradotte in JAX-RS e CDI a build time. Non c'è il runtime di Spring.

Parte X

Mindset: Troubleshooting e Live Coding

Capitolo 61

Debugging Avanzato: Oltre il Breakpoint

"Un Junior cerca di indovinare dove è il bug cambiando codice a caso (Shotgun Debugging). Un Senior formula un'ipotesi, isola il problema e lo dimostra."

Il debugging non è solo conoscere i tasti dell'IDE. È l'arte di ridurre lo spazio di ricerca. In questo capitolo vedremo strumenti avanzati, il protocollo di debug remoto (fondamentale per Docker) e come analizzare i thread dump.

61.1 Il Mindset Scientifico

Prima di toccare la tastiera, segui il ciclo:

1. **Riproduzione:** Se non puoi riprodurlo costantemente, non puoi fixarlo. Isola lo scenario (test unitario).
2. **Ipotesi:** "Credo che sia colpa della transazione non committata".
3. **Verifica:** Usa i log o il debugger per confermare l'ipotesi.
4. **Fix:** Applica la correzione.
5. **Regression:** Verifica di non aver rotto altro.

61.2 IDE Power Tools (IntelliJ/Eclipse)

Tutti sanno fare "Step Over" (F8). Ecco cosa usano i Pro.

61.2.1 1. Conditional Breakpoints

Hai un ciclo che gira 10.000 volte. Il bug si verifica solo all'iterazione 9.500. Non puoi premere F8 novemila volte.

Soluzione: Tasto destro sul pallino rosso del breakpoint → Condition:

```
user.getId().equals("buggy-user-123")
```

Il debugger si fermerà **solo** quando la condizione è vera.

61.2.2 2. Exception Breakpoints

Il programma crasha con `NullPointerException`, ma i log sono confusi e non sai dove succede. **Soluzione:** Configura l'IDE per fermarsi automaticamente **appena viene lanciata una specifica eccezione**, prima ancora che venga catturata dal `try-catch`.

61.2.3 3. Drop Frame (Rewind)

Hai appena fatto "Step Over" su una funzione, ma ti sei accorto che il bug era *dentro* quella funzione. Troppo tardi? No. Usa **Drop Frame**. Questa funzione "smonta" lo stack corrente e riporta l'esecuzione all'inizio del metodo chiamante, permettendoti di rientrare nel metodo (Step Into) come se avessi riavvolto il tempo. *Nota: Non annulla le modifiche al DB o alle variabili globali, solo lo stack di esecuzione.*

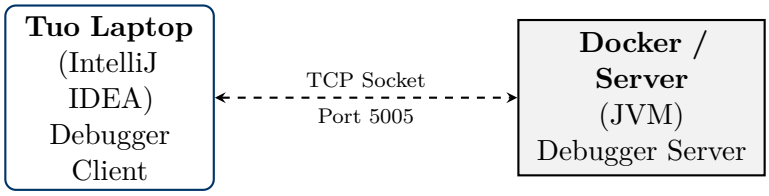
61.2.4 4. Evaluate Expression (Alt+F8)

Mentre sei in pausa, puoi scrivere ed eseguire codice Java arbitrario nel contesto corrente. Puoi persino fare chiamate al DB (`repo.findById(...)`) per verificare lo stato del sistema in quel micro-istante.

61.3 Remote Debugging (JDWP)

Come debuggo un'applicazione che gira dentro un container Docker o su un server di staging? Non ho la GUI lì.

Java possiede il **JDWP (Java Debug Wire Protocol)**. La JVM agisce come un server e l'IDE si connette via TCP.



61.3.1 Configurazione

Per abilitare il debug remoto, devi avviare l'applicazione Java (nel Dockerfile o nello script di avvio) con questi flag:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005 -jar app.jar
```

- **transport=dt_socket:** Usa TCP.
- **server=y:** La JVM aspetta la connessione.
- **suspend=n:** L'app parte subito (se metti 'y', l'app resta congelata all'avvio finché non ti colleghi).
- **address=*:5005:** Ascolta sulla porta 5005 su tutte le interfacce di rete (fondamentale per Docker).

Deep Dive: Security Warning

MAI lasciare la porta di debug (5005) aperta pubblicamente in Produzione. Il protocollo JDWP non ha autenticazione e permette l'esecuzione remota di codice arbitrario. Un hacker potrebbe prendere il controllo totale del server in secondi.

61.4 Analisi dello Stack Trace

Uno stack trace va letto **dal basso verso l'alto** per capire il flusso, ma **dall'alto verso il basso** per trovare l'errore.

```
java.lang.NullPointerException: Cannot invoke "String.length()" because "name" is null
    at com.myapp.service.UserService.process(UserService.java:42)  <-- IL TUO CODICE
    at com.myapp.controller.UserController.create(UserController.java:20)
    at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at ... (Linee di framework Spring/Tomcat inutili) ...
    at java.lang.Thread.run(Thread.java:833)
```

Regola Senior: Ignora le righe che iniziano con `org.springframework`, `org.hibernate`, `java.lang.reflect`. Cerca la prima riga che appartiene al tuo package (`com.myapp`). Lì è esploso il codice.

61.4.1 Caused By: La vera radice

Spesso le eccezioni sono "wrappate". Hibernate lancia `PersistenceException`, che contiene una `ConstraintViolationException`, che contiene una `SQLException`. Scorri sempre fino all'ultimo **"Caused by:"**. Quella è la radice del male.

61.5 Thread Dumps: Debugging dei Blocchi

L'applicazione è attiva, la CPU è al 100% (o allo 0%), ma non risponde. I log sono fermi. Probabile causa: **Deadlock** o **Loop Infinito**.

Un **Thread Dump** è una fotografia istantanea di cosa stanno facendo tutti i thread della JVM.

61.5.1 Come catturarlo

1. Trova il PID: `jps`
2. Cattura dump: `jstack <PID> > dump.txt`

61.5.2 Cosa cercare nel file

Cerca lo stato dei thread:

- **RUNNABLE:** Sta lavorando (consuma CPU). Se molti thread sono runnable nello stesso metodo, hai un loop infinito o un algoritmo lento.
- **BLOCKED:** Sta aspettando un `synchronized` lock posseduto da altri.
- **WAITING:** Sta aspettando un segnale (es. connessione DB dal pool).

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0000... (Object A),
  which is held by "Thread-2"
"Thread-2":
  waiting to lock monitor 0x0000... (Object B),
  which is held by "Thread-1"
```

Esempio di Deadlock nel Dump

Se vedi questo, devi riavviare e correggere l'ordine di acquisizione dei lock nel codice.

61.6 Interview Questions

Colloquio: Heisenbug

Domanda: Cos'è un Heisenbug? **Risposta:** È un bug che scompare o cambia comportamento quando provi a studiarlo (es. attivando il debugger o aggiungendo log). Spesso è causato da **Race Conditions** (concorrenza). L'atto di fermare il thread col debugger altera il timing dell'esecuzione, "risolvendo" temporaneamente il problema di sincronizzazione.

Colloquio: Debug in Prod

Domanda: Il sistema è lento in produzione ma non riesci a riprodurlo in locale. Che fai?
Risposta:

1. Analizzo metriche (CPU/RAM) e Log.
2. Controllo il Distributed Tracing (Span lenti).
3. Eseguo un **Thread Dump** per vedere dove sono bloccati i thread.
4. Eseguo un **Heap Dump** (con cautela, "freeza" la JVM per secondi) per analizzare Memory Leaks con strumenti come Eclipse Memory Analyzer (MAT).

Capitolo 62

Nozioni base di Algoritmi nel Live Coding

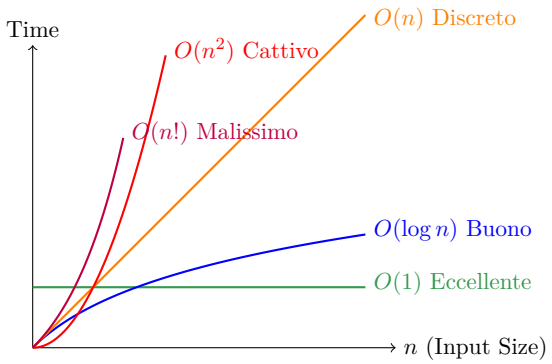
"Non importa quanto conosci Spring Boot se non riesci a scrivere un ciclo for efficiente."

Il Live Coding (o Whiteboard Interview) terrorizza molti sviluppatori Senior. Perché? Perché nel lavoro quotidiano usiamo librerie già pronte. Nelle interviste, invece, dobbiamo dimostrare di saper ragionare sulla **Complessità Computazionale**.

L'obiettivo non è trovare la soluzione "che funziona", ma la soluzione **Scalabile**.

62.1 Big O Notation: Il Linguaggio delle Performance

Non dire "questo codice è veloce". Di' "questo codice è $O(n)$". La Big O descrive come cresce il tempo di esecuzione al crescere dei dati di input (n).



- $O(1)$ **- Costant Time:** Accesso a Map o Array per indice. Il Santo Graal.
- $O(\log n)$ **- Logarithmic:** Binary Search. Tagli il problema a metà ad ogni passo.
- $O(n)$ **- Linear:** Un ciclo for che scorre tutto. Accettabile.
- $O(n^2)$ **- Quadratic:** Due cicli for annidati (Nested Loops). Da evitare come la peste su grandi dataset.

Colloquio: Space Complexity

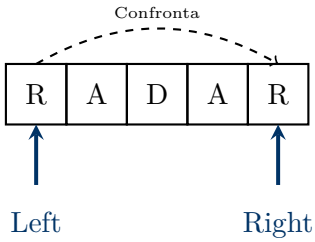
Domanda: "La tua soluzione è $O(n)$ in tempo. E in spazio?" **Risposta:** Spesso dimenticata.

- Se crei una `HashMap` o un array di supporto grande quanto l'input, sei $O(n)$ space.
- Se usi solo poche variabili (`int i, int j`), sei $O(1)$ space.

I senior cercano sempre di ottimizzare lo spazio se il tempo è già ottimo.

62.2 Pattern 1: Two Pointers (I Due Puntatori)

Problema: Data una stringa o un array ordinato, trovare qualcosa (es. palindromo, coppia somma). **Soluzione Naive ($O(n^2)$):** Due cicli for annidati. **Soluzione Two Pointers ($O(n)$):** Un puntatore all'inizio, uno alla fine, si muovono verso il centro.



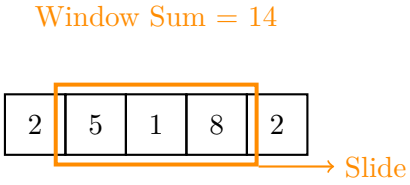

```java
public boolean isPalindrome(String s) {
    int left = 0;
    int right = s.length() - 1;

    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

Valid Palindrome (O(n))

62.3 Pattern 2: Sliding Window (Finestra Scorrevole)

Problema: Trovare il subarray (o sottostringa) contiguo che soddisfa una condizione (es. somma massima di 3 elementi consecutivi). **Soluzione:** Invece di ricalcolare tutto ogni volta, "trascini" una finestra. Sottrai l'elemento che esce, aggiungi quello che entra.




```java
public int maxSubArraySum(int[] arr, int k) {
    int maxSum = 0;
    int windowSum = 0;

    // 1. Calcola somma prima finestra
```

```
    for (int i = 0; i < k; i++) windowSum += arr[i];
    maxSum = windowSum;

    // 2. Slide
    for (int i = k; i < arr.length; i++) {
        windowSum += arr[i];     // Aggiungi elemento nuovo
        windowSum -= arr[i - k]; // Togli elemento vecchio
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}
```

62.4 Pattern 3: Fast e Slow Pointers (Tartaruga e Lepre)

Usato per rilevare cicli nelle LinkedList. Un puntatore va a velocità 1 (`next`), l'altro a velocità 2 (`next.next`). Se c'è un ciclo, prima o poi si incontrano.

```
public boolean hasCycle(ListNode head) {
    ListNode slow = head;
    ListNode fast = head;

    while (fast != null && fast.next != null) {
        slow = slow.next;       // 1 step
        fast = fast.next.next;  // 2 steps

        if (slow == fast) return true; // Collisione!
    }
    return false;
}
```

62.5 Pattern 4: HashMap come "Memoria"

Spesso per abbattere la complessità da $O(n^2)$ a $O(n)$, basta usare memoria ($O(n)$ space). **Esempio (Two Sum):** Trova due numeri che sommati danno il target.

- **Naive:** Doppio ciclo for. "Per ogni numero, cerco se esiste il complementare nel resto dell'array".
- **HashMap:** "Mentre scorro, salvo i numeri visti in una mappa. Per ogni numero, chiedo alla mappa in $O(1)$: 'Hai già visto il mio complementare?'".

```
public int[] twoSum(int[] nums, int target) {
    // Key: Numero, Value: Indice
    Map<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];

        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(nums[i], i);
    }
    throw new IllegalArgumentException("No solution");
}
```

62.6 Binary Search: Oltre l'Array

Tutti sanno fare binary search su un array ordinato. Ma i Senior sanno che si può usare Binary Search su **qualsiasi spazio di ricerca monotono**.

Esempio: "Trova la versione del software che ha introdotto il bug". Se la v10 funziona e la v50 è rotta, il bug è nel mezzo. Testo la v30.

- Se v30 rotta $\rightarrow$ Bug è tra v10 e v29.
- Se v30 funziona $\rightarrow$ Bug è tra v31 e v50.

Questo è $O(\log n)$ invece di testare tutte le versioni.

62.7 Consigli per il Live Coding

Colloquio: Parla mentre scrivi

L'errore più grande è stare zitti per 5 minuti e poi scrivere la soluzione perfetta. L'intervistatore vuole vedere **come ragioni**.

1. **Restate:** Ripeti il problema per essere sicuro di aver capito.
2. **Examples:** Scrivi input/output di esempio sulla lavagna.
3. **Brute Force:** Dì ad alta voce: "Potrei fare due cicli for, ma sarebbe $O(n^2)$. Proviamo a ottimizzare".
4. **Coding:** Scrivi codice pulito. Nomi variabili sensati.
5. **Dry Run:** Esegui mentalmente il codice con un esempio prima di dire "Finito".

Capitolo 63

Algoritmi Avanzati: Grafi, DP e Backtracking

"Chiunque sa scrivere un ciclo for. Pochi sanno scrivere una funzione che chiama se stessa senza causare uno StackOverflow."

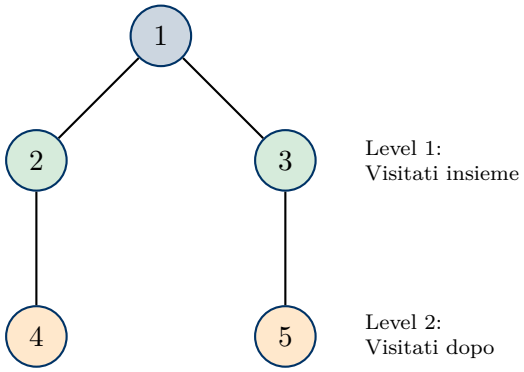
I problemi "Hard" delle Big Tech (Google, Amazon, ma anche banche d'affari) richiedono di modellare la realtà come un Grafo o di ottimizzare la ricorsione tramite la Programmazione Dinamica.

63.1 Grafi: BFS vs DFS

Quasi tutti i problemi di connessione (social network, mappe, dipendenze tra task) sono grafi. Esistono due modi per esplorarli.

63.1.1 1. BFS (Breadth-First Search) - A Livelli

Esplora "a macchia d'olio". Prima i vicini, poi i vicini dei vicini. **Uso:** Trovare il percorso più breve (Shortest Path) in grafi non pesati. **Struttura Dati:** `Queue` (FIFO).




```
public void bfs(Node start) {
    Queue<Node> queue = new LinkedList<>();
    Set<Node> visited = new HashSet<>();

    queue.add(start);
    visited.add(start);

    while (!queue.isEmpty()) {
```

```
        Node current = queue.poll();
        // Processa current...

        for (Node neighbor : current.neighbors) {
            if (!visited.contains(neighbor)) {
                queue.add(neighbor);
                visited.add(neighbor);
            }
        }
    }
}
```

BFS Template

63.1.2 2. DFS (Depth-First Search) - In Profondità

Va il più a fondo possibile in un ramo prima di tornare indietro (Backtrack). **Uso:** Trovare se esiste un percorso, risolvere labirinti, topological sort. **Struttura Dati:** `Stack` (o Ricorsione).

```
public void dfs(Node node, Set<Node> visited) {
    if (visited.contains(node)) return;

    visited.add(node);
    // Processa node...

    for (Node neighbor : node.neighbors) {
        dfs(neighbor, visited);
    }
}
```

DFS Recursion Template

63.2 Recursion e Backtracking

Il Backtracking è una "brute force intelligente". Proviamo tutte le combinazioni possibili, ma ci fermiamo subito (Pruning) se una strada non è valida.

Pattern Classico: Generare permutazioni, Sudoku, N-Queens.

Il Template del Backtracking

1. **Choose:** Fai una scelta (aggiungi elemento alla lista temporanea).
2. **Explore:** Chiama la funzione ricorsiva per il passo successivo.
3. **Un-Choose:** Annulla la scelta (togli elemento) per provare un'altra strada.

```
public void backtrack(List<Integer> temp, int[] nums, boolean[] used) {
    // Caso Base: Soluzione trovata
    if (temp.size() == nums.length) {
        result.add(new ArrayList<>(temp));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (used[i]) continue; // Skip se già usato
```

```
        // 1. Choose
        used[i] = true;
        temp.add(nums[i]);

        // 2. Explore
        backtrack(temp, nums, used);

        // 3. Un-Choose (Backtrack)
        used[i] = false;
        temp.remove(temp.size() - 1);
    }
}
```

Permutazioni [1,2,3]

63.3 Dynamic Programming (DP)

La DP è semplicemente **Ricorsione + Memoria**. Se un problema può essere diviso in sottoproblemi che si ripetono (Overlapping Subproblems), usa la DP.

63.3.1 Top-Down (Memoization)

Scrivi la soluzione ricorsiva, ma prima di calcolare, controlli in una Mappa se l'hai già fatto.

```
Map<Integer, Integer> memo = new HashMap<>();

public int fib(int n) {
    if (n <= 1) return n;

    // Check Cache
    if (memo.containsKey(n)) return memo.get(n);

    // Calcola e Salva
    int res = fib(n - 1) + fib(n - 2);
    memo.put(n, res);

    return res;
}
```

Fibonacci con Memoization

Senza memoization: $O(2^n)$ (Esponenziale - Lento!). Con memoization: $O(n)$ (Lineare - Veloce!).

63.4 Priority Queue (Heap)

Spesso ti chiedono: "Trova i K elementi più grandi in uno stream di numeri". Ordinare tutto l'array ($O(n \log n)$) è troppo lento.

Usa una **Min-Heap** di dimensione K.

- Mantieni nella heap solo i K numeri più grandi visti finora.
- Il più piccolo dei grandi è in cima (root).
- Se arriva un numero nuovo X:
  - Se $X > root$: togli root, inserisci X.

– Se $X < root$: ignoralo (è troppo piccolo per entrare nella top K).

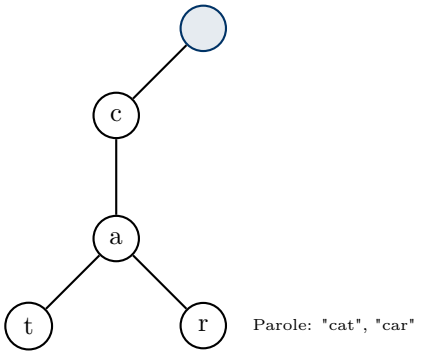
```java
public int findKthLargest(int[] nums, int k) {
    // Min-Heap (default in Java)
    PriorityQueue<Integer> heap = new PriorityQueue<>();

    for (int n : nums) {
        heap.add(n);
        if (heap.size() > k) {
            heap.poll(); // Rimuove il più piccolo
        }
    }
    // In cima rimane il K-esimo elemento più grande
    return heap.peek();
}
```

Complessità: $O(n \log k)$. Molto meglio di ordinare tutto.

63.5 Trie (Prefix Tree)

Se ti chiedono di implementare un **Autocomplete** o un correttore ortografico, la risposta è il Trie. È un albero dove ogni nodo è un carattere.



63.6 Riepilogo Strategia di Scelta

Problema	Algoritmo/Struttura
Shortest Path (non pesato)	**BFS** (Queue).
Maze / Puzzle / Path Exists	**DFS** (Stack/Recursion).
Combinazioni / Permutazioni	**Backtracking**.
Ottimizzazione (Max/Min profitto)	**Dynamic Programming**.
Top K elements	**PriorityQueue** (Heap).
Dictionary / Prefix Search	**Trie**.

Capitolo 64

Soft Skills: Consigli per il Colloquio

"Le Hard Skills ti portano al colloquio. Le Soft Skills ti fanno ottenere il lavoro."

Essere un Senior Developer non significa sapere a memoria le API di Java. Significa saper comunicare, saper gestire l'incertezza e dimostrare di essere un problem solver affidabile. In questo capitolo finale, analizziamo la strategia psicologica per affrontare le selezioni tecniche e comportamentali.

64.1 Gestire il "Non lo so"

La paura più grande del candidato è ricevere una domanda a cui non sa rispondere. **Spoiler:** Succederà. È fatto apposta.

Un Junior prova a indovinare, arrampicandosi sugli specchi. Un Senior usa l'**Onestà Intellettuale**.

Colloquio: Come rispondere quando non sai la risposta

Scenario: "Come funziona l'algoritmo di Garbage Collection G1 nel dettaglio?" (E tu non lo sai).

Risposta Sbagliata: "Eh, pulisce la memoria... è veloce... credo usi dei thread..." (Vago, insicuro).

Risposta Senior: "Onestamente non conosco i dettagli interni del G1 a questo livello di profondità. So che divide la memoria in regioni invece che in generazioni fisiche contigue per ridurre le pause. Se dovessi ottimizzarlo in produzione, leggerei la documentazione ufficiale di Oracle e analizzerei i log del GC con strumenti come GCeasy."

Analisi: Hai ammesso l'ignoranza (Onestà), hai mostrato di conoscere il concetto generale (Competenza) e hai spiegato come risolveresti il problema (Metodo).

64.2 Live Coding: "Think Out Loud"

Durante le sessioni di coding (LeetCode/HackerRank), il silenzio è il tuo nemico. L'intervistatore non è interessato solo alla soluzione finale, ma al tuo **processo mentale**.

Il Protocollo del Live Coding

1. **Ripeti la domanda:** "Se ho capito bene, devo trovare i duplicati in una lista, giusto?". Evita malintesi catastrofici.
2. **Discuti i casi limite:** "Cosa succede se la lista è null? Se è vuota? Se contiene numeri negativi?". Questo dimostra che pensi ai bug prima di scrivere codice.
3. **Proponi prima la Brute Force:** "L'approccio ingenuo sarebbe usare due cicli for ($O(n^2)$). Possiamo ottimizzarlo usando una HashMap ($O(n)$)".
4. **Parla mentre scrivi:** "Ora inizializzo la mappa... qui scorro l'array...".

64.3 Behavioral Interview: Il Metodo STAR

Le domande comportamentali ("Parlami di una volta in cui hai avuto un conflitto con un collega") sono trappole. Se rispondi a caso, sembri disorganizzato o lamentoso.

Usa la struttura **S.T.A.R.**:

Fase	Significato	Esempio
S - Situation	Contesto	"Eravamo vicini al rilascio e il DB era lento."
T - Task	Obiettivo	"Dovevo ottimizzare la query senza rompere il report."
A - Action	Cosa HAI fatto	"Ho analizzato l'Explain Plan, ho visto un Full Table Scan e ho proposto di aggiungere un indice composito." (Usa "IO", non "NOI").
R - Result	Risultato	"La query è passata da 5s a 200ms e il cliente è stato felice." (Usa numeri!).

64.4 System Design: Niente Panico

Se ti chiedono "Disegna un clone di Twitter", non iniziare a disegnare tabelle SQL. È un errore da Junior. Segui questo flusso "Top-Down":

1. **Requisiti Funzionali:** Cosa deve fare? (Postare tweet, seguire utenti).
2. **Requisiti Non Funzionali:** Quanti utenti? (1M o 100M?). Alta disponibilità o Coerenza stretta? (CAP Theorem).
3. **Design Alto Livello:** Disegna i blocchi grandi (Load Balancer, Web Server, Database, Cache).
4. **Deep Dive:** "Il collo di bottiglia sarà la lettura della Timeline. Usiamo Redis per cachare i tweet degli utenti famosi".

64.5 Reverse Interviewing: Intervista l'Azienda

Alla fine del colloquio chiedono sempre: "Hai domande per noi?". Rispondere "No" è un segnale di disinteresse. Fai domande che dimostrano che sei un professionista che tiene alla qualità del proprio lavoro.

Deep Dive: Domande da fare (The Senior Checklist)

- **CI/CD:** "Quanto tempo passa dal commit al deploy in produzione?" (Se dicono "3 mesi", scappa).
- **Testing:** "Qual è la vostra copertura di test? Fate TDD?"
- **Tech Debt:** "Come gestite il debito tecnico? Avete sprint dedicati al refactoring?"

- **On-Call:** "C'è reperibilità notturna? Quante volte suona il telefono?" (Fondamentale per il work-life balance).
- **Team:** "Come è composto il team? Chi decide l'architettura?"

64.6 Red Flags: Quando scappare

Il colloquio è bidirezionale. Loro valutano te, tu valuti loro. Fai attenzione a questi segnali di pericolo:

- **"Siamo una grande famiglia":** Traduzione: ci aspettiamo che tu faccia straordinari non pagati e non abbia confini tra vita e lavoro.
- **"Usiamo tecnologie proprietarie":** Traduzione: imparerai cose inutili fuori da qui e il tuo CV perderà valore.
- **"Il codice è auto-documentante":** Traduzione: non abbiamo documentazione e nessuno sa come funziona il sistema.
- **"Urgenza":** Se ti fanno un'offerta dopo 20 minuti di colloquio, sono disperati. Probabilmente il progetto è un incendio.

64.7 Conclusione

Sei arrivato alla fine di questo manuale. Hai studiato le Collection, dominato i Thread, configurato Spring, ottimizzato Hibernate e imparato a gestire Kubernetes.

Ora hai tutte le carte in regola. Ricorda: la tecnologia cambia, ma la curiosità e la capacità di imparare a fondo ("Under the Hood") non invecchiano mai.