MASSIMILIANO TARQUINI

MATTONE DOPO MATTONE

Java

Un approccio moderno
alla programmazione

EDIZIONE JAVA 25



Nota editoriale

Edizione Java 25 · revisione 14 ottobre 2026

Questa edizione di Java Mattone dopo Mattone, aggiornata a Java 25, è in revisione. Il volume comprende 31 capitoli e 8 raccolte di esercitazioni. Le conclusioni chiudono il percorso dopo le attività.

Le verifiche editoriali e didattiche non sono ancora completate. Il testo e le esercitazioni possono ricevere correzioni e integrazioni.

Nel PDF, indice e segnalibri permettono di raggiungere le diverse parti; nella guida online sono disponibili anche ricerca e navigazione fra le unità.

Indice · Fondamenti

1. Premessa **5**

2. La programmazione a oggetti **13**

3. Linguaggio, JDK e JVM **29**

4. Sintassi, variabili, operatori e array **44**

5. Controllare il flusso del programma **60**

6. Package e moduli **73**

7. Definizione di classi e oggetti **81**

8. Incapsulamento **98**

9. Ereditarietà e composizione **116**

10. Object, classi wrapper ed enumerazioni **140**

11. Stringhe e testo Unicode **149**

12. Eccezioni e gestione degli errori **156**

13. Interfacce e polimorfismo **172**

14. Annotazioni: metadati e controlli **182**

15. Generics e controllo dei tipi **196**

16. Funzioni e lambda in Java **207**

17. Scegliere una collezione **228**

18. Stream API: pipeline, riduzioni e Gatherer **250**

19. Optional: quando una ricerca non trova nulla **267**

20. Thread e concorrenza **278**

Indice · API, pratica e chiusura

20A. Operazioni atomiche **315**

21. Classloader **329**

23. Vector API · approfondimento **339**

24. Reflection API **344**

25. I/O, file e risorse **355**

26. Tempo e numeri esatti **370**

27. Networking e HTTP **376**

28. Memoria e funzioni native **383**

29. Build, test e distribuzione **389**

30. Qualità, sicurezza e diagnosi **403**

Attività C01–C06 **416**

Attività C07–C12 **425**

Attività C13–C18 **433**

Attività C19–C24 **438**

Attività C20A **441**

Attività C25–C28 **444**

Attività C29–C30 **449**

Attività conclusive **453**

Conclusioni. Continuare a costruire, mattone dopo mattone **455**

1. Premessa

Quando nel 2000 iniziai a scrivere la prima versione di *Java Mattone dopo Mattone*, non immaginavo che quel testo avrebbe continuato a circolare per tanto tempo. Lo misi gratuitamente in rete e, in pochi giorni, cominciò a passare da un programmatore all'altro. Non c'era un piano di distribuzione: c'erano persone che lo trovavano utile e lo consigliavano. Anni dopo ho continuato a riconoscerlo sulle scrivanie, spesso accanto a libri e strumenti arrivati molto più tardi. Per chi aveva iniziato mettendo insieme pagine per spiegare Java, era una sorpresa difficile da dimenticare.

Poi arrivò l'edizione Hoepli. Era il 2002 e il libro fu presentato allo SMAU. Le copie disponibili allo stand andarono esaurite in pochi giorni. Ricordo ancora l'emozione di vederlo nella vetrina di una libreria universitaria: un libro che fino a poco prima esisteva soltanto nei miei file era diventato un oggetto che qualcun altro poteva prendere in mano, leggere e magari usare per risolvere un problema. Quella vetrina non era soltanto un traguardo personale. Mi diceva che il metodo con cui avevo provato a costruire le spiegazioni, un mattone dopo l'altro, aveva raggiunto lettori che non conoscevo.



Figura 1.1 – Quattro copertine, una storia. Le prime tre immagini provengono dalla figura della premessa dell'edizione 2022; a queste si affianca la copertina di quella stessa edizione, con i mattoncini colorati. Il montaggio rende visibile il percorso del libro fino al 2022. La qualità delle piccole riproduzioni dipende dagli originali disponibili.

Devo molto a questo libro. Mi ha aperto porte professionali e mi ha permesso di conoscere persone da cui ho imparato. È un debito che non si esaurisce ringraziando i lettori: chiede anche di non lasciare loro fra le mani un'immagine ferma del linguaggio. Quando incontrai Java nel 1996 molte cose che ora diamo per normali non c'erano ancora. Nel frattempo il linguaggio, le librerie e la piattaforma sono cambiati; alcune idee iniziali sono diventate abitudini, altre sono state corrette o abbandonate. È cambiato anche il modo in cui scriviamo, distribuiamo e controlliamo il software.

Tornare sul libro significa quindi fare due lavori insieme. Il primo è aggiornare ciò che il lettore userà quando aprirà un JDK attuale. Il secondo, più delicato, è conservare il modo in cui queste pagine accompagnano chi legge: una domanda concreta, i passaggi che servono per rispondere, un esempio da seguire e le eccezioni che impediscono di imparare una regola sbagliata. Non voglio che l'aggiornamento cancelli i ricordi, le analogie o la pazienza delle spiegazioni.

L'aggiornamento segue questo criterio: rendere più chiaro il percorso e collegare ogni novità al problema che aiuta a risolvere.

Il riferimento tecnico di queste pagine è **Java 25**. La sigla *LTS*, associata a questa versione nella roadmap di supporto Oracle, riguarda il ciclo di supporto di una distribuzione: non significa che ogni programma debba aggiornarsi immediatamente, né che tutte le novità del linguaggio siano nate in Java 25. Quando un costrutto è stato introdotto in una versione precedente, lo dirò nel punto in cui l'informazione aiuta davvero a capirlo o a usarlo. Alcuni esempi potranno mostrare una soluzione storica, ma il percorso principale userà strumenti stabili e pertinenti alla versione di riferimento.

> **Versione Java – Che cosa insegnano le pagine principali.** Una funzionalità *stabile* fa parte del linguaggio o delle API secondo la specifica della versione indicata. Una funzionalità *preview* è disponibile per la prova, ma può cambiare e richiede opzioni esplicite. Un'API *incubator* appartiene a moduli sperimentali. Gli ultimi due casi compariranno, quando utili, in approfondimenti separati: nessun lettore dovrà scambiare un esperimento per la sintassi ordinaria di Java 25. La documentazione ufficiale di Java 25 è il riferimento per questa distinzione.

Il libro che non esiste

Ricordo bene la sensazione di cercare una spiegazione precisa e di trovare, invece, frammenti sparsi: una pagina che dava per scontata la terminologia, un esempio che saltava il passaggio decisivo, un libro che ripartiva ogni volta da ciò che conoscevo già. Cercavo il libro che non c'era, quello che mi aiutasse a crescere senza obbligarmi a leggere tutto nello stesso modo. Con il tempo ho capito anche un'altra cosa: conoscere la sintassi di Java non basta, da sola, a scrivere software comprensibile e affidabile.

Chi ha già perso un pomeriggio per una sola informazione sa che cosa intendo. Il problema non è sempre la mancanza di documentazione: spesso è la sua frammentazione. Trovi la firma di un metodo, ma non il motivo per cui dovresti usarlo; trovi un esempio che funziona, ma non il caso in cui fallisce; trovi una spiegazione completa, ma se conosci già i primi passaggi devi attraversarla tutta per arrivare alla domanda che ti interessa. Il «libro che non esiste» era il desiderio di tenere insieme profondità e libertà di percorso, senza chiedere a ogni lettore di fingere di partire dallo stesso punto.

Questo libro prova a rispondere a entrambe le esigenze. Chi comincia può partire dai concetti più semplici, mettere alla prova un esempio e tornare sul punto che non è ancora chiaro. Chi programma già può raggiungere rapidamente gli argomenti che gli mancano o confrontare le proprie abitudini con strumenti più recenti. Le sezioni avanzate sono riconoscibili; rimandarle è una scelta normale, non una sconfitta. Sono i mattoni da aggiungere quando la struttura sottostante è pronta.

Non ho scritto due libri separati, uno per principianti e uno per esperti. Ho cercato un percorso in cui chi inizia non venga abbandonato al primo termine nuovo e chi ha esperienza possa saltare una base nota senza perdere il filo. Per questo un approfondimento può essere lasciato da parte e ripreso più tardi, ma la spiegazione principale non deve dipendere da ciò che vi è nascosto. Allo stesso modo, una

novità di Java non sarà presentata come un trucco da copiare: vedremo quale problema risolve, come cambia un esempio che già comprendiamo e quali limiti restano.

> **Concetto chiave – Un mattone alla volta.** Un nuovo termine viene spiegato quando serve, il codice viene introdotto prima di essere mostrato e poi letto insieme al risultato. Se un esempio usa un concetto che arriverà più avanti, il rinvio è esplicito. Il lettore non deve indovinare il passaggio mancante.

Il percorso inizia dal modo di pensare un programma e dagli strumenti necessari per eseguirlo. Prosegue con sintassi, oggetti, tipi, API e concorrenza; arriva poi a costruzione, verifica e manutenzione del codice. Le parti più avanzate richiedono qualche fondamento in più, ma non cambiano la promessa iniziale: ogni capitolo deve spiegare perché un problema conta, come affrontarlo e quali limiti ha la soluzione mostrata.

Il percorso si articola in **tre parti**. Nella prima costruiamo il linguaggio: partiamo dalle istruzioni e arriviamo a classi, oggetti e strumenti con cui Java esprime un modello. Non basta riconoscere una parola chiave; voglio che tu sappia leggere un programma e prevedere che cosa accade quando lo esegui. Per questo, quando una nuova forma sintattica compare, la useremo in un problema piccolo e ne seguiremo il risultato.

Nella seconda parte usiamo le API fondamentali e affrontiamo attività che chiedono più di una classe: raccolte di dati, file, comunicazione e lavoro concorrente. Qui le domande cambiano: che cosa promette un'API, chi possiede una risorsa, che cosa accade se due operazioni procedono insieme? Sono passaggi nei quali un programma può compilare ed essere ancora sbagliato. La parte finale guarda al codice nel tempo: qualità, verifiche, manutenzione, strumenti di progetto e analisi statica. Un esempio didattico che funziona oggi è soltanto il primo passo; vogliamo anche capire se un altro programmatore saprà modificarlo domani.

Il confine fra le parti non è un muro. Una buona abitudine di scrittura può comparire già nel primo programma, e una nozione del linguaggio tornerà quando parleremo di manutenzione. I rimandi servono a indicare dove un tema viene sviluppato per intero, non a lasciare incompleta la spiegazione nel punto in cui ne hai bisogno. Se un argomento richiede strumenti che ancora non conosci, lo segnalerò e torneremo a costruirlo quando avremo i mattoni necessari.

Tre modi di attraversarlo

Se Java è nuovo per te, leggi nell'ordine i capitoli iniziali e prova a riprodurre gli esempi prima di guardare le soluzioni. Fermati sui box **Definizione** e **Nota bene**: spesso contengono il dettaglio che evita un errore nelle pagine successive. Puoi lasciare gli **Approfondimenti** per una seconda lettura.

Se programmi già in Java, usa l'indice e le attività di autoverifica per trovare le sezioni che mettono alla prova ciò che sai. Una spiegazione apparentemente elementare può chiarire una distinzione importante, come quella fra classe e oggetto oppure fra compilazione ed esecuzione. Se un risultato ti sorprende, torna all'esempio e verifica il comportamento prima di proseguire.

Se conosci Java di qualche versione fa, cerca i box **Versione Java**. Indicano quando una possibilità è diventata stabile e, dove serve, perché il codice mostrato in un

vecchio progetto può apparire diverso. L'obiettivo non è trasformare il libro in un elenco di release: è arrivare al modello del linguaggio che userai adesso.

I tre percorsi possono incrociarsi. Un programmatore esperto può aver bisogno di tornare alla distinzione fra riferimento e oggetto; un principiante può essere curioso di un box avanzato e leggerlo senza doverlo padroneggiare subito. La marcatura delle sezioni serve a rendere onesta questa libertà. Quando un capitolo richiama un concetto precedente, il rinvio deve indicare il punto giusto; quando una pagina anticipa qualcosa che verrà spiegato dopo, deve dirlo e offrire nel frattempo abbastanza contesto per proseguire. Saltare una sezione non deve significare accettare una lacuna invisibile.

> **Buona pratica – Leggere e provare.** Gli esempi completi devono avere un file sorgente corrispondente nei materiali del libro. Se trascrivi un frammento, controlla nome del file, nome della classe, package e comando indicato. Un errore di battitura è normale; saper leggere il messaggio del compilatore fa parte dell'apprendimento.

Convenzioni

Le convenzioni non servono a decorare la pagina. Ti dicono che cosa stai guardando e quanto contesto occorre per comprenderlo. La figura seguente riunisce i richiami principali, rendendo riconoscibile la funzione di ogni simbolo.

La legenda affianca al simbolo il livello della sezione, così il significato rimane comprensibile anche in una copia in bianco e nero. **Base** vuol dire che stiamo costruendo un prerequisito e che conviene leggere con calma anche se l'argomento sembra semplice. **Approfondimento** segnala un passaggio più impegnativo, utile ma rinviabile senza interrompere il percorso principale.

Questi richiami possono accompagnare sia il titolo di un capitolo sia una sezione interna. Un capitolo può iniziare con nozioni per tutti e contenere più avanti un approfondimento: il segno sul titolo offre una prima indicazione, quello sulla sezione è più preciso. Se manca una marcatura avanzata, il lettore non deve dedurre che l'argomento sia misteriosamente più difficile; il testo seguirà il percorso comune. Le icone non sostituiscono i prerequisiti dichiarati: indicano dove fermarsi, non che cosa si possa conoscere senza spiegazione.

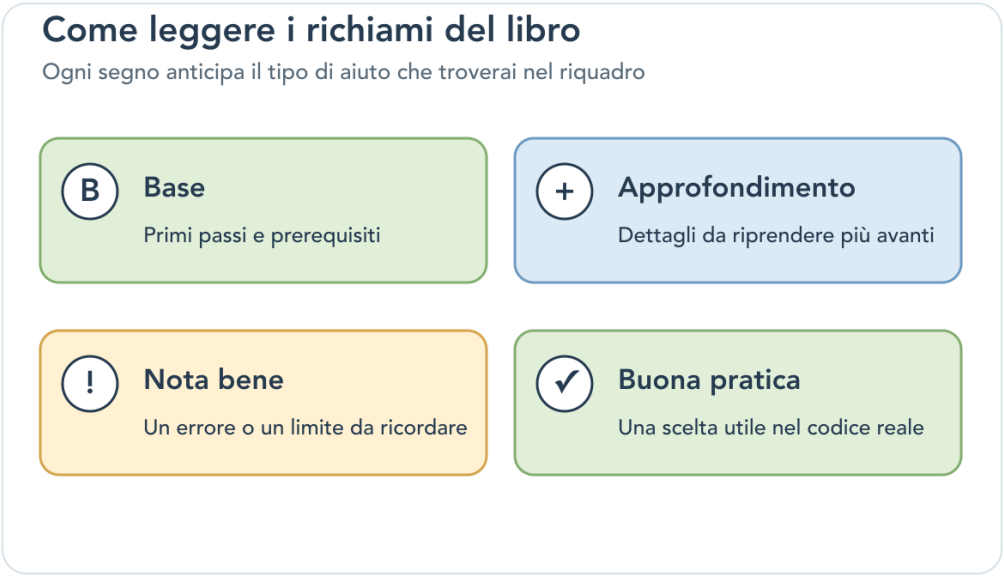


*Figura 1.2 – I richiami aiutano a scegliere il ritmo di lettura. **Base** introduce un prerequisito; **Approfondimento** aggiunge dettagli che possono attendere; **Nota bene** segnala un errore probabile; **Buona pratica** collega un concetto a una scelta concreta.*

I box **Definizione** fissano il significato di un termine che useremo ancora. I box **Concetto chiave** offrono il modello mentale necessario per seguire una sezione. Una **Nota** può aggiungere contesto o una curiosità senza interrompere il filo del discorso. Il box **Versione Java** specifica a quale release e a quale stato appartiene una funzionalità. Queste etichette sono scritte per esteso, così restano comprensibili anche se il libro viene letto senza colori o con un dispositivo che non riproduce bene le icone.

Una **Nota bene** merita attenzione particolare: di solito mette in evidenza un punto su cui è facile costruire un'idea sbagliata. Non è un invito a memorizzare un'eccezione isolata, ma a fermarsi un momento e verificare il modello che abbiamo appena usato. Una **Buona pratica** è diversa: propone una scelta concreta di scrittura o progettazione e ne spiega la ragione. Una definizione, infine, non è un ornamento tipografico; è il patto sul significato di una parola che ritroverai nelle pagine successive.

I richiami di versione distinguono ciò che appartiene stabilmente a Java 25 da ciò che è in prova. Quando indichiamo la versione in cui una funzione è stata introdotta, il numero non serve a creare una cronologia da imparare a memoria. Serve a capire perché un esempio non compila con un JDK precedente o perché due progetti scritti in anni diversi adottano forme differenti. Se una possibilità è ancora preview o incubator, il box lo dice e riporta le opzioni necessarie per provarla in modo separato dal percorso principale.

Nel testo, parole chiave, comandi, nomi di file, classi e package sono scritti come codice: `class`, `javac`, `Saluto.java`, `java.util`. Il corsivo distingue il titolo di un'opera o un termine che stiamo introducendo; il grassetto richiama una parola decisiva nella spiegazione. Un programma completo è separato dalla prosa e ha un

nome di file preciso. Un frammento, invece, viene dichiarato tale: non devi aspettarti che compili da solo.

Anche le figure che rappresentano classi e interfacce seguono una regola semplice. Una freccia di **ereditarietà** sale dalla classe più specifica verso quella più generale; una freccia di **uso** indica chi si serve di un altro oggetto. Quando incontreremo questi rapporti nel capitolo sugli oggetti li interpreteremo insieme. Il colore aiuta a distinguere gli elementi, ma sono etichette e frecce a dare il significato.

Nelle figure tecniche una classe e un'interfaccia hanno forme o etichette distinguibili. «Estende» descrive la relazione fra una sottoclasse e la sua classe base, oppure fra interfacce; «implementa» collega una classe al contratto di un'interfaccia; «usa» dice che un elemento collabora con un altro senza dichiarare una parentela di tipo. Se disegniamo una classe `B` che estende `A`, non stiamo dicendo che `B` contiene fisicamente un oggetto `A` separato. Se `B` implementa l'interfaccia `A`, non stiamo dicendo che eredita lo stato di un'altra classe. E se `B` usa `A`, non abbiamo ancora specificato se la relazione è temporanea o conservata in un campo: lo spiegherà il testo vicino alla figura. Colori e orientamento possono variare per esigenze di pagina; nomi e didascalie devono lasciare intatto il significato.

> **Nota bene – Un esempio non è una prescrizione universale.** Nei primi capitoli alcuni programmi sono piccoli di proposito. Servono a isolare un'idea e possono tralasciare controlli, gestione degli errori o organizzazione che sarebbero necessari in un'applicazione reale. Indicherò questi limiti, senza far passare una scorciatoia didattica per una buona pratica generale.

Licenza d'uso

Questa edizione è distribuita con licenza Creative Commons Attribuzione – Non commerciale 4.0 Internazionale. La licenza consente, alle condizioni indicate nel testo ufficiale, di condividere e adattare il materiale attribuendone la paternità, segnalando le modifiche e senza usarlo per scopi commerciali. Altri diritti, come quelli relativi a immagini di terzi, possono richiedere verifiche specifiche. Per conoscere tutte le condizioni e le eccezioni bisogna leggere la licenza, non affidarsi a questa sintesi.

Vale la pena leggere il significato delle condizioni, perché «libro aperto» non vuol dire privo di regole. Questa licenza permette la condivisione del materiale e la creazione di adattamenti; chi lo fa deve indicare in modo adeguato l'autore, collegare la licenza e rendere riconoscibili le modifiche. L'attribuzione non deve far pensare che l'autore approvi l'uso che altri fanno del testo. La clausola non commerciale limita gli usi rivolti prevalentemente a un vantaggio commerciale o a un compenso. Non si possono poi aggiungere vincoli legali o misure tecnologiche che impediscano agli altri gli usi già permessi dalla licenza.

Ci sono anche limiti che un lettore potrebbe non aspettarsi. Le parti già nel pubblico dominio e gli usi consentiti da eccezioni di legge seguono le proprie regole. La licenza non fornisce garanzie e non risolve automaticamente diritti di terzi: una fotografia, un marchio, l'immagine di una persona o un contributo altrui possono richiedere controlli distinti. Il riepilogo ufficiale Creative Commons aiuta a

orientarsi, mentre il testo giuridico collegato da quella pagina contiene le condizioni complete. Le eventuali attribuzioni e condizioni specifiche degli elementi di terzi vanno considerate insieme alla licenza del volume.

Le fonti

La prima fonte di questo libro è stata l'esperienza di programmare, sbagliare, correggere e spiegare. Quando scrivevo l'edizione del 2022 avevo alle spalle ventisei anni di lavoro professionale come programmatore. In quel conto non entravano le ore passate a scrivere assembler sul Commodore 64, né gli esperimenti con C e Pascal sul mio Amiga 2000. Ho lavorato con linguaggi diversi e in ambiti molto diversi fra loro, dal settore militare alle telecomunicazioni fino ai sistemi enterprise. Ciascuno mi ha insegnato qualcosa sul modo in cui un programma viene scritto, usato e poi mantenuto da altri. Ho incontrato Java nel 1996: per usare le parole con cui lo raccontavo allora, fu amore a prima vista.

Il metodo di questo libro è costruire una spiegazione passo dopo passo. Ogni esempio deve aiutare chi legge a riconoscere un problema, seguire una soluzione e metterla alla prova. Una novità di Java acquista significato quando risponde a una necessità concreta e si collega a un'idea già costruita. Per questo non basta ricordare la sintassi: occorre capire che cosa permette di esprimere, quali condizioni richiede e dove smette di essere utile.

La documentazione ufficiale di Java è il riferimento per sintassi, comportamento della piattaforma e API. Le specifiche descrivono il significato dei costrutti del linguaggio e delle operazioni della macchina virtuale; la documentazione delle librerie dice che cosa promette un metodo a chi lo usa. Consultarle non è soltanto un dovere di precisione: aiuta a distinguere ciò che Java garantisce da ciò che sembra accadere in un singolo programma o in una particolare implementazione. Quando incontreremo una regola delicata, proverò prima a renderla comprensibile con un esempio e poi a indicare dove controllarne la formulazione esatta.

Accanto alla documentazione, *Java Magazine* di Oracle offre spunti, tecniche e riflessioni sul linguaggio; Baeldung è una risorsa pratica per approfondire problemi specifici. Queste letture hanno alimentato il percorso del libro. Un articolo può far nascere la domanda giusta o suggerire una prova; se però stiamo stabilendo che cosa garantisce Java 25, torniamo alla specifica e alle API della versione pertinente. È un modo per usare fonti diverse senza confondere un'idea interessante con una regola del linguaggio.

Infine ci sono le persone. Programmatori amici, colleghi, conoscenti e lettori mi hanno offerto consigli, correzioni e frammenti di codice interessanti. A volte basta una domanda posta nel momento giusto per accorgersi che la spiegazione data per chiara non lo è affatto. Altre volte è un errore segnalato da chi prova davvero un esempio a far migliorare la pagina. Si impara da chiunque abbia qualcosa da dire. Non dimentichiamolo quando una soluzione ci sembra già ovvia.

Ringraziamenti e una piccola scusa

Un grazie particolare va a **Catia Niccolai**, compagna nella vita e collega nel lavoro. Ha seguito da vicino le varie versioni del libro, sopportando i tempi lunghi e le revisioni che una pagina comporta, e ha riletto le bozze. È un lavoro meno visibile di una copertina, ma chi ha provato a portare a termine un libro sa quanto conti una lettura attenta quando l'autore non riesce più a vedere i propri errori.

Infine, una scusa preventiva per qualche esempio che non somiglia a un problema di lavoro. A volte una situazione piccola e perfino un po' artificiale permette di vedere un meccanismo che un'applicazione reale nasconderebbe fra molti dettagli. Quando accade, il motivo sarà dichiarato. E appena quel meccanismo sarà chiaro, proveremo a usarlo in un contesto più credibile.

Per verificare di aver capito

Prima di iniziare il capitolo successivo, prova a rispondere senza tornare indietro: che differenza c'è fra una sezione **Base** e un **Approfondimento**? Dove cercheresti lo stato di una funzionalità sperimentale? Che cosa controlli se un programma copiato dal libro non compila? Le risposte sono nelle convenzioni: riconoscerle adesso ti farà risparmiare tempo durante la lettura.

Per un'attività pratica, prepara una cartella di lavoro per gli esempi e annota la versione del JDK che hai installato eseguendo `java --version` e `javac --version` in un terminale. Conserva i due output. L'attività è completata quando entrambi i comandi sono disponibili e riportano una versione coerente; se uno dei due non viene trovato, il capitolo sugli strumenti spiegherà come distinguere un JDK assente da un percorso di esecuzione non configurato.

2. La programmazione a oggetti

Introduzione

Questo capitolo prepara il terreno per il codice che scriveremo più avanti. Parleremo di oggetti, classi, responsabilità, incapsulamento ed ereditarietà senza pretendere di esaurire argomenti su cui sono stati scritti interi libri. Se è la prima volta che incontri queste parole, non cercare subito di memorizzarle: prova a seguire il problema che le rende utili.

Immagina una piccola biblioteca. Un lettore chiede un libro, qualcuno controlla se è disponibile e, se lo è, registra il prestito. Possiamo descrivere il lavoro come una sequenza di istruzioni, come un insieme di funzioni oppure come una collaborazione fra entità che rappresentano libri, lettori e prestiti. Sono modi diversi di guardare lo stesso problema. Java permette di usarli insieme; l'orientamento agli oggetti è il filo che useremo per organizzare l'applicazione.

> **Definizione – Paradigma di programmazione.** È un modo di formulare i problemi e organizzare le soluzioni. Non è un elenco di parole chiave, né una garanzia di qualità. Un buon programma può contenere parti procedurali, funzioni e oggetti, purché le scelte siano coerenti con il problema.

Un passo indietro: dalla macchina ai concetti

La storia dei linguaggi non è una scala sulla quale ogni gradino rende inutile il precedente. Il linguaggio macchina descrive istruzioni eseguibili da un processore; l'assembly dà nomi simbolici a istruzioni e dati, ma resta vicino all'architettura della macchina. I linguaggi di livello più alto consentono di esprimere algoritmi, strutture e vincoli senza riscriverli ogni volta come una successione di istruzioni elementari. Rimangono, però, linguaggi: nessuno di essi comprende da solo ciò che intendiamo per *libro disponibile* o *prestito valido*. Quel significato dobbiamo costruirlo noi.

Fermiamoci su Ada Lovelace e Charles Babbage, perché la domanda che li accompagnava è la nostra: come si descrive a una macchina una sequenza di operazioni, senza confondere la sequenza con il problema che vogliamo risolvere? Nelle note che accompagnavano la sua traduzione di un articolo sulla macchina analitica di Babbage, pubblicata nel 1843, Lovelace presentò anche una tabella di istruzioni per il calcolo dei numeri di Bernoulli. La riproduzione e la spiegazione conservate dal Computer History Museum mostrano il documento storico. Non occorre conoscere quei numeri per cogliere il punto: un calcolo complesso viene scomposto in passi che una macchina progettata per eseguirli avrebbe potuto seguire. Chiamarla semplicemente «il primo programma» cancellerebbe discussioni storiche che qui non ci servono; ciò che ci interessa è l'idea di rendere esplicite operazioni e dati.

Un altro nome importante per questo sguardo storico è **Konrad Zuse**. Il suo *Plankalkül* fu un progetto di linguaggio ad alto livello assai precoce, ricordato nella scheda storica del Computer History Museum. Non immaginare una linea retta che va da quelle idee a Java, come se ogni linguaggio fosse nato per correggere il

precedente. I problemi, le macchine disponibili e gli scopi erano diversi. Il breve sguardo storico serve a vedere che la ricerca di un'espressione più vicina al problema accompagna la programmazione da molto prima dei moderni linguaggi a oggetti.

La figura confronta i due modi di descrivere il calcolo attraverso uno schema, senza presentarsi come facsimile dei documenti. Nel riquadro di Lovelace guarda le tre colonne: un'operazione usa variabili e porta a un risultato. Nel riquadro di Zuse osserva la diversa ambizione: una notazione per descrivere dati e regole a un livello meno legato alle istruzioni immediate della macchina. I due documenti non sono tappe di una discendenza diretta verso Java; sono prove storiche di domande che torneranno nel nostro esempio della biblioteca.

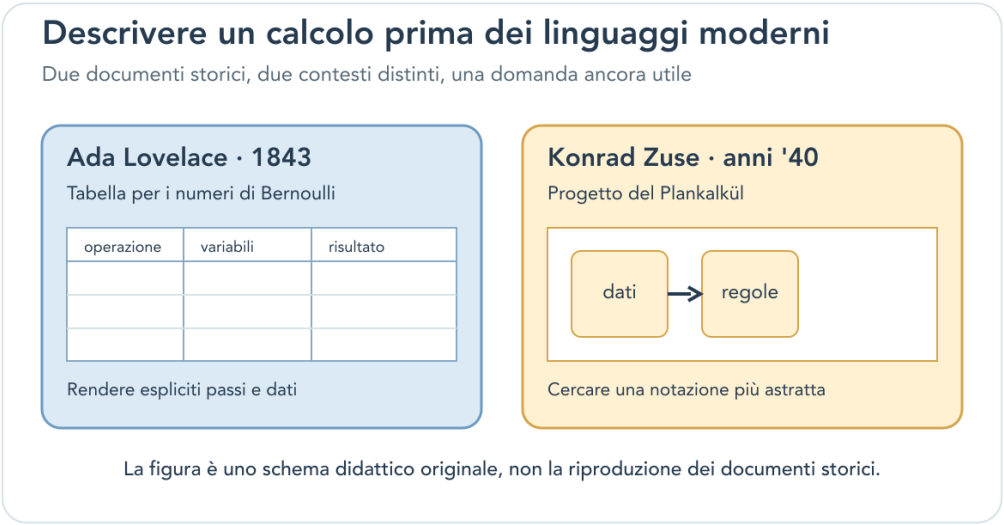


Figura 2.1 – La tabella di Lovelace e il progetto di Zuse sono richiamati con uno schema originale, non con riproduzioni dei documenti. Le fonti storiche collegate nel testo permettono di vedere i materiali; lo schema indica che cosa osservare in essi.

Partiamo allora da ciò che esegue il processore. Nel *linguaggio macchina* le istruzioni sono codificate in una forma specifica dell'architettura: un programma costruito per una famiglia di processori non diventa automaticamente eseguibile su un'altra. Il linguaggio **assembly** sostituisce a molte codifiche numeriche nomi simbolici per operazioni, registri ed etichette. Se vogliamo sommare due valori, anziché ricordare l'intera codifica dell'istruzione possiamo scrivere un'operazione con un nome mnemonico. Questo aiuta una persona a leggere il programma, ma non elimina la necessità di ragionare su registri, indirizzi e istruzioni disponibili su quella macchina. Un esempio di assembly richiede sempre l'indicazione dell'architettura a cui appartiene: non esiste una singola sintassi assembly valida per ogni processore.

Un piccolo «Hello, world!» in assembly fa vedere *quali dettagli* il programmatore deve seguire. Questo è un **esempio storico per Linux a 32 bit su x86**, scritto nella sintassi NASM e con le vecchie chiamate di sistema tramite `int 0x80`; non è un listato Java e non va letto come una ricetta per un sistema moderno.

```
section .text
    global _start
_start:
    mov edx, len
    mov ecx, msg
    mov ebx, 1
    mov eax, 4
    int 0x80
    mov eax, 1
    mov ebx, 0
    int 0x80

section .data
msg db 'Hello, world!', 0xa
len equ $ - msg
```

Le ultime due righe definiscono il messaggio e ne calcolano la lunghezza. Le istruzioni `mov` collocano nei registri i dati richiesti dalla chiamata di scrittura: lunghezza, indirizzo del messaggio, destinazione e numero dell'operazione. `int 0x80` passa il controllo al sistema operativo; la seconda chiamata termina il processo. Persino per stampare una frase dobbiamo quindi sapere **dove** si trovano i dati e **come** quella piattaforma chiede l'output. Nel programma della biblioteca vorremmo invece poter ragionare prima sulla richiesta di un prestito. La distanza fra questi due livelli spiega l'utilità delle astrazioni successive, senza cancellare il fatto che alla fine il programma deve pur essere eseguito da una macchina.

Con i linguaggi di livello più alto possiamo dare un nome a una procedura come `registraPrestito` e lasciare che compilatore e ambiente di esecuzione traducano la descrizione nelle operazioni necessarie. La distanza si riduce, ma non scompare: il computer non sa da solo che una copia già in prestito non può essere prestata di nuovo. Questa è una regola del nostro problema. Lo stesso valeva per l'esempio storico del linguaggio C, nato in un contesto diverso da Java: offriva strumenti per comporre procedure e lavorare vicino alla macchina, ma la qualità della soluzione dipendeva ancora da come il programmatore organizzava dati e responsabilità.

> **Approfondimento – Perché guardare indietro.** La storia non dimostra che ogni paradigma sia «migliore» di quello precedente. Mostra piuttosto che le persone hanno inventato astrazioni diverse per governare la complessità. Un'astrazione è una descrizione che lascia sullo sfondo alcuni dettagli per mettere a fuoco quelli importanti in quel momento. L'assembly nasconde parte della codifica numerica delle istruzioni; una procedura nasconde i passi necessari a un compito; un oggetto può nascondere il modo in cui conserva il proprio stato. In ciascun caso bisogna ancora sapere quali dettagli tornano importanti quando qualcosa non funziona.

Guardare il lavoro come una procedura

Torniamo alla biblioteca. Possiamo scrivere il flusso come quattro passi: ricevere la richiesta, cercare il libro, controllarne la disponibilità e registrare il prestito. La figura mostra questo ordine. Una *procedura* raccoglie istruzioni che svolgono un compito; una *funzione* è una forma di procedura che produce un risultato utilizzabile da chi la chiama. Nel linguaggio quotidiano i termini si sovrappongono talvolta, ma qui ci interessa la domanda pratica: che cosa deve ricevere ciascun pezzo di programma e che cosa deve restituire?



Figura 2.2 – Dal titolo richiesto alla registrazione del prestito. Lo schema rende visibile il flusso procedurale usando nomi che ritroveremo nelle figure successive.

Dividere il lavoro in passi piccoli rende più semplice leggere e correggere il programma. Il pericolo nasce quando ogni passo può modificare liberamente dati condivisi. Se, per esempio, il numero dei prestiti attivi è una variabile accessibile a qualsiasi parte dell'applicazione, chi registra un prestito può aggiornarla mentre un'altra parte la cambia per un motivo diverso. L'errore non dipende dal fatto che il programma sia *procedurale*: dipende da una dipendenza nascosta fra operazioni e stato.

> **Nota bene – I dati globali non sono obbligatori.** Una procedura può ricevere i dati necessari come parametri e restituire un risultato. Anche un programma procedurale può essere modulare e ben progettato. La frase «le procedure usano necessariamente variabili globali» darebbe un'immagine falsa del paradigma.

Chiamiamo **divide et impera** questo modo di affrontare un problema: dividere ciò che sembra troppo grande in compiti più piccoli, ciascuno abbastanza semplice da poter essere compreso e controllato. La metafora non dice ancora *come* dividere. Se spezziamo una funzione lunga in dieci funzioni che leggono tutte la stessa variabile globale, abbiamo ottenuto file più corti ma non responsabilità più chiare.

Se invece una funzione riceve il libro richiesto e restituisce un esito che dichiara se il prestito è stato registrato, chi la chiama può ragionare sul suo contratto senza ispezionare ogni istruzione interna.

La distinzione fra **parametro**, **valore restituito** e **dato condiviso** sarà più facile da vedere nel codice dei capitoli successivi, ma possiamo già seguirla a parole. Un parametro è un'informazione che il chiamante passa alla procedura; un valore restituito è la risposta che la procedura consegna al chiamante. Un dato condiviso rimane invece accessibile da più parti del programma anche quando non compare nella chiamata. Può essere utile, ma è una dipendenza che deve essere progettata e dichiarata. In una biblioteca il catalogo può essere un dato condiviso; il titolo che il lettore sta cercando è più naturalmente un parametro della ricerca. La funzione che cerca può restituire l'elenco degli esemplari trovati.

Seguiamo un errore, invece di limitarci a nominarlo

Immaginiamo che la biblioteca conservi in una variabile condivisa il numero di prestiti attivi. La procedura `registraPrestito` aggiunge una registrazione e aumenta quel numero. La procedura `stampaRicevuta`, invece, si limita a mostrare un messaggio. Finché tutti passano da `registraPrestito`, il numero corrisponde alle registrazioni. Un nuovo collega aggiunge `prestitoRapido`: crea direttamente la registrazione e poi chiama `stampaRicevuta`. Il lettore riceve la sua ricevuta, perciò a prima vista sembra tutto corretto. Solo più tardi un controllo mostra un prestito in meno del dovuto.

Il difetto non sta nella stampa né nel valore iniziale del contatore. Sta nel fatto che **due operazioni che dovevano avvenire insieme erano accessibili separatamente**. Se correggessimo soltanto `prestitoRapido`, un'altra funzione potrebbe ripetere domani lo stesso errore. La soluzione più solida è concentrare la regola in un unico punto: per creare un prestito valido bisogna chiamare un'operazione che registra il prestito e aggiorna, o ricava, il totale. La stampa riceve il risultato di quell'operazione e non modifica il conteggio.

Nel nostro schema, `Print` e `Output` rendono visibile lo stesso rischio. `Print` preparava la stampa e aggiornava alcuni dati; `Output` effettuava soltanto l'operazione di basso livello verso il terminale. Se una nuova funzione chiamava direttamente `Output`, saltava il passaggio affidato a `Print`. Il messaggio appariva sullo schermo, ma lo stato che il resto del programma si aspettava aggiornato rimaneva vecchio. L'errore poteva emergere molto più tardi, in una funzione che leggeva quel dato e sembrava del tutto estranea alla stampa. Non è necessario sostenere che ogni programma procedurale sia difficile da debuggare per riconoscere il problema: qui il nesso fra chiamata e modifica dei dati era invisibile a chi usava `Output`.



Figura 2.2a – Due percorsi producono lo stesso messaggio, ma soltanto quello che passa da `Print` *mantiene il dato condiviso atteso. La freccia tratteggiata indica l'aggiornamento mancante; la conseguenza può propagarsi fino a* `main`*.*

Possiamo seguire la propagazione senza attribuirle qualcosa di misterioso. Supponi che `f4()` legga il contatore rimasto indietro e scelga un ramo errato; `f2()` e `f3()` ricevono allora valori incoerenti; infine il programma principale mostra un risultato sbagliato o termina per un controllo fallito. Il punto in cui si osserva il problema non coincide con il punto in cui è stato introdotto. Una catena che arriva fino a `main` è un buon motivo per disegnare i confini di un modulo prima di guardare soltanto la riga che fallisce.

> **Definizione – Effetto collaterale.** Un'operazione produce un effetto collaterale quando modifica qualcosa che chi la chiama può osservare oltre al suo valore di ritorno: per esempio un contatore condiviso, un file o il contenuto dello schermo. Un effetto collaterale può essere necessario; diventa pericoloso quando resta nascosto oppure non è coordinato con le altre modifiche.

Supponiamo ora che una schermata legga il totale mentre un'altra parte del programma sta aggiungendo un prestito. Abbiamo un problema ulteriore, quello della concorrenza, che richiederà regole specifiche. Per il momento ci basta capire la causa più semplice: i dati condivisi moltiplicano i punti dai quali una regola può essere infranta. Limitare l'accesso a quei dati riduce i punti da controllare, ma non sostituisce la verifica delle operazioni che rimangono pubbliche.

Correggere una dipendenza nascosta

Raggruppiamo le operazioni secondo la loro responsabilità. Un modulo riceve la richiesta, un modulo conosce il catalogo, un modulo registra i prestiti. Chi usa il catalogo non deve conoscere i dettagli con cui è memorizzato; gli basta un'operazione che risponda alla domanda «questo libro è disponibile?». Nella figura, i nomi delle parti rendono riconoscibile lo stesso lavoro mostrato prima.



Figura 2.3 – Lo stesso problema suddiviso per responsabilità. La separazione limita ciò che ogni parte deve conoscere e rende più facile localizzare una modifica.

Questo è un primo esempio di *incapsulamento*: i dettagli interni di una parte sono raggiungibili soltanto attraverso le operazioni che essa sceglie di esporre. L'incapsulamento non nasce con gli oggetti. Linguaggi procedurali e moduli lo possono esprimere con strumenti diversi. Gli oggetti porteranno la stessa idea in una forma che associa stato e comportamento a un'entità del modello.

> **Approfondimento – Due significati di `static` in C.** Una variabile locale `static` conserva il proprio valore fra chiamate della funzione che la contiene, mentre il suo nome resta visibile soltanto in quella funzione. Una funzione dichiarata `static` a livello di file ha invece visibilità limitata al file sorgente in cui è definita. Il primo uso riguarda la durata del dato, il secondo la visibilità di una funzione. In Java incontreremo la stessa parola chiave, ma con regole proprie del linguaggio: non dobbiamo trasferire automaticamente l'intuizione costruita sul C.

Nella figura il modulo dei prestiti espone `registra` e `cerca`, mentre trattiene la struttura che contiene le registrazioni. Il modulo delle ricevute non aggiorna il conteggio: riceve i dati da mostrare. Questa distribuzione non è un trucco del linguaggio; è una decisione su **chi può fare che cosa**. Se cambiamo il modo di conservare i prestiti, chi stampa una ricevuta continua a usare l'operazione pubblica, purché il suo significato resti uguale.

Vale la pena riprendere con precisione il piccolo esempio storico del C. Una variabile locale dichiarata `static` conserva il valore fra chiamate alla funzione, pur essendo visibile solo lì. Una funzione dichiarata `static` a livello di file è invece visibile soltanto all'interno di quell'unità di traduzione. Sono due usi diversi della stessa parola chiave. Nessuno dei due rende automaticamente corretta la logica: una funzione che somma valori in una variabile persistente, per esempio, richiede comunque una decisione su come azzerare la somma e su che cosa accade se viene

usata da chiamate concorrenti. In Java vedremo altri strumenti per esprimere visibilità e stato; non trasferiremo meccanicamente la regola del C alla parola `static` che incontreremo nel codice Java.

E le funzioni?

Possiamo affrontare una parte del problema come una trasformazione: dato un insieme di libri e un titolo, otteniamo i libri corrispondenti. Una *funzione pura* restituisce lo stesso risultato per gli stessi argomenti e non produce effetti osservabili all'esterno della funzione. È una proprietà utile perché rende più facile ragionare sul risultato. Non tutti i compiti possono essere puri: registrare un prestito cambia lo stato della biblioteca e, prima o poi, deve produrre un effetto.

Java mette a disposizione strumenti funzionali che incontreremo più avanti, in particolare quando tratteremo le funzioni come valori e lavoreremo con le collezioni. Per ora basta una distinzione: scegliere una funzione pura per una trasformazione di dati può essere opportuno; scegliere un oggetto per rappresentare un prestito con identità e durata può esserlo altrettanto. Le prestazioni e la leggibilità dipendono dal programma concreto, non dall'etichetta del paradigma.

Proviamo a formulare due domande diverse. «Quali libri del catalogo contengono la parola *mare* nel titolo?» è una domanda su dati già disponibili. Possiamo darle una funzione che riceve il catalogo e la parola cercata, e restituisce un risultato senza cambiare il catalogo. Se la richiamiamo con gli stessi dati, ci aspettiamo lo stesso risultato. «Presta questo esemplare al lettore» è invece un comando: dopo averlo eseguito, la disponibilità dell'esemplare cambia. Cercare di descrivere entrambe le operazioni come se fossero prive di effetti non chiarisce il problema.

Immutabilità e ricorsione possono comportare costi e rendere un programma difficile da leggere. È una possibilità, non un destino della programmazione funzionale. Un algoritmo può usare dati immutabili con strutture che condividono parti interne, oppure può essere espresso senza una ricorsione profonda. L'unico modo serio di confrontare due soluzioni per costo e chiarezza è guardare l'algoritmo, le strutture impiegate e il carico reale. Nei prossimi capitoli useremo oggetti e funzioni insieme quando ciascuno aiuta a raccontare una parte del problema.

Guardare il lavoro come una collaborazione fra oggetti

Nella nostra biblioteca riconosciamo almeno un `Libro`, un `Lettore` e un `Prestito`. Un libro ha informazioni, per esempio titolo e autore. Un prestito collega un lettore a un libro e conosce quando è iniziato. La biblioteca offre operazioni per cercare e registrare. Queste entità non sono copie automatiche degli oggetti fisici: sono scelte di modello. Se il programma deve soltanto stampare un catalogo, forse non serve rappresentare ogni prestito. Se deve controllare restituzioni e scadenze, quel concetto diventa importante.

Definizione – Oggetto. Un oggetto è un'entità del programma con un'identità nel modello, uno stato osservabile attraverso le operazioni previste e un comportamento. In Java molti oggetti sono istanze di classi. Non tutto ciò che il programma usa è un oggetto: il linguaggio ha anche tipi primitivi, che incontreremo nel capitolo sulla sintassi.

Chi decide le operazioni di un oggetto decide anche quali dettagli nascondere. Un `Prestito`, per esempio, può offrire un'operazione per chiuderlo. Se dall'esterno fosse possibile cambiare liberamente la data di restituzione senza rispettare le regole, il programma potrebbe rappresentare una situazione impossibile. Un nome significativo aiuta a leggere il codice, ma da solo non lo rende corretto o *autodocumentante*: servono responsabilità chiare, contratti comprensibili ed esempi che mostrino i limiti.

Seguiamo una richiesta completa. `Lettore` identifica chi chiede il volume; `Catalogo` trova l'esemplare; un'operazione della `Biblioteca` verifica che quell'esemplare sia disponibile e crea un `Prestito`. Infine la ricevuta legge i dati del prestito. Abbiamo costruito una sequenza di collaborazioni, non un elenco di sostantivi da trasformare automaticamente in classi. «Disponibile» potrebbe essere uno stato conservato nell'esemplare oppure una risposta calcolata dalle registrazioni: la scelta dipende da come il sistema dovrà mantenere coerenti i dati. Quando aggiungeremo le classi vere, questa domanda determinerà metodi e controlli.

Concetto chiave – Responsabilità e messaggi. Dire che un oggetto «chiede» qualcosa a un altro è un modo leggibile di descrivere una chiamata a un'operazione. Chi chiama conosce il contratto dell'operazione, non necessariamente i passi interni. Se il `Catalogo` offre `cercaPerTitolo`, la `Biblioteca` deve sapere quali dati fornire e che tipo di risposta aspettarsi; non deve conoscere l'algoritmo con cui il catalogo effettua la ricerca.

Riferimento essenziale – Una fotografia storica del paradigma. La ricerca di David E. Monarchi e Gretchen I. Puhr, *A Research Typology for Object-Oriented Analysis and Design*, pubblicata nel 1992, offre un quadro storico dell’analisi e della progettazione a oggetti. Quel quadro è utile per capire quali domande ponevano metodi e strumenti dell'epoca; non è una specifica di Java e non rende obbligatoria ogni caratteristica per ogni programma. Nel nostro esempio possiamo verificare le idee una per una: un `Prestito` conserva una parte dello stato, espone operazioni e collabora con il `Catalogo`; la qualità del modello dipende dalle regole che quelle operazioni proteggono, non dal semplice numero di oggetti.

Il nome di un oggetto nel codice merita una distinzione. In un programma Java possiamo assegnare il nome `prestito` a una variabile, ma quel nome appartiene al riferimento nel codice, non all'oggetto come identità intrinseca: un altro riferimento può indicare la stessa istanza. Anche gerarchie e classi astratte richiedono una scelta di modellazione. Una classe generale può essere dichiarata astratta quando non ha senso creare direttamente un'istanza che soddisfi il suo contratto; non serve rendere astratta ogni classe base. Ereditarietà e messaggi

sono strumenti possibili, mentre nomi chiari e responsabilità leggibili richiedono lavoro di progettazione: gli oggetti non diventano autodocumentanti da soli.

Infine, modellare lo stato della biblioteca non implica che ogni cambiamento del sistema sia rappresentato da un campo modificato in un oggetto. Alcuni risultati possono essere calcolati da dati immutabili; altri compiti, come registrare un prestito, modificano uno stato. Più attività possono agire contemporaneamente e allora occorrono regole di coordinamento, che studieremo nel capitolo sui thread. Java consente anche procedure e funzioni: la scelta utile è quella che rende riconoscibili il problema, la responsabilità della modifica e la prova che la regola resta vera.

Classi di oggetti

Pensa al libro che stai leggendo. Puoi parlare di **un libro** come concetto generale e, nello stesso tempo, distinguere questo esemplare dagli altri. La classe descrive le caratteristiche e le operazioni comuni; ogni oggetto creato secondo quella descrizione possiede il proprio stato. Due libri possono avere lo stesso titolo senza essere lo stesso esemplare. Più avanti vedremo anche che *identità* e *uguaglianza* sono domande diverse: due oggetti distinti possono avere contenuti uguali.

> **Definizione – Classe e istanza.** Una classe è una dichiarazione che descrive un tipo di oggetti e ne può definire stato e comportamento. Un'istanza è un oggetto creato secondo quella classe. Una classe non è «l'unico oggetto generale» da cui nascono copie identiche: è la descrizione che la JVM usa per creare oggetti distinti.

Un modello utile nasce da ciò che il programma deve fare, non da ogni proprietà che un libro possiede nel mondo reale. Se dobbiamo trovare un volume nel catalogo, titolo e autore sono importanti. Il numero di pagine può esserlo per un libro cartaceo; per un libro digitale potrebbe avere un significato diverso. La figura mostra una generalizzazione possibile, non una classificazione valida per ogni biblioteca.

Per evitare una confusione frequente, distinguiamo anche il **titolo** dall'**esemplare**. «Il nome della rosa» è un titolo che può comparire in più copie fisiche. Se una sola copia è in prestito, le altre possono essere disponibili. Rappresentare tutto con un unico oggetto `Libro` dotato di un solo campo `disponibile` potrebbe quindi essere insufficiente. Possiamo descrivere l'opera con un oggetto e ogni copia con un altro, oppure usare un modello diverso se l'applicazione gestisce soltanto titoli digitali. La classe non arriva prima del problema: nasce dalle distinzioni che il problema ci obbliga a fare.

Allo stesso modo, due oggetti che descrivono copie distinte possono avere lo stesso titolo e lo stesso autore. Sono simili per contenuto, ma non sono la stessa copia. Il programma deve sapere quando sta chiedendo «sono la medesima istanza?» e quando invece sta chiedendo «rappresentano lo stesso valore?». Non serve conoscere ora gli operatori Java per fare questa distinzione; ci servirà quando confronteremo oggetti nel codice.

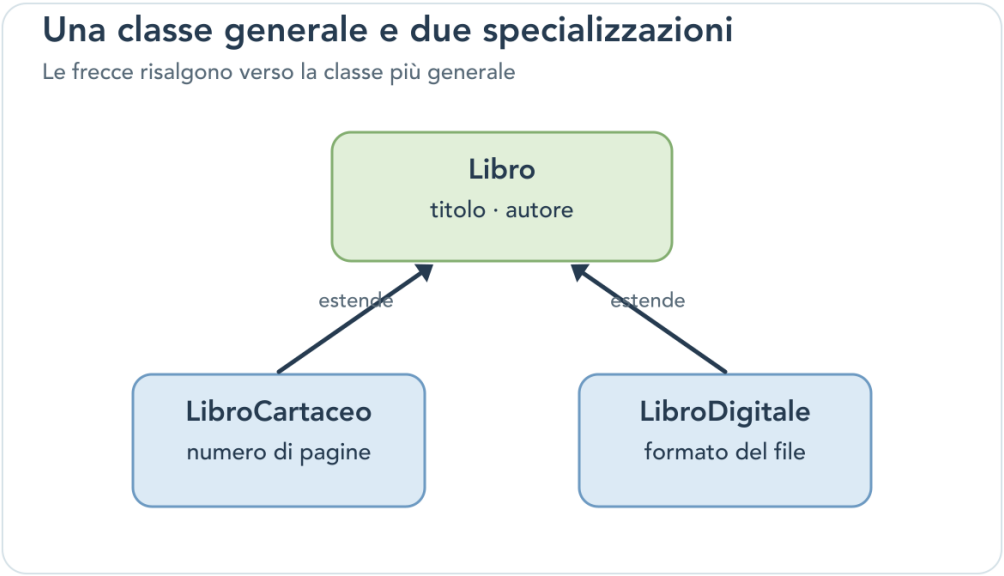


Figura 2.4 – `LibroCartaceo` e `LibroDigitale` sono due possibili specializzazioni di `Libro`. Le frecce puntano verso la classe più generale; le proprietà specifiche restano nella classe che le possiede.

Possiamo classificare i libri anche per argomento: da `Libro` potremmo disegnare rami come storia, fisica, letteratura, scienze e matematica. Questo schema distingue categorie più generali e più specifiche, ma non decide da solo come progettare le classi. Un libro di matematica per la scuola appartiene sia alla matematica sia ai testi scolastici: sono davvero due *tipi* di oggetto da esprimere con due superclassi, oppure due proprietà con cui il catalogo lo classifica? Nel nostro modello di biblioteca il tema e la destinazione scolastica possono essere dati del libro, anche quando la distinzione cartaceo/digitale cambia le operazioni disponibili. Possiamo quindi assegnare più categorie allo stesso esemplare senza inventare una sottoclasse per ogni combinazione. La scelta si verifica chiedendo quali operazioni cambiano: una categoria usata soltanto per cercare libri può restare un dato; una differenza di comportamento richiede invece un contratto da spiegare.

Ereditarietà e polimorfismo

Quando diciamo che `LibroDigitale` **è un** `Libro`, stiamo proponendo una relazione di specializzazione. In Java una classe può estendere un'altra classe e acquisire le caratteristiche accessibili della **classe base**, detta anche *superclasse*. La **sottoclasse** aggiunge o modifica ciò che rende specifica la variante. Useremo questi due nomi in modo coerente: «classe base» per la parte generale e «sottoclasse» per quella specializzata.

> **Definizione – Polimorfismo.** Il codice può usare un riferimento di tipo generale e incontrare oggetti di tipi più specifici. Quando chiama un metodo ridefinito, viene eseguita l'implementazione appropriata all'oggetto effettivo. Non significa che un oggetto cambi classe durante l'esecuzione: significa che un'operazione comune può avere comportamenti specifici.

Supponiamo che ogni `Libro` sappia produrre una breve descrizione. Il catalogo può chiedere la descrizione a un libro senza conoscere in anticipo se sia cartaceo o digitale; l'oggetto specifico fornisce il comportamento previsto. È un vantaggio reale quando la relazione fra i tipi è stabile e le regole sono chiare. Non è una scorciatoia automatica per riusare codice: una gerarchia sbagliata rende più difficile cambiare il programma.

I mezzi di trasporto ci aiutano a isolare il meccanismo. Un treno, un aereo e un'automobile possono tutti rispondere alla richiesta «descrivi come ti sposti». La richiesta è comune, la risposta cambia secondo il mezzo concreto. Il codice che raccoglie le descrizioni può lavorare attraverso un tipo generale. Questo non vuol dire che i tre mezzi siano intercambiabili in ogni operazione: soltanto le promesse contenute nel contratto comune valgono per tutti. Un'operazione «apri le ali» non appartiene a quel contratto.

In Java il riferimento attraverso cui usiamo un oggetto ha un tipo dichiarato, mentre l'oggetto creato ha un tipo effettivo. Se il metodo chiamato può essere ridefinito, la scelta dell'implementazione avviene in base all'oggetto effettivo. È il *binding dinamico* che rende concreto il polimorfismo in questo caso. La differenza tra tipo del riferimento e tipo dell'oggetto merita esempi di codice: la riprenderemo quando sapremo leggere dichiarazioni e metodi.

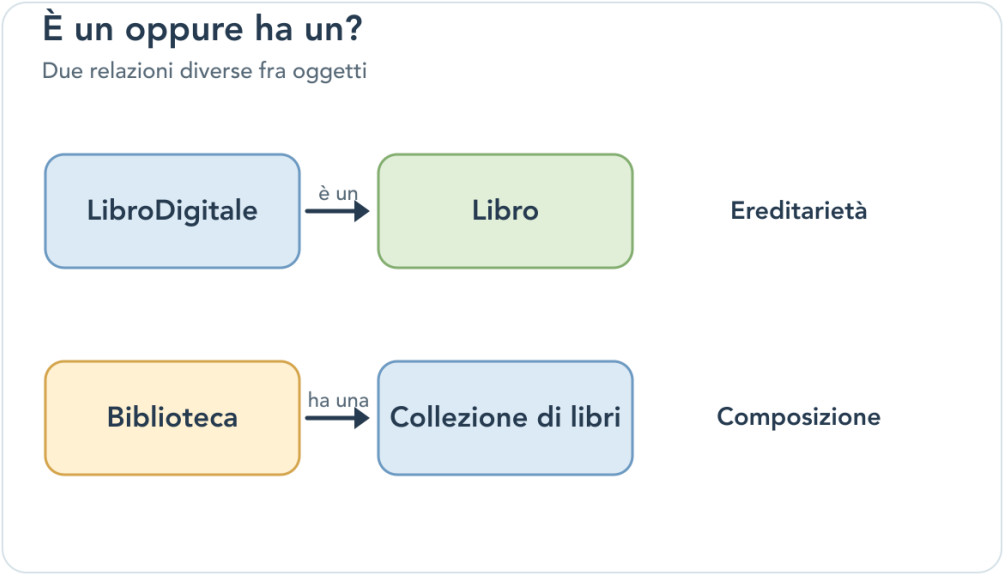


Figura 2.5 – «È un» e «ha una» portano a scelte diverse. `LibroDigitale` *può essere un tipo di* `Libro`*; una* `Biblioteca` *ha una collezione di libri, ma non è una collezione di libri.*

Un libro di matematica può appartenere a più categorie del catalogo. Se però usiamo frecce di ereditarietà per rappresentare queste categorie, rischiamo di confondere la classificazione dei libri con le regole delle classi Java. Java consente a una classe di estendere **una sola classe**, mentre può implementare più interfacce. Un'etichetta di catalogo può anche essere un semplice dato. Prima di disegnare le frecce chiediamo quindi quale contratto comune serve al codice e quali nomi descrivono soltanto modi diversi di trovare un libro.

La seconda relazione della figura è la *composizione*: un oggetto contiene o usa altri oggetti per svolgere il proprio lavoro. Quando una proprietà può cambiare indipendentemente dall'identità dell'oggetto, o quando vogliamo sostituire un comportamento senza inventare una parentela, la composizione è spesso più semplice. Nel seguito del libro torneremo sul confronto con esempi di codice.

L'ereditarietà permette anche di riusare implementazioni, ma il riuso da solo non basta a giustificare una gerarchia. Se `LibroDigitale` estende `Libro`, eredita ciò che la classe base rende disponibile e può aggiungere comportamenti specifici. Se però `Libro` promette che ogni esemplare ha pagine fisiche da sfogliare, il libro digitale non rispetta quella promessa: abbiamo scelto una generalizzazione troppo stretta. Conviene correggere il concetto comune oppure comporre l'oggetto con servizi distinti, invece di costringere una sottoclasse a simulare operazioni che non possiede.

> **Nota bene – Una modifica alla classe base si propaga davvero.** Cambiare un metodo della superclasse può cambiare il comportamento di molte sottoclassi; non è automaticamente un vantaggio di manutenzione. Chi modifica una gerarchia deve verificare anche i contratti delle classi che la estendono. La promessa «basta cambiare una classe e tutto il resto resta corretto» non può essere fatta senza prove.

Incapsulamento e stato valido

Un oggetto non è un sacchetto di variabili liberamente modificabili. Se rappresenta un prestito, dovrebbe impedire combinazioni prive di senso: una restituzione precedente all'inizio, oppure una chiusura registrata due volte senza una regola. Nascondere i campi è un passo utile, ma non basta. Anche un metodo pubblico può rompere lo stato se permette qualunque valore. L'incapsulamento funziona quando le operazioni esposte proteggono le *invarianti*, cioè le condizioni che devono restare vere mentre l'oggetto viene usato.

> **Concetto chiave – Nascondere non significa garantire.** Una classe con campi privati può comunque avere un comportamento sbagliato. I metodi devono controllare gli argomenti e mantenere le regole dell'oggetto; i test devono verificare anche i casi che potrebbero violarle.

Una pila di dati rende concreti `push` e `pop` e rende visibile che cosa l'incapsulamento deve difendere. Una pila conserva elementi in un ordine preciso: **l'ultimo inserito è il primo estratto**. La chiamata `push("Ada")` inserisce un nome; `push("Grace")` colloca il secondo sopra il primo. A questo punto il contenuto, letto dalla cima, è:

Posizione nella pila	Valore dopo i due inserimenti
Cima: prossimo `pop`	`"Grace"`
Sotto la cima	`"Ada"`

Una prima chiamata a `pop()` restituisce `"Grace"` e lascia `"Ada"` in cima; una seconda restituisce `"Ada"`. Se chi usa la pila potesse spostare direttamente i due elementi nel contenitore interno, la regola non sarebbe più garantita. Se chiama `pop()` una terza volta, però, non basta che il contenitore sia nascosto: il contratto dell'operazione deve dire che cosa succede a pila vuota. Può segnalare un errore o scegliere un'altra risposta dichiarata, ma non deve far indovinare al chiamante un comportamento accidentale. La tabella rende esplicito il comportamento della pila; un'implementazione completa della struttura dati potrà approfondire le scelte senza lasciare incompleto l'esempio di incapsulamento.

Un'interfaccia pubblica ben progettata permette spesso di cambiare la rappresentazione interna senza cambiare chi usa l'oggetto. Non promette però che ogni modifica resti locale: se cambiano il significato delle operazioni, i tempi di risposta o le eccezioni, gli altri componenti possono risentirne. Per questo gli oggetti richiedono attenzione alle dipendenze e verifiche, proprio come i moduli procedurali.

La pila ci mostra anche la differenza fra una promessa sull'**ordine** e una promessa sulla **capacità**. Una pila con spazio fisso può essere piena; una che cresce può incontrare comunque limiti di risorse. Le operazioni pubbliche devono rendere riconoscibili questi casi, e la rappresentazione interna deve sostenere il contratto scelto. Più avanti costruiremo una pila; il modello che serve a capire l'incapsulamento è già qui, con il caso normale e quello vuoto.

Un'altra regola utile è distinguere **visibilità** e **consistenza**. Rendere un campo `private` limita chi può raggiungerlo direttamente; non assicura che il valore sia valido. Una funzione pubblica `chiudiPrestito` che accetti qualunque data potrebbe comunque registrare una restituzione precedente al prestito. L'oggetto deve controllare la data al confine dell'operazione e decidere come comunicare il rifiuto. Questa è la parte dell'incapsulamento che protegge il significato, oltre alla memoria.

Alcune buone regole per cominciare

Un oggetto dovrebbe avere una responsabilità che il lettore sappia descrivere in una frase. Il suo stato dovrebbe restare valido dalla creazione e dopo ogni operazione. Prima di scegliere l'ereditarietà, chiediti se la frase «è un» ha davvero senso e se il comportamento della sottoclasse rispetta le attese di chi usa la classe base. Un televisore e una lampadina possono entrambi essere accesi, ma non per questo il televisore è una lampadina: possono condividere un'operazione senza appartenere alla stessa famiglia.

Torniamo al prestito. Se affidiamo a un unico oggetto `Biblioteca` la ricerca dei libri, il controllo delle copie, il calendario delle scadenze e la stampa delle ricevute, ogni modifica sembra richiedere di entrare nella stessa classe. Per esempio, cambiare il formato della ricevuta potrebbe costringerci a rileggere la regola che impedisce di prestare una copia già occupata. Possiamo dare al `Catalogo` la

responsabilità di trovare una copia, al `Prestito` quella di rappresentare la durata e l'esito, e a un componente di presentazione quella di produrre la ricevuta. La `Biblioteca` coordina la richiesta senza custodire per forza tutti i dettagli. La divisione funziona se sappiamo dire quale oggetto conserva l'invariante «la stessa copia non può essere prestata due volte nello stesso periodo»: spezzare una classe grande in tre classi che si modificano liberamente a vicenda non risolverebbe il problema.

Prova a cambiare il requisito: la biblioteca consente una prenotazione quando tutte le copie sono occupate. Una prenotazione è una specie di libro, un comportamento del lettore o un nuovo concetto che collega lettore, opera e posizione in attesa? Disegna due alternative e segui una richiesta fino alla risposta. La soluzione migliore è quella in cui puoi individuare chi decide se la prenotazione è valida e chi aggiorna la disponibilità quando una copia torna libera. Questo esercizio verifica la regola delle responsabilità su un caso nuovo; non si risolve contando le classi o ripetendo l'espressione «è un».

Queste regole non sostituiscono la pratica. All'inizio è normale modellare troppi oggetti o dare a uno solo tutte le responsabilità. Il rimedio è verificare un caso d'uso concreto, cambiare il modello quando crea attrito e tenere il codice abbastanza piccolo da poterlo correggere.

È tutto oro quello che luccica?

No. Il linguaggio offre classi, metodi, interfacce ed ereditarietà; il programmatore sceglie come usarli. Si può organizzare un programma in modo simile agli oggetti anche in un linguaggio che non possiede classi, passando esplicitamente una struttura dati alle funzioni che la gestiscono. Immagina in C una struttura `Libro` e funzioni come `presta(Libro *esemplare, Lettore *lettore)`: i dati e le operazioni collaborano attorno allo stesso concetto, ma il legame deve essere espresso dal programmatore. I prototipi devono dichiarare gli stessi parametri usati nelle definizioni, e la memoria associata ai nomi richiede una gestione esplicita. Questi dettagli fanno parte della correttezza dell'esempio, non sono una differenza cosmetica rispetto alla sintassi Java.

Allo stesso modo, si può scrivere Java pieno di classi ma difficile da leggere. Un nome elegante non salva una responsabilità confusa. Un oggetto che nasconde ogni dato senza spiegare che cosa fa non aiuta il lettore del codice. L'obiettivo è un modello che renda esplicite le regole del problema e consenta di verificarle.

> **Approfondimento – Un po' di storia.** Simula rese pratici i concetti di classe e oggetto per descrivere sistemi da simulare. Alan Kay diede grande rilievo all'idea di entità che collaborano scambiandosi messaggi; in Smalltalk questo modo di pensare divenne centrale. La diffusione delle interfacce grafiche e dei calcolatori più potenti rese interessanti modelli capaci di tenere insieme stato e comportamento. Negli anni Ottanta il nome *object oriented* si diffuse fra linguaggi e strumenti anche molto diversi, alimentando un dibattito su che cosa fosse essenziale: messaggi, incapsulamento, collegamento dinamico delle chiamate o ereditarietà. Questa storia ci ricorda che una parola condivisa non garantisce un identico modello di programmazione.

Un oggetto senza la parola `class`

Una struttura C accompagnata da funzioni che ricevono un puntatore alla struttura mostra che **organizzare dati e operazioni attorno a un concetto** è una scelta di progetto, anche quando il linguaggio non offre la sintassi delle classi. Se omettiamo il parametro necessario nella dichiarazione delle funzioni o lasciamo incompleta la gestione della memoria associata al nome, il programma non è verificabile. Conviene rendere espliciti entrambi gli aspetti prima di usarlo come esempio.

Il confronto importante si può già fare senza compilare C. Nel modello con funzioni, una chiamata del tipo `presta(esemplare, lettore)` indica esplicitamente quale struttura sarà modificata. In Java una chiamata come `esemplare.prestaA(lettore)` fa comparire l'esemplare a sinistra del punto: il metodo viene invocato su quell'oggetto e riceve gli altri dati necessari. La forma sintattica cambia, ma in entrambi i casi dobbiamo decidere chi possiede la regola del prestito. Java ci dà strumenti per rappresentarla e limitarne l'accesso; non la progetta al posto nostro.

Come si è arrivati fin qui

Nella storia del paradigma a oggetti ricorrono Simula, Smalltalk e il lavoro di Alan Kay. Simula rese centrali classi e oggetti per descrivere sistemi; Smalltalk diede grande importanza agli oggetti che comunicano mediante messaggi. Il lettore può incontrare l'espressione *invio di messaggi* anche quando, in Java, scriviamo più concretamente *chiamata di metodo*. Non si tratta di due istruzioni Java diverse: è una differenza nel modo di raccontare la collaborazione fra entità.

Negli anni successivi linguaggi e strumenti hanno dato pesi diversi a ereditarietà, incapsulamento, interfacce e collegamento dinamico delle chiamate. Per questo non serve cercare una definizione storica che renda identici tutti i linguaggi «a oggetti». Per lavorare in Java useremo una domanda più utile: quali dati e operazioni appartengono a un concetto, che cosa promette agli altri e come manteniamo quella promessa quando il programma cambia? La popolarità di un paradigma non garantisce da sola risposte buone; gli esempi e le verifiche sì.

Mettiamo alla prova il modello

Immagina che la biblioteca debba ricordare la data di restituzione di ogni prestito. Aggiungeresti la data al `Libro`, al `Lettore` o al `Prestito`? Motiva la scelta dicendo a quale evento appartiene quel dato e quali operazioni possono modificarlo. Poi prova un secondo caso: la biblioteca deve sapere quanti libri sono disponibili. Conviene conservare un numero modificabile da chiunque, oppure ricavarlo o aggiornarlo attraverso un'operazione controllata? Non c'è bisogno di scrivere Java per rispondere; serve rendere visibile la responsabilità.

L'attività è riuscita se il tuo modello distingue almeno `Libro`, `Lettore` e `Prestito`, indica chi conosce la disponibilità e impedisce uno stato impossibile. Se due responsabilità sembrano plausibili, descrivi il caso d'uso che ne decide la scelta: ci servirà quando costruiremo le classi vere.

3. Il linguaggio Java e gli strumenti per usarlo

Introduzione

Nel capitolo precedente abbiamo parlato di programmi come collaborazioni fra oggetti. È arrivato il momento di vedere che cosa serve per trasformare una descrizione scritta da noi in qualcosa che il computer possa eseguire. Qui incontreremo il linguaggio Java, il compilatore, la Java Virtual Machine e il JDK. Sono nomi vicini, ma indicano cose diverse. Se li distingui subito, molti messaggi del terminale diventeranno meno misteriosi.

Il linguaggio Java stabilisce come scrivere dichiarazioni ed espressioni e che cosa significano. **Java SE** indica una piattaforma standard con linguaggio, macchina virtuale e API. Il **JDK**, *Java Development Kit*, è una distribuzione degli strumenti necessari per sviluppare: contiene, fra l'altro, il compilatore `javac` e il comando `java` per avviare un programma. La **JVM**, *Java Virtual Machine*, è l'ambiente che esegue le istruzioni contenute nei file di classe. Puoi pensare al JDK come alla cassetta degli attrezzi; la JVM è una parte essenziale dell'ambiente di esecuzione, non il nome del linguaggio.

Potresti incontrare anche la sigla **JRE**, *Java Runtime Environment*. Indica un ambiente pensato per eseguire programmi Java senza gli strumenti completi di sviluppo. La distinzione concettuale resta utile: compilare richiede un compilatore, eseguire richiede un ambiente capace di avviare la JVM e di fornire le API necessarie. Non dedurne però che, oggi, ogni sistema debba installare prima un «JRE separato» e poi un JDK. Formati di distribuzione e pacchetti disponibili dipendono dal fornitore. Per seguire gli esempi del libro scegli un JDK completo e verifica la presenza sia di `java` sia di `javac`; quando distribuirai un'applicazione, deciderai quale runtime fornire in base al progetto.

Queste quattro parole rispondono dunque a domande diverse. «Quale sintassi posso scrivere?» riguarda il linguaggio e la versione selezionata. «Quali classi della libreria posso usare?» riguarda le API disponibili alla compilazione e all'esecuzione. «Con quale comando trasformo il sorgente?» riguarda gli strumenti del JDK. «Che cosa interpreta o compila il bytecode mentre il programma gira?» riguarda l'implementazione della JVM. Se un esempio non funziona, separare le domande ci impedisce di cercare la risposta nella scatola sbagliata.

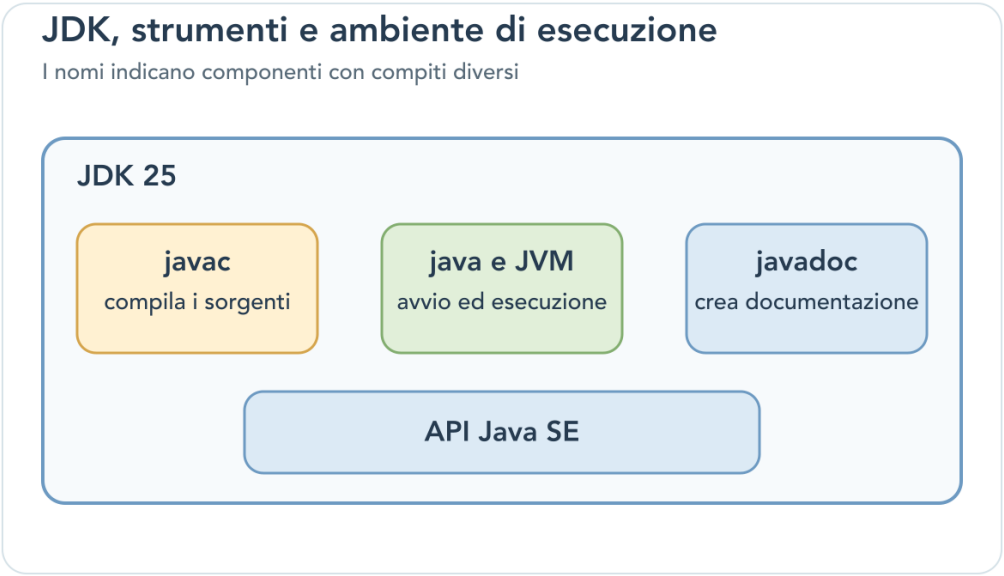


Figura 3.1 – Il JDK raccoglie strumenti con scopi diversi. `javac` *compila,* `java` *avvia il programma nella JVM,* `javadoc` *crea documentazione e le API standard forniscono classi da usare nel codice.*

> **Definizione – API.** Un'API è l'insieme di operazioni e contratti che un componente mette a disposizione di altri componenti. Le API standard di Java includono, per esempio, classi per stringhe, collezioni, file e comunicazione in rete. Impareremo a usarle quando avremo i concetti necessari per capirne i metodi.

Da Oak a Java 25

Il linguaggio nacque nel contesto di Sun Microsystems con il nome Oak e fu poi presentato come Java. In seguito il progetto e le sue implementazioni hanno avuto molti cambiamenti: nuove funzioni, nuove API, modifiche al modello di distribuzione e un ritmo di rilascio più regolare. Date, tabelle di supporto e licenze commerciali sono importanti quando devi scegliere un JDK per un'organizzazione, ma possono cambiare durante la vita del libro. Per una decisione di acquisto o distribuzione, consulta sempre le condizioni della distribuzione scelta e la relativa documentazione aggiornata.

Questa edizione usa Java 25 come riferimento. Non significa che il codice di Java 8 sia diventato improvvisamente illeggibile. Significa che, quando esistono più modi stabili di risolvere un problema, spiegheremo quello appropriato al linguaggio attuale e segnaleremo il requisito di versione se conta. La specifica della piattaforma Java 25 e il manuale di migrazione del JDK sono i riferimenti per distinguere comportamento corrente e compatibilità storica.

> **Nota bene – Distribuzione e linguaggio.** Diversi fornitori possono distribuire un JDK conforme alla stessa versione di Java. Le licenze, la durata del supporto e i pacchetti disponibili sono questioni da verificare per ciascun prodotto e per la data in cui lo userai. Non si può dedurre una differenza di prestazioni fra due distribuzioni dal solo nome commerciale.

Perché la portabilità era un problema interessante

All'inizio degli anni Novanta il gruppo di Sun che sviluppava Oak pensava anche a dispositivi diversi dai normali computer. Un programma legato alle istruzioni di un solo processore avrebbe richiesto molto lavoro per essere trasferito su un altro apparecchio. Il progetto cambiò direzione mentre cresceva il Web, e Java fu presentato pubblicamente nel 1995. Se ti interessa la storia, vale la pena ricordare questa motivazione: la portabilità non nacque come uno slogan pubblicitario appeso a un linguaggio già finito, ma come risposta alla varietà dei dispositivi e degli ambienti di esecuzione.

Il motto «scrivi una volta, esegui ovunque» esprime bene l'obiettivo, purché lo leggiamo con attenzione. Compilare in bytecode evita di produrre ogni volta istruzioni native per un singolo processore. Occorre però un ambiente compatibile sulla macchina di destinazione, e l'applicazione deve usare soltanto risorse disponibili lì. Un percorso di file scritto per un sistema, una libreria nativa o una funzione non presente nel JDK di destinazione possono interrompere la promessa. Per questo nei prossimi esempi distingueremo sempre la **versione del linguaggio**, la **versione delle API** e le **risorse esterne** richieste dal programma.

Le distribuzioni del JDK possono avere licenze e periodi di supporto diversi. Una tabella di scadenze stampata in un libro può invecchiare rapidamente. Prima di scegliere, consulta il contratto e la pagina di supporto del fornitore: il nome della versione Java non basta a stabilire le condizioni d’uso. Per seguire gli esempi, verifica invece quale JDK è installato e quali caratteristiche stabili appartengono a Java 25. Il primo controllo riguarda il prodotto scelto; il secondo la piattaforma su cui compilare ed eseguire il codice.

Che cosa rende riconoscibile Java

Java usa classi e interfacce per descrivere tipi e comportamenti, ma non è corretto dire che «tutto è un oggetto». Un valore `int`, per esempio, appartiene a un tipo primitivo e non è un'istanza di una classe. La differenza diventerà concreta quando assegneremo valori a variabili e quando chiameremo metodi. Una variabile ha un tipo noto al compilatore: questo è il senso operativo di *tipizzazione statica*. Il compilatore può rifiutare alcune combinazioni incompatibili prima che il programma parta, mentre altri errori dipendono da valori conosciuti soltanto durante l'esecuzione.

> **Definizione – Tipo.** Un tipo descrive quali valori può rappresentare una variabile o un'espressione e quali operazioni sono consentite. Java possiede tipi primitivi, tipi riferimento come classi, interfacce e array, e regole precise per conversioni e assegnazioni. `void` indica che un metodo non restituisce un valore; non è una variabile nella quale conservare un dato.

La gestione automatica della memoria libera il programmatore dal rilascio manuale della memoria degli oggetti, ma non chiude per lui ogni risorsa. Java dispone anche di eccezioni per rappresentare condizioni anomale e di strumenti per eseguire più attività in concorrenza. Sono caratteristiche importanti; non rendono però un programma sicuro, corretto o veloce per definizione. Un file lasciato aperto, un'eccezione ignorata o una modifica concorrente senza coordinamento restano problemi del programma, anche se il linguaggio offre strumenti per affrontarli.

Le librerie standard completano il linguaggio: stringhe, collezioni, file, rete e molte altre funzioni arrivano come API documentate. Strumenti esterni, come sistemi di build e librerie di terzi, appartengono all'ecosistema e seguono cicli di rilascio propri. Maven è utile nell'ecosistema Java, ma non è una parola chiave del linguaggio né un componente obbligatorio di ogni JDK. Lo incontreremo quando un programma avrà abbastanza file e dipendenze da giustificare uno strumento di progetto.

Esaminiamo le caratteristiche di Java a partire dalle domande a cui rispondono, perché ogni caratteristica risponde a una domanda che si presenterà durante il libro. La sintassi ci permette di esprimere istruzioni e dichiarazioni; le classi e le interfacce ci aiutano a definire responsabilità e contratti; il sistema dei tipi permette al compilatore di riconoscere molti usi incompatibili prima dell'avvio. Sono strumenti collegati, ma nessuno sostituisce gli altri. Una classe può essere sintatticamente valida e avere responsabilità confuse; una chiamata può rispettare i tipi ed essere logicamente sbagliata. Per questo non useremo «compila» come sinonimo di «funziona».

Le **eccezioni** danno una forma a condizioni che interrompono il percorso normale di un'operazione. Non significano che Java ripari l'errore da solo: chi progetta l'API deve decidere quali condizioni segnalare e chi la usa deve sapere come reagire. La **gestione della memoria** toglie al programmatore molte operazioni manuali legate agli oggetti, ma non garantisce che ogni risorsa venga chiusa o che il programma non conservi oggetti inutili. Le **API standard** offrono componenti già progettati e documentati; usarli bene richiede comunque di leggere i loro contratti, compresi i casi eccezionali. Sono tre esempi di una promessa concreta, accompagnata dal suo limite.

La **concorrenza** permette a più attività di procedere durante lo stesso intervallo di tempo. Se due attività toccano gli stessi dati, però, l'ordine in cui accadono le operazioni può contare. Non basterà creare due thread per ottenere un programma più rapido o corretto; dovremo distinguere cooperazione, visibilità dei dati e coordinamento. Allo stesso modo, la **sicurezza** non nasce automaticamente dal fatto che un programma gira nella JVM. La piattaforma offre controlli, API e confini, mentre l'applicazione deve gestire input, permessi e dipendenze secondo il proprio contesto. Presentare Java come «sicuro» senza spiegare rispetto a quale rischio preparerebbe il lettore a fidarsi della cosa sbagliata.

La **portabilità**, infine, riguarda il rapporto fra sorgente, bytecode e ambienti di esecuzione compatibili; non rende identici sistemi operativi, file e librerie native. La presenza di una comunità ampia, di documentazione e di strumenti di progetto aiuta a trovare soluzioni e a far crescere un'applicazione, ma appartiene all'**ecosistema** più che alla grammatica del linguaggio. Se confondiamo questi piani, finiamo per attribuire a una parola chiave ciò che dipende da un compilatore, a una JVM ciò che dipende dal sistema operativo o al JDK ciò che fa una libreria

esterna. Nei capitoli successivi torneremo su ciascuno di questi temi con esempi, anziché chiederti di imparare un elenco di aggettivi.

Dalla sorgente al programma in esecuzione

Un file sorgente Java è testo, di solito con estensione `.java`. `javac` lo legge, controlla che le dichiarazioni rispettino il linguaggio e produce uno o più file `.class`. Questi file contengono *bytecode*, istruzioni per la JVM. Il comando `java` avvia una JVM che carica le classi necessarie ed esegue il programma. Compilare ed eseguire sono quindi due passaggi distinti: un programma può compilare correttamente e fallire durante l'esecuzione, oppure non arrivare neppure a essere eseguito perché il compilatore trova un errore.

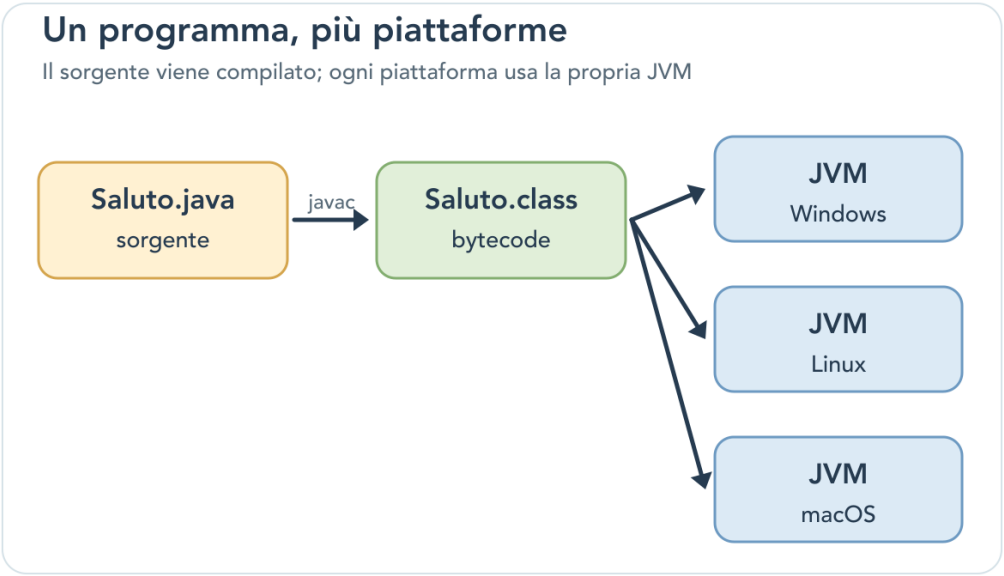


Figura 3.2 – Il bytecode consente a JVM per piattaforme diverse di eseguire il programma. Il file sorgente viene compilato una volta nell'esempio; ciascuna piattaforma dispone del proprio ambiente di esecuzione.

Questa architettura aiuta la portabilità, ma non promette che ogni programma si comporti in modo identico ovunque. Un'applicazione può dipendere da file, permessi, codifica dei dati, librerie native o dettagli del sistema operativo. Se una dipendenza cambia, può cambiare anche il risultato. L'affermazione utile è più precisa: il bytecode conforme può essere eseguito da una JVM compatibile, a condizione che siano disponibili le API e le risorse di cui il programma ha bisogno.

> **Concetto chiave – Tre momenti diversi.** Il compilatore controlla e traduce il sorgente; il caricamento rende disponibili le classi necessarie alla JVM; l'esecuzione produce gli effetti del programma. La compilazione non «carica le classi nella JVM». Torneremo sul caricamento delle classi in un capitolo dedicato.

Fermiamoci su un esempio che useremo fra poco. Scriveremo `Saluto.java` come testo leggibile. Il comando `javac` produrrà `Saluto.class`, che non è un documento

da modificare con l'editor: contiene il risultato della compilazione. Il comando `java` cercherà la classe `Saluto`, avvierà il metodo d'ingresso e farà apparire una riga nel terminale. Se cambi soltanto il testo dentro `Saluto.java`, il file `.class` non cambia da solo. Dovrai ricompilare. Questo piccolo dettaglio è la causa di molti «ho corretto il programma, ma continua a stampare la vecchia frase».

> **Nota bene – Il bytecode non è codice macchina universale.** È un formato di istruzioni per la macchina virtuale Java. Le diverse implementazioni della JVM eseguono quel formato sui sistemi che supportano. La JVM può interpretare istruzioni, compilare parti del programma durante l'esecuzione o combinare più strategie; il file `.class` resta distinto dall'eseguibile nativo prodotto per una singola architettura.

Memoria e caricamento, senza scorciatoie

Java gestisce automaticamente la memoria degli oggetti mediante un *garbage collector*. In termini semplici, il runtime può recuperare memoria di oggetti che non sono più raggiungibili dal programma. Questo evita molte operazioni manuali, ma non elimina ogni problema di memoria: un programma può conservare riferimenti inutili, consumare troppe risorse o dover chiudere esplicitamente file e connessioni. Una coppia di oggetti che si riferiscono a vicenda pone una domanda utile: il ciclo impedisce al garbage collector di recuperarli? Vediamolo con precisione.

Un *class loader* contribuisce a trovare e caricare le definizioni delle classi richieste dal programma. Non significa semplicemente «cercare un `.class` nella cartella corrente»: la ricerca dipende da class path, moduli, loader e configurazione. Qui basta sapere che, dopo aver compilato, il comando di avvio deve poter trovare le classi necessarie. Il capitolo dedicato al caricamento approfondirà delega, collegamento e inizializzazione.

Due oggetti che si riferiscono l'uno all'altro non restano necessariamente in memoria: per i collector Java conta la raggiungibilità dal programma. Immagina `A` che contiene un riferimento a `B` e `B` che ne contiene uno ad `A`: se nessun riferimento raggiungibile dal programma conduce più a nessuno dei due, il ciclo non li rende automaticamente vivi. Viceversa, una lista conservata per errore in un campo ancora raggiungibile può trattenere migliaia di oggetti anche senza alcun ciclo. Non basta quindi contare le frecce fra gli oggetti; occorre chiedersi se esiste un percorso a partire da ciò che il programma può ancora usare.

Situazione	Percorso dal programma	Conseguenza per la comprensione
Una variabile ancora utilizzabile indica `A`; `A` indica `B` e `B` indica `A`.	Esiste: programma → `A` → `B`.	Il ciclo è raggiungibile; entrambi gli oggetti restano utilizzabili attraverso quel percorso.
Nessuna radice raggiungibile conduce ad `A` o `B`, anche se si indicano a vicenda.	Non esiste.	I due riferimenti interni non costituiscono da soli una ragione per conservarli.

La tabella rende visibile la raggiungibilità degli oggetti senza far dipendere il ragionamento da un particolare algoritmo di raccolta. Una *radice* è, in questa spiegazione introduttiva, un punto da cui il runtime può ancora raggiungere oggetti usati dal programma, per esempio riferimenti disponibili durante l'esecuzione. Il momento esatto della raccolta non è promesso: «può essere recuperato» non significa «sarà recuperato subito». La documentazione Java 25 sulla raggiungibilità dà la definizione formale; quando studieremo la memoria in dettaglio distingueremo anche le diverse aree e le scelte delle implementazioni, ma la domanda decisiva per il ciclo è già risolta qui.

Un file aperto chiarisce un limite diverso. Se il programma smette di usare l'oggetto Java che rappresenta quel file, la memoria dell'oggetto potrà essere recuperata secondo le regole del runtime. La risorsa del sistema operativo, però, ha un proprio ciclo di vita: il programma deve chiuderla nel modo previsto dall'API. Più avanti useremo `try` con risorse proprio per rendere esplicita questa responsabilità. Memoria automatica e gestione delle risorse non sono la stessa promessa.

Anche il caricamento delle classi merita un'immagine corretta. Quando scriviamo `java -cp build Saluto`, indichiamo al comando dove cercare le classi dell'applicazione. Il runtime ha già modo di accedere alle proprie classi della piattaforma; non dobbiamo elencarle una per una in `CLASSPATH`. L'ambiente può caricare altre classi quando servono. Evitiamo perciò di memorizzare una sequenza fissa del tipo «prima tutte le classi di base, poi tutte le librerie, infine tutte le classi utente»: il comportamento reale dipende dalle richieste del programma e dalla configurazione.

Preparare il JDK

Installa una distribuzione di JDK 25 per il tuo sistema operativo seguendo le istruzioni del fornitore scelto. Dopo l'installazione apri un terminale nuovo ed esegui:

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

Il primo comando controlla l'avvio dell'ambiente Java; il secondo controlla la presenza del compilatore. Se `java` funziona ma `javac` non viene trovato, potresti avere un ambiente di sola esecuzione oppure un percorso dei comandi incompleto. Se entrambi mancano, verifica l'installazione e la variabile d'ambiente usata dal sistema per cercare gli eseguibili. Il messaggio esatto dipende dal sistema operativo: per questo le vecchie schermate di un gestore di pacchetti non sono una guida affidabile per tutti i lettori.

> **Buona pratica – Registrare l'ambiente.** Quando un esempio non dà il risultato previsto, annota la versione mostrata dai due comandi e il comando che hai eseguito. Sono i primi dati utili per riprodurre il problema. Una schermata senza il comando completo o senza la versione è molto meno utile.

Trovare il programma che il terminale sta usando

Se sul computer sono presenti più JDK, il comando `java --version` rivela quale eseguibile viene scelto in quel terminale, non necessariamente quale JDK hai installato per ultimo. La ricerca passa attraverso il percorso configurato nell'ambiente. Su macOS e Linux puoi usare `command -v java` e `command -v javac` per vedere i percorsi risolti; su Windows puoi usare `where java` e `where javac`. Se i due comandi puntano a installazioni diverse, la compilazione e l'esecuzione potrebbero usare versioni non allineate. Prima di modificare variabili d'ambiente alla cieca, annota entrambi i percorsi e confrontali con la directory del JDK che intendi utilizzare.

Installare tramite il gestore dei pacchetti del sistema può essere comodo, e un pacchetto JDK deve includere `javac`. I nomi dei pacchetti e le versioni disponibili cambiano fra distribuzioni e nel tempo. Per questo un lettore Fedora, Ubuntu, Windows o macOS deve partire dalla documentazione del proprio fornitore e poi fare lo stesso controllo verificabile: `java --version`, `javac --version`, compilazione di un file e avvio del risultato. La conferma non è «il programma di installazione ha finito», ma «riesco a compilare ed eseguire».

Il JDK comprende altri strumenti. `jshell` permette di provare piccoli frammenti senza preparare ogni volta un progetto; `javadoc` genera documentazione da commenti strutturati; `jar` raccoglie classi e risorse in un archivio; `javap` aiuta a osservare la struttura di un file di classe. Li useremo quando risponderanno a una domanda concreta. Per ora bastano `javac` e `java`.

Le opzioni di `javac` che useremo davvero

Il manuale ufficiale del compilatore distingue i file sorgente, la destinazione dei file compilati e i luoghi nei quali cercare classi già disponibili. Se non specifichi `-d`, il file `.class` viene normalmente scritto accanto al sorgente. Noi scegliamo `-d build` per tenere separati i file che modifichiamo da quelli generati. È una distinzione pratica: quando una prova non funziona, sappiamo che cosa cancellare e ricostruire senza toccare il sorgente.

Con `-cp` o `--class-path` indichiamo dove cercare classi e librerie necessarie; `--source-path` riguarda la ricerca di altri sorgenti. Nei nostri primi esempi tutto sta in una cartella e queste opzioni sono poco visibili. Quando organizzeremo il codice in package, il nome del package corrisponderà a una gerarchia di directory e la destinazione indicata da `-d` ne rispecchierà la struttura. Se il compilatore non trova una classe, la domanda giusta non è subito «Java è rotto?», ma «dove dovrebbe cercare quella classe, e con quale nome completo?».

Se dobbiamo compilare molti sorgenti, possiamo passarli tutti al comando oppure scrivere i nomi in un file di argomenti e anteporre `@` al nome del file. Per esempio, un file `sorgenti.txt` che contiene `Saluto.java` e `Argomenti.java`, una riga per nome, può essere passato come `javac --release 25 -d build @sorgenti.txt`. Il carattere `@` dice a `javac` di leggere gli argomenti da quel file; non fa parte del nome di una classe. Questo meccanismo evita righe di comando molto lunghe.

Leggiamo il comando dall'inizio, senza saltare un pezzo. `javac` è il programma che stiamo avviando. `--release 25` chiede di compilare per la release indicata, controllando anche le API ammesse per quel livello; non installa Java 25 su un altro computer. `-d build` indica la radice sotto cui scrivere i file compilati: se il sorgente

dichiara un package, il compilatore crea sotto quella radice la gerarchia corrispondente. `@sorgenti.txt` fornisce la lista dei file da trattare. Se `sorgenti.txt` non si trova nella cartella da cui esegui il comando, il problema non è nella sintassi di una classe Java: è nel percorso dell'argomento passato a `javac`.

L'opzione `--source-path` risponde a una domanda diversa da `-d`. La prima indica dove cercare ulteriori sorgenti; la seconda indica dove scrivere le classi prodotte. L'opzione `-cp` riguarda invece classi e archivi già disponibili come dipendenze. In un progetto con una sola classe non percepiamo ancora la differenza, ma quando un file usa un altro tipo vedremo perché il compilatore deve saperlo trovare. Il class path usato per **compilare** e quello usato per **avviare** un programma vanno entrambi considerati: riuscire a compilare contro una libreria non garantisce che quella libreria sia disponibile quando il programma parte.

> **Nota bene – Dove guardare dopo un errore.** Se manca un sorgente, controlla la cartella di lavoro e il nome scritto nel comando. Se manca un tipo durante la compilazione, controlla package, visibilità, source path e class path. Se la compilazione riesce ma l'avvio non trova una classe, controlla il class path passato a `java` e il nome qualificato richiesto. Scrivere il messaggio esatto e il comando eseguito accanto al problema è molto più efficace che cambiare più opzioni alla cieca.

La prima applicazione

Ho scritto il classico «Hello world» tante volte, in tanti linguaggi, che voglio spezzare il rito con una prima applicazione da osservare davvero. Il messaggio dice «Ciao, Java 25!», ma la domanda non è se abbiamo saputo copiare una frase. Vogliamo capire quali file cambiano quando compiliamo, quale classe viene avviata e come riconoscere un errore in ciascun passaggio.

Creiamo un file chiamato `Saluto.java` con il contenuto seguente. Non è necessario conoscere già ogni parola: subito dopo leggeremo il programma dall'esterno verso l'interno.

```
/** Mostra un saluto e permette di verificare compilazione ed esecuzione. */
public class Saluto {
    public static void main(String[] args) {
        System.out.println("Ciao, Java 25!");
    }
}
```

Il nome della classe pubblica, `Saluto`, corrisponde al nome del file senza `.java`. Le parentesi graffe racchiudono prima la classe e poi il metodo `main`. Il metodo è il punto da cui questa applicazione comincia quando la avviamo. `System.out.println` scrive una riga nel terminale. Le parole `public`, `static` e `void` hanno significati precisi: per adesso trattale come parte della forma di questo punto di ingresso; le riprenderemo quando studieremo metodi e visibilità.

La lunghezza di un sorgente può aiutarci a riconoscere una responsabilità diventata troppo ampia, ma non costituisce una regola di correttezza. Non esiste una soglia del linguaggio, di 500 o 2.000 righe, oltre la quale una classe smette di essere corretta. Se un file cresce tanto da rendere difficili lettura e modifiche,

chiediamoci quali responsabilità vi convivono e se alcune meritano un tipo separato. Il nome del file deve invece rispettare una regola concreta: se contiene una classe pubblica `Saluto`, il sorgente si chiama `Saluto.java`, con le stesse maiuscole. Il sistema operativo può trattare i nomi dei file in modi diversi, ma affidarsi a una differenza di maiuscole invisibile su una macchina crea problemi quando il progetto si sposta.

Nella cartella che contiene il file, esegui:

```
javac --release 25 -d build Saluto.java
java -cp build Saluto
```

Il primo comando chiede a `javac` di compilare per Java 25 e di mettere i file prodotti nella cartella `build`. L'opzione `--release` controlla versione del linguaggio, formato del bytecode e API disponibili per la release indicata; non installa quella versione su un'altra macchina. Il secondo comando indica con `-cp` dove cercare la classe e la avvia usando il nome `Saluto`, senza `.class`. L'output atteso è `Ciao, Java 25!` seguito da un a capo. Il manuale di `javac` documenta le opzioni usate qui.

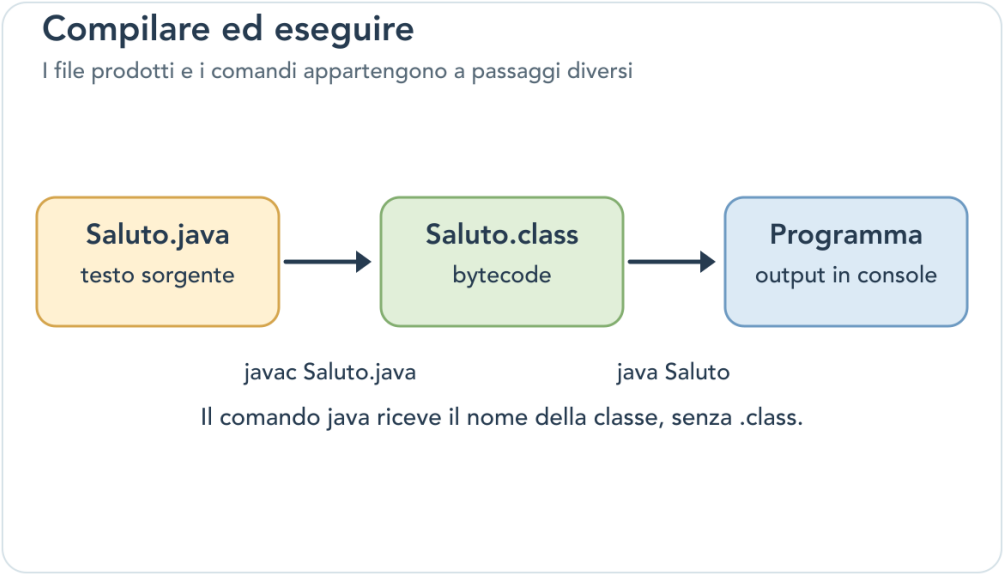


Figura 3.3 – Il compilatore produce bytecode, poi la JVM esegue la classe. Le frecce separano ciò che produce `javac` *da ciò che fa* `java`*.*

Se scrivi `javac Saluto` senza l'estensione del file, il compilatore non riceve il sorgente che intendevi indicare. Se scrivi `java Saluto.class`, il comando di avvio riceve un nome di classe non corretto per questa forma. Un errore di questo tipo è una buona occasione per osservare il messaggio, correggere un solo dettaglio e riprovare.

Anche il **momento** in cui compare l'errore aiuta a capire dove cercare. Se `javac` segnala una parentesi mancante, il programma non è stato ancora avviato: correggi il sorgente e ricompila. Se `javac` termina senza errori ma `java` non trova la classe principale, guarda il nome passato al comando e la radice del class path; cambiare il contenuto di `println` non risolve quel problema. Se invece il

programma parte, stampa una riga e poi mostra un'eccezione, classi e punto di ingresso erano stati trovati: ora bisogna seguire la traccia dell'errore nell'esecuzione. Sono tre diagnosi diverse che nel terminale possono sembrare tutte «Java non funziona».

Facciamo una prova controllata. Compila `Saluto.java` e verifica che `build/Saluto.class` sia stato creato. Cambia il testo del saluto nel sorgente, ma avvia senza ricompilare: vedrai ancora il testo precedente. Ora ricompila e avvia di nuovo. Non è un comportamento capriccioso della JVM: stavi eseguendo il bytecode vecchio, mentre la modifica era rimasta soltanto nel file sorgente. Ripetere questo esperimento una volta è più utile che memorizzare la frase «bisogna compilare» senza aver visto quali file cambiano davvero.

Possiamo rompere la tradizione di iniziare sempre da «Hello world» e chiamare il programma `LaMiaPrimaApplicazione`. Il saluto resta corto per isolare la compilazione: scegli una frase tua e guarda quando cambia l'output. Il nome del file deve seguire esattamente quello della classe pubblica, comprese maiuscole e minuscole. Su alcuni filesystem un errore nel maiuscolo può essere meno evidente che su altri; la regola del linguaggio e degli strumenti, però, non va affidata alla tolleranza del disco.

Argomenti: dal caso minimo al caso completo

L'esempio con `args[0]` ci fa vedere un solo valore. Proviamo ora un programma che funziona con **qualunque numero** di argomenti, anche zero. Il file completo si chiama `TuttiGliArgomenti.java`.

```
/** Mostra nell'ordine tutti gli argomenti ricevuti. */
public class TuttiGliArgomenti {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argomento " + (i + 1) + " = " + args[i]);
        }
    }
}
```

Il ciclo `for` parte da `i = 0`, ripete il corpo finché `i` è minore di `args.length` e aumenta `i` dopo ogni passaggio. `args.length` è il numero di elementi presenti; l'ultimo indice valido è quindi `args.length - 1`. Nell'output usiamo `i + 1` soltanto per numerare gli argomenti come farebbe una persona: internamente il primo elemento resta `args[0]`. Se l'array è vuoto, la condizione `0 < 0` è falsa subito e il programma termina senza stampare nulla, evitando l'errore dell'esempio minimo.

Compila con `javac --release 25 -d build TuttiGliArgomenti.java`, poi esegui `java -cp build TuttiGliArgomenti primo secondo terzo quarto`. Dovresti leggere quattro righe, dalla prima `Argomento 1 = primo` all'ultima `Argomento 4 = quarto`. Prova anche `java -cp build TuttiGliArgomenti` senza parole aggiuntive e spiega perché non compare alcuna riga. Quando studieremo i cicli, torneremo sulla stessa istruzione con maggiore dettaglio.

Il parametro `String[] args` contiene gli argomenti passati dopo il nome della classe sulla riga di comando. Nel nostro programma non li usiamo. Proviamo ora a leggerne uno con un esperimento minimo. Crea `Argomenti.java`:

```
/** Mostra il primo argomento ricevuto dalla riga di comando. */
public class Argomenti {
    public static void main(String[] args) {
        System.out.println("Primo argomento: " + args[0]);
    }
}
```

Compilalo con `javac --release 25 -d build Argomenti.java`, poi esegui `java -cp build Argomenti primo`. Vedrai `Primo argomento: primo`. La parola `primo` dopo il nome della classe arriva al programma come testo. `args` è un *array*, una sequenza di elementi che approfondiremo nel capitolo 4; `[0]` indica il primo elemento, perché gli indici partono da zero. Se avvii questo esempio senza l'argomento, il programma fallisce tentando di leggere un elemento inesistente. Nel capitolo 5 impareremo a controllare `args.length` prima di accedere all'array. La documentazione del comando `java` conferma che gli argomenti scritti dopo il nome della classe sono passati al metodo `main`.

Commenti e documentazione

Un commento `//` occupa il resto della riga. Un commento `/* ... */` può coprire più righe. Il compilatore non li esegue. Servono a spiegare un motivo o una regola che il codice, da solo, non rende evidente. Scrivere `// stampa il saluto` accanto a `println("Ciao")` aggiunge poco; spiegare perché un valore deve rispettare una condizione può evitare un errore futuro.

Prova a leggere questi tre casi come un futuro collega che deve modificare il programma. `// aumenta i` accanto a `i++` ripete ciò che il codice mostra. `// gli indici partono da zero` vicino al primo accesso a un array può invece aiutare un lettore nuovo, ma diventa ridondante quando il concetto è stato imparato. Una nota come `// manteniamo l'ordine originale perché la ricevuta segue l'ordine di inserimento` conserva una **ragione progettuale** che l'istruzione da sola non esprime. I commenti migliori non compensano nomi confusi: prima rendiamo il codice leggibile, poi documentiamo il motivo delle scelte non ovvie.

I tre delimitatori non sono intercambiabili. `//` termina con la riga; `/* ... */` delimita un blocco ordinario; `/** ... */` introduce un commento che lo strumento `javadoc` può associare a una dichiarazione. Un blocco di commento non va usato per «spegnere» temporaneamente codice che contiene già altri commenti a blocco: i delimitatori non si annidano nel modo che potresti aspettarti. Per una prova breve è più chiaro usare il controllo di versione o intervenire direttamente sulle righe interessate.

Il commento che precede `Saluto` comincia con `/**`: è un *documentation comment*. Lo strumento `javadoc` può usarlo per produrre documentazione consultabile. In classi destinate ad altri programmatori, una buona documentazione dice che cosa promette un metodo, che cosa significano i parametri e quali casi eccezionali deve considerare chi lo usa. I tag come `@param` e `@return` aiutano a strutturare l'informazione, ma non devono ripetere il nome del parametro senza spiegare nulla. Il manuale di `javadoc` e la specifica dei commenti chiariscono la forma riconosciuta dal JDK.

Consideriamo un piccolo metodo che calcola il prezzo dopo uno sconto. Un commento utile direbbe se lo sconto è una percentuale fra 0 e 100, come viene

arrotondato il risultato e che cosa accade per un valore fuori intervallo. La frase «calcola il prezzo scontato» è meno utile del nome stesso del metodo. Nel commento strutturato `@param percentuale` dovrebbe descrivere il significato e il dominio del parametro; `@return` dovrebbe descrivere il risultato, non limitarsi a scrivere «ritorna un numero»; `@throws` dovrebbe indicare in quale circostanza un'eccezione può essere lanciata. Sono promesse per chi userà il metodo, e devono restare allineate al codice.

Per leggere la documentazione quotidiana, conviene conoscere questi tag: `@param` descrive un parametro, `@return` il valore restituito, `@throws` una condizione eccezionale, `@see` un riferimento utile e `@since` l'introduzione di una parte dell'API. `@deprecated` accompagna un elemento che non si dovrebbe più scegliere per nuovo codice, ma l'annotazione Java `@Deprecated` ha un ruolo distinto nel segnalare la deprecazione agli strumenti. Non inventare `@return void`: un metodo `void` non ha un valore di ritorno da documentare. Il tag corretto è `@version`, non `@versione`, se decidiamo di usarlo.

Per leggere un commento di documentazione senza trattarlo come una raccolta di formule da copiare, tieni accanto questa tabella. Le descrizioni indicano il **contenuto da spiegare**: il solo tag non aggiunge conoscenza a chi userà il metodo. La specifica Javadoc del JDK 25 chiarisce anche in quali dichiarazioni ciascun tag è ammesso.

Tag	Che cosa deve capire chi legge	Un errore da evitare
`@param nome`	Significato di un parametro, valori ammessi e unità se rilevanti.	Ripetere soltanto «il parametro nome».
`@return`	Significato del risultato e casi particolari.	Scrivere `@return void` per un metodo senza risultato.
`@throws Eccezione`	Condizione in cui il metodo segnala il problema.	Elencare un'eccezione senza dire quando si presenta.
`@see`	Un elemento correlato che aiuta a usare l'API.	Rimandare a una pagina senza spiegare la relazione.
`@since`	Versione in cui l'elemento documentato è stato introdotto.	Confonderla con la versione corrente del progetto.
`@version`	Versione del software documentato, quando ha senso esporla.	Scrivere `@versione`, che non è il tag standard.
`@deprecated`	Motivo per cui l'elemento è superato e alternativa, se esiste.	Limitarsi a «non usare» senza una via praticabile.

La differenza fra `@since` e `@version` sembra piccola finché non aggiorni una libreria. Se una classe è comparsa nella versione 2 del tuo progetto ed è ancora presente nella versione 5, `@since 2` continua a raccontare **quando è arrivata**; `@version 5` può descrivere **la versione del software documentato**. Cambiare

ogni `@since` a ogni rilascio cancellerebbe la storia utile a chi deve sapere da quando può usare quella classe. Il tag `@version`, invece, viene mostrato dal doclet standard quando si richiede la relativa opzione; non bisogna presumere che ogni informazione scritta nel sorgente compaia sempre nelle pagine generate.

Per un metodo che restituisce un `int`, il lettore ha bisogno di sapere se il valore rappresenta euro, centesimi, secondi o un conteggio. Per un metodo che riceve una percentuale, deve sapere se `20` significa venti per cento oppure se si attende `0,20`. Queste sono informazioni che il tipo da solo non esprime. È qui che `@param`, `@return` e, quando una condizione può fallire, `@throws` diventano parte del contratto dell'API. L'esempio del prezzo scontato non è scelto per insegnare la sintassi dei tag: mostra perché documentare il significato di un numero può evitare errori anche quando il programma compila senza alcun avviso.

Puoi generare la documentazione di una classe pubblica con `javadoc -d docs Saluto.java`, dalla cartella che contiene il sorgente. L'opzione `-d docs` indica la destinazione delle pagine prodotte. Apri il file iniziale nella cartella `docs` e confronta il testo con il commento sorgente. Vedrai subito perché un commento incompleto non può diventare una documentazione completa soltanto perché è stato trasformato in HTML.

Il comando `javadoc` non è un correttore automatico delle spiegazioni. Se una frase promette che un metodo restituisce sempre un risultato e il codice può invece lanciare un'eccezione, la pagina HTML ripeterà una promessa sbagliata con una veste più elegante. Lo strumento può segnalare vari problemi strutturali e dispone di controlli come DocLint, ma la correttezza del contratto va letta confrontando commento, implementazione ed esempi. Per una libreria destinata ad altri programmatori questa lettura non è un lusso: la documentazione è spesso il primo punto di contatto con il comportamento del codice.

> **Versione Java – Commenti Javadoc in Markdown.** Il JDK 25 riconosce anche commenti di documentazione composti da righe che iniziano con `///`, oltre alla forma tradizionale `/** ... */`. La specifica del doclet standard ne descrive sintassi e limiti. Nei primi esempi manteniamo la forma tradizionale, già presente in molti sorgenti Java; conoscere l'altra forma aiuta a leggere sorgenti più recenti senza scambiare quelle righe per tre commenti ordinari scollegati.

Che cosa fa HotSpot?

HotSpot è una delle implementazioni della JVM. Durante l'esecuzione può osservare il programma e compilare in codice nativo parti che conviene ottimizzare; per questo si parla di compilazione *just in time*, o JIT. Non cambia il significato del sorgente Java e non implica che ogni programma diventi più veloce in ogni situazione. Prestazioni e consumi si misurano con un carico reale e un ambiente dichiarato. Per imparare a scrivere Java non devi regolare subito il compilatore JIT: è sufficiente distinguere ciò che fa `javac` prima dell'avvio da ciò che può fare una JVM mentre il programma gira.

Prova a immaginare un'applicazione che ripete milioni di volte la stessa elaborazione e un'altra che parte, stampa una riga e termina. Nel primo caso la JVM può avere tempo e dati per individuare parti eseguite spesso e ottimizzarle; nel secondo, il lavoro necessario per avviare e osservare il programma può pesare

più del tempo risparmiato su quella singola riga. «HotSpot» richiama proprio l'attenzione sulle zone calde dell'esecuzione, ma la scelta delle ottimizzazioni è più complessa di un conteggio di chiamate. La JVM può anche rivedere una decisione quando cambiano le condizioni osservate.

La compilazione JIT non fa «uscire» il programma dalla JVM: il codice nativo prodotto durante l'esecuzione rimane gestito dal runtime, che continua a fornire servizi come gestione della memoria, collegamento delle classi ed eccezioni. Anche il confronto assoluto con C++ non regge senza un compito e una misura dichiarati. Per un programmatore che inizia, la regola pratica è semplice: prima costruisci un programma corretto e misurabile; poi, se la velocità è un requisito, osserva dove spende il tempo invece di attribuire ogni differenza alla presenza del bytecode.

Una verifica pratica

Modifica il testo del saluto, ricompila e riesegui. Poi cambia soltanto il sorgente senza ricompilare: che cosa compare? La prova è riuscita se sai spiegare perché il file `.class` conserva il comportamento precedente finché non viene prodotto di nuovo. Come autoverifica, indica quale comando useresti per conoscere la versione del compilatore, quale file è il sorgente e dove si trova il bytecode dopo l'opzione `-d build`.

Se la prova non restituisce ciò che ti aspettavi, non cambiare tre cose contemporaneamente. Comincia con `javac --version`: stai chiamando il compilatore che pensavi di aver installato? Poi ricompila `Saluto.java` con `javac --release 25 -d build Saluto.java` e controlla che `build/Saluto.class` sia stato aggiornato. Infine esegui `java -cp build Saluto`. Il nome dopo `-cp` è una directory da cui cercare la classe, mentre `Saluto` è il nome della classe da avviare; scrivere `Saluto.class` in quella posizione confonde i due ruoli. Se compare ancora il vecchio messaggio, verifica la directory da cui lanci i comandi e cerca eventuali altre copie di `Saluto.class` nel class path usato. La diagnosi segue la stessa catena del disegno: sorgente, compilazione, posizione del bytecode, avvio.

Prova anche a fare un errore intenzionale nel sorgente, per esempio togliendo la virgoletta finale da una stringa. `javac` deve rifiutare il file e segnalare una posizione vicina al problema. Non interpretare un eventuale vecchio `.class` ancora presente come prova che il nuovo sorgente sia corretto: quel file è il risultato dell'ultima compilazione riuscita, non di quella appena fallita. Dopo aver ripristinato la virgoletta, ricompila e confronta l'output. Questo esercizio distingue un errore di compilazione da un errore durante l'esecuzione meglio di una definizione imparata a memoria.

4. Sintassi, variabili, operatori e array

Diamo un nome ai dati

Il primo programma stampava una frase sempre uguale. Un programma interessante, invece, riceve dati, li conserva per il tempo necessario e produce un risultato. Cominciamo con tre voti: `24`, `27` e `30`. Prima di decidere come calcolarne la media dobbiamo capire che cosa sono una variabile, un tipo e un'espressione. Sembrano mattoni piccoli, ma un errore qui si propaga in quasi tutto il codice che verrà.

Un'**espressione** viene valutata e produce un valore, come `primoVoto + secondoVoto`. Un'**istruzione** è un'unità eseguibile del programma: `int somma = primoVoto + secondoVoto;` è una dichiarazione con inizializzazione. Non tutte le istruzioni terminano con `;`: un blocco delimitato da `{` e `}` e un `if` hanno una struttura diversa. Non è neppure vero che un'espressione non possa avere effetti: una chiamata a metodo può produrre un valore e, nello stesso tempo, modificare uno stato. Distingueremo il risultato dagli effetti quando leggeremo il codice.

Java distingue **tipi primitivi** e **tipi di riferimento**. I primitivi comprendono `boolean`, i tipi interi `byte`, `short`, `int`, `long` e `char`, e i tipi a virgola mobile `float` e `double`. Una variabile di tipo primitivo contiene un valore di quel tipo; una variabile di tipo di riferimento contiene un riferimento a un oggetto oppure `null`. Un array è un oggetto, quindi anche `int[]` è un tipo di riferimento, benché i suoi elementi siano interi primitivi. Questa distinzione è descritta dalla specifica dei tipi di Java 25.

La specifica chiama *null type* il tipo speciale dell'espressione `null`: non gli diamo un nome in una dichiarazione come faremmo con `String`, ma possiamo assegnare quel valore a una variabile di tipo riferimento. `null` non indica un oggetto vuoto; indica che quel riferimento non designa un oggetto. Se poi proviamo a chiamare un metodo attraverso di esso, il problema compare a esecuzione avviata. `void` risponde a un'altra domanda ancora: nella firma di un metodo indica che quel metodo non restituisce un valore utilizzabile dal chiamante. Non è un nono tipo primitivo da scegliere per una variabile.

> **Definizione – Tipo e variabile.** Il tipo stabilisce quali valori e operazioni sono ammessi. Una variabile è un nome associato a una posizione che può contenere un valore compatibile con quel tipo. Il nome scelto dal programmatore non cambia il significato delle operazioni, ma può renderlo molto più evidente a chi legge.

La tabella dei tipi primitivi chiarisce due equivoci frequenti: `boolean` non ha una dimensione in memoria fissata a un bit dal linguaggio, e `void` non è un tipo di variabile. Le dimensioni degli interi e dei formati a virgola mobile sono definite dalla specifica, indipendentemente dal sistema operativo.

Tipo	Valori o intervallo	Quando lo incontreremo
`boolean`	`true` oppure `false`	Condizioni di `if` e cicli; non è un numero.
`byte`	Da −128 a 127, intero con segno a 8 bit	Dati binari e API che lavorano su byte.
`short`	Da −32.768 a 32.767, intero con segno a 16 bit	Alcuni formati e API; per i piccoli contatori spesso resta più semplice `int`.
`int`	Da −2.147.483.648 a 2.147.483.647, intero con segno a 32 bit	Indici, conteggi e calcoli interi ordinari.
`long`	Da −9.223.372.036.854.775.808 a 9.223.372.036.854.775.807, intero con segno a 64 bit	Conteggi che possono superare l'intervallo di `int`.
`char`	Da `\u0000` a `\uffff`, unità UTF-16 senza segno a 16 bit	Elaborazione di unità di codice, non di ogni simbolo visibile.
`float`	Numeri nel formato binario IEEE 754 a 32 bit	Dati numerici che accettano questa precisione.
`double`	Numeri nel formato binario IEEE 754 a 64 bit	Calcoli con frazioni quando la rappresentazione approssimata è adatta.

Leggi la tabella come una scelta di significato, non come una gara a usare il tipo più piccolo. Se un contatore arriva al massimo a cento, un `byte` potrebbe contenere il valore, ma molte operazioni aritmetiche su `byte` producono un `int`: il codice richiederebbe conversioni che non aiutano a capire il problema. `int` è spesso la scelta più semplice. Se invece stai leggendo un formato binario che definisce esattamente un byte, quel tipo racconta qualcosa del formato. Con `long` devi ricordare il suffisso `L` per un letterale che non rientra nell'intervallo di `int`; senza il suffisso, scrivere un numero troppo grande non diventa automaticamente una dichiarazione `long` valida.

La tabella non assegna una dimensione a `boolean`, perché il linguaggio ne definisce i due valori, non un ingombro universale di un bit in memoria. Non mette neppure `void` fra i valori che una variabile può contenere: `void` appartiene alla dichiarazione di un metodo che non restituisce un risultato. La dimensione di un tipo e il significato di una dichiarazione rispondono quindi a domande diverse. Se devi sapere quanta memoria occupa una particolare struttura in una JVM concreta, non ricavare il dato dalla sola tabella dei tipi: entrano in gioco anche rappresentazione degli oggetti, allineamento e implementazione.

Proviamo a leggere alcuni valori concreti prima di scrivere altre dichiarazioni. `127` entra in un `byte`, `128` no; `32_767` entra in uno `short`, `32_768` no. Il limite non

riguarda il numero di cifre che vediamo sullo schermo, ma l'intervallo del tipo. La scrittura `3_000_000_000L` indica un `long`: la lettera finale fa parte del letterale Java, mentre i trattini bassi separano visivamente gruppi di cifre e non modificano il valore. La forma `3.0` indica invece un valore a virgola mobile di tipo `double`. Pur rappresentando matematicamente un intero, non può essere assegnata direttamente a una variabile `int`: Java non decide di scartare da sé l'eventuale parte frazionaria.

Per `float` e `double` la domanda non è soltanto «qual è il valore massimo?». La precisione è finita: fra due numeri consecutivi rappresentabili esistono altri numeri reali che il computer deve approssimare. Se scrivi `0.1 + 0.2`, non puoi dare per scontato che il confronto esatto con `0.3` sia vero. Questo non rende inutili i tipi a virgola mobile; li rende adatti ai problemi che ammettono un errore di arrotondamento controllato. Il conteggio dei centesimi di una fattura, invece, richiede un modello diverso, per esempio un numero intero di centesimi o `BigDecimal` con regole di arrotondamento esplicite. Non useremo queste alternative come scorciatoia prima di aver capito il problema che risolvono.

> **Approfondimento – Dimensione e significato.** Gli interi con segno seguono intervalli fissati dal linguaggio. `char` è diverso: i suoi 16 bit rappresentano un'unità UTF-16 senza segno. `boolean` rappresenta una scelta fra due valori logici; la specifica non lo trasforma in un intero sul quale fare somme. Tre tipi possono quindi richiedere tutti una forma di memorizzazione, ma offrire operazioni e significati molto diversi.

Alcune regole sintattiche

Una dichiarazione semplice come `int voto = 24;` contiene il tipo, il nome, un valore iniziale e un punto e virgola che termina l'istruzione. Le parentesi graffe delimitano blocchi; le parentesi tonde racchiudono, fra le altre cose, argomenti e condizioni. Java distingue maiuscole e minuscole: `voto` e `Voto` sono due nomi diversi. Non scegliere nomi che differiscono soltanto per una lettera maiuscola; il compilatore li accetterebbe, ma il lettore dovrebbe guardarli due volte.

I nomi possono contenere lettere Unicode e altri caratteri ammessi dalla specifica, ma negli esempi useremo nomi italiani semplici e senza accenti per evitare problemi di lettura e strumenti. Le parole chiave del linguaggio, come `class` e `int`, non possono diventare nomi di variabili. Le parentesi e il punto e virgola non sono ornamenti: cambiarne uno può modificare la struttura del programma o impedirne la compilazione.

L'indentazione non cambia normalmente la grammatica di un blocco Java: sono le parentesi graffe a delimitarlo. Rimane però fondamentale per una persona che legge. Un programma scritto tutto su una riga può compilare e risultare quasi impossibile da verificare. Una piccola eccezione pratica riguarda il testo di alcuni letterali e commenti, dove gli spazi fanno parte del contenuto: «gli spazi non contano» è quindi una semplificazione da usare con cautela.

Per costruire la sintassi, possiamo partire da tre regole schematiche e arrivare a un `if` dentro un altro `if`. Il punto è che un blocco può contenere istruzioni, fra le quali un altro blocco o una decisione. Per esempio, quando controlliamo prima se un

voto è valido e poi se è sufficiente, la seconda domanda ha senso soltanto all'interno del primo ramo:

```
int voto = 24;
if (voto >= 0 && voto <= 30) {
    if (voto >= 18) {
        System.out.println("Esame superato");
    }
}
```

Partendo dalla prima graffa, il compilatore incontra una seconda istruzione `if` e poi il blocco che contiene la stampa. Le due graffe di chiusura terminano prima il blocco interno e poi quello esterno. Se il voto è `-1`, non si arriva nemmeno alla domanda sulla sufficienza; se vale `17`, si entra nel blocco esterno ma non in quello interno. Questa lettura passo dopo passo conserva la funzione dell'esempio storico senza presentare come grammatica completa tre formule che non descrivono tutte le istruzioni Java.

Un punto e virgola dopo una semplice assegnazione termina quell'istruzione; un punto e virgola messo immediatamente dopo `if (voto >= 18)` costruirebbe invece un'istruzione vuota. Il blocco che segue non sarebbe più il ramo condizionato, anche se l'indentazione lo facesse sembrare tale. Le graffe sono una piccola spesa di caratteri che evitano un grande equivoco. Nel prossimo capitolo torneremo su `if`, `else` e sull'annidamento; qui ci interessa vedere come blocchi, espressioni e istruzioni si incastrano davvero.

> **Buona pratica – Un nome racconta lo scopo.** `sommaVoti` è più utile di `x` quando il valore serve per una media. Un nome breve come `i` può andare bene per un indice locale il cui significato è visibile in due righe; fuori da quel contesto richiede troppe supposizioni.

Dichiarazione, inizializzazione e ambito

Una variabile locale viene dichiarata dentro un metodo o un blocco. Prima di leggerla deve avere un valore assegnato in ogni percorso possibile fino a quel punto. Questo controllo si chiama *definite assignment*: il compilatore impedisce che una variabile locale venga usata quando potrebbe non essere inizializzata. I campi di un oggetto seguono regole diverse e ricevono valori iniziali predefiniti; non sfruttare questa differenza per nascondere un'inizializzazione dimenticata.

Possiamo dichiarare più variabili dello stesso tipo nella stessa istruzione, come `int primo = 24, secondo = 27;`. È sintassi lecita, ma due righe separate diventano più facili da modificare se i valori acquistano significati diversi. La dichiarazione introduce un nome; l'assegnazione `primo = 25;` sostituisce il valore che quel nome contiene. Nella matematica il segno = afferma un'uguaglianza; qui indica un'azione. Per questo `primo = primo + 1;` ha senso in Java: si legge il vecchio valore, si calcola il nuovo e lo si memorizza nella stessa variabile.

Una dichiarazione collocata dentro le graffe interne non è disponibile dopo la loro chiusura. Se dichiarassimo `int somma = 51;` soltanto nel blocco interno dell'esempio precedente, una stampa di `somma` nel blocco esterno non compilerebbe. Se usassimo anche variabili non inizializzate, introdurremmo due

guasti nello stesso frammento. Per isolare la regola, inizializziamo i dati e spostiamo la dichiarazione di `somma` nel blocco che deve leggerla. Non confondiamo la visibilità del *nome* con la memoria fisica: la JVM non è obbligata a rappresentare ogni variabile locale come una casella immobile sullo stack, e un oggetto può sopravvivere all'ambito del nome locale che lo ha creato.

Per vedere la differenza, immagina `int totale;` dentro `main`. La dichiarazione introduce il nome e il tipo, ma leggere `totale` prima di un'assegnazione produce un errore di compilazione. Se invece crei `new int[3]`, i tre elementi dell'array esistono già e valgono inizialmente `0`. Un campo `int` di un nuovo oggetto riceve a sua volta il valore predefinito `0`; un campo di tipo riferimento parte da `null` e un campo `boolean` da `false`. Questo non significa che `0`, `false` o `null` siano sempre valori sensati per il modello. Una classe `Prestito`, per esempio, non dovrebbe considerare valida una data mancante soltanto perché il campo riferimento è stato inizializzato automaticamente a `null`. La regola del linguaggio e la regola del problema restano due cose distinte.

```
int primoVoto = 24;
int secondoVoto = 27;
int sommaVoti = primoVoto + secondoVoto;
```

L'**ambito**, o *scope*, dice in quali punti del sorgente un nome può essere usato. Le parentesi graffe di un blocco delimitano spesso l'ambito di una variabile locale. Ambito non significa durata dell'oggetto: un oggetto creato in un blocco può restare raggiungibile anche dopo la fine del blocco se un riferimento è stato conservato altrove. Per il momento, la regola operativa è semplice: dichiara una variabile vicino al punto in cui serve e non allargare inutilmente l'area in cui può essere modificata.

Il tipo dichiarato e `var`

In alcune dichiarazioni locali il compilatore può ricavare il tipo dal valore iniziale. Scrivere `var somma = 24 + 27;` non crea una variabile «senza tipo» e non permette di trasformarla poi in una stringa: il tipo inferito è `int`. Il nome `var` si usa soltanto nelle posizioni previste per l'inferenza locale, non al posto del tipo di un campo o di un parametro ordinario.

> **Nota bene – `var` non è un tipo dinamico.** Dopo `var numero = 3;`, un'assegnazione come `numero = "tre";` non compila. Usa l'inferenza quando il tipo resta evidente dal contesto; scrivi il tipo per esteso quando aiuta il lettore a capire il significato del dato.

`final` e la parola «costante»

Una variabile dichiarata `final` può ricevere un valore una sola volta. Questo è utile quando una riassegnazione sarebbe un errore:

```
final int numeroVoti = 3;
```

Se il valore è un riferimento a un oggetto modificabile, `final` blocca la riassegnazione del riferimento, non le modifiche dell'oggetto. `final int[] voti = {24, 27, 30};` impedisce di far puntare `voti` a un altro array, ma permette `voti[0] = 25;`. Per parlare di *immutabilità* occorre controllare anche lo stato accessibile attraverso il riferimento. Torneremo su questo punto quando progetteremo le classi.

Il problema emerge quando passiamo l'array a un metodo che dovrebbe soltanto calcolare una media. Scrivere `final` davanti al parametro impedisce al metodo di assegnargli un altro array; non gli impedisce di eseguire `voti[0] = 0`. Se il chiamante si aspetta che i tre voti restino invariati, deve esserci un contratto rispettato dal metodo oppure una copia difensiva dei dati al confine opportuno. Possiamo verificarlo con una variante piccola: dopo la chiamata, stampiamo `voti[0]` e controlliamo se è ancora `24`. Una parola nella dichiarazione non sostituisce questa prova.

> **Per trasferire il mattone.** Quando incontrerai una lista non modificabile nel capitolo sulle collezioni, chiediti di nuovo *che cosa* non può essere modificato: la variabile che contiene il riferimento, la struttura della lista o gli oggetti che contiene. Sono tre promesse distinte. `final` risponde soltanto alla prima; il tipo di collezione e il contratto degli elementi rispondono alle altre.

Numeri, caratteri e testo

`int` è una scelta naturale per piccoli contatori e valori interi; `long` serve quando occorre un intervallo più ampio. Per numeri con frazione useremo spesso `double`. Il fatto che un valore possa essere scritto con la virgola non significa però che ogni numero decimale sia rappresentabile esattamente in binario: calcoli monetari o confronti numerici delicati richiedono scelte specifiche, che discuteremo più avanti. `boolean` contiene `true` o `false` e permette di rappresentare una condizione.

Il tipo `char` merita una pausa. Non corrisponde sempre a un «carattere» nel senso in cui una persona vede un simbolo sullo schermo: rappresenta una **unità di codice UTF-16** di 16 bit. Alcuni caratteri Unicode richiedono due unità di codice, e una sequenza visibile può contenere più punti di codice. Per manipolare testo ordinario useremo `String`; quando servirà contare o trasformare i punti di codice lo diremo esplicitamente.

Nei letterali Java alcune sequenze iniziano con una barra rovesciata e rappresentano caratteri che non scriviamo comodamente come testo normale. `\n` è un a capo, `\t` una tabulazione, `\\` una barra rovesciata e `\"` una virgoletta doppia dentro una stringa. Un letterale `char` usa apici singoli, come `'A'`, mentre una `String` usa virgolette doppie, come `"A"`. La notazione Unicode `\uXXXX` richiede quattro cifre esadecimali e ha regole di elaborazione proprie; non va confusa con qualsiasi sequenza preceduta da `\`.

> **Approfondimento – Una misura può rispondere a domande diverse.** La lunghezza di una `String` conta unità di codice UTF-16. Il numero di punti di codice e quello dei simboli percepiti dal lettore possono essere diversi. Perciò «questa stringa ha cinque caratteri» è ambiguo finché non diciamo che cosa stiamo contando.

I letterali possono essere resi più leggibili: `1_000_000` è lo stesso intero di `1000000`; `0b1010` usa la base due; `0xFF` la base sedici. Il suffisso `L` rende esplicito un valore `long`, per esempio `3_000_000_000L`; il suffisso `F` identifica un letterale `float`. Queste forme non cambiano il valore matematico, ma aiutano a esprimere il contesto.

Le basi diverse sono modi di scrivere lo stesso tipo di valore. `10`, `0b1010` e `0xA` indicano tutti l'intero dieci. La forma binaria fa vedere quali bit sono attivi, quella esadecimale compatta gruppi di quattro bit e la forma decimale resta la più comoda per una quantità ordinaria. In un protocollo che attribuisce un significato a ciascun bit, `0b1010` racconta più di `10`; nel numero di studenti di una classe accadrebbe il contrario. I trattini bassi possono separare gruppi, ma non si collocano arbitrariamente accanto al prefisso di base o al punto decimale: prova sempre un letterale nuovo nel compilatore prima di adottarlo in molti esempi.

Per il testo, `'A'` e `"A"` non sono intercambiabili: il primo è un valore `char`, il secondo un oggetto `String`. Anche `""`, la stringa vuota, è diverso da `null`: la prima esiste e ha lunghezza zero, il secondo indica che manca un riferimento a una stringa. Questa distinzione diventerà importante quando confronteremo o concateneremo valori. Se scriviamo `"Voto: " + 24`, l'operatore `+` produce una stringa; `24 + 3` è invece una somma numerica. Basta spostare le parentesi per cambiare il risultato: `"Somma: " + (24 + 3)` stampa `Somma: 27`, mentre `"Somma: " + 24 + 3` stampa `Somma: 243`. Java valuta da sinistra verso destra le due concatenazioni già determinate dalla struttura dell'espressione.

Operatori e conversioni

L'operatore `=` assegna un valore, mentre `==` confronta due valori. `+=` combina un'operazione con l'assegnazione, per esempio `somma += voto;`. Gli operatori aritmetici più comuni sono `+`, `-`, `*`, `/` e `%`. Con due interi, `/` produce una divisione intera: `5 / 2` vale `2`; `5 % 2` vale `1`, il resto. Per ottenere un risultato con frazione, almeno un operando deve essere convertito in un tipo a virgola mobile, come vedremo nell'esempio della media.

L'ordine delle operazioni conta. In `3 + 4 * 2`, la moltiplicazione precede l'addizione e il risultato è `11`; in `(3 + 4) * 2`, le parentesi fanno calcolare prima la somma e il risultato è `14`. Una tabella di precedenze può aiutare, ma conviene anzitutto imparare a riconoscere le espressioni e usare parentesi quando l'intenzione non è immediata. La tabella completa e normativa resta nella specifica delle espressioni di Java 25.

Per non ridurre quella tabella a un rinvio, leggiamo le famiglie più frequenti nell'ordine in cui ci servono. Le parentesi esplicitano il raggruppamento; gli operatori unari `+`, `-`, `!` e `~` si applicano a un operando; `*`, `/` e `%` precedono `+` e `-` binari. I confronti numerici come `<` precedono i confronti di uguaglianza come `==`; `&&` precede `||`; l'assegnazione arriva dopo il calcolo del lato destro. Non è necessario imparare a memoria una lunga fila di simboli prima di fare il primo

esercizio. Se due operatori diversi convivono nella stessa riga e l'ordine non salta agli occhi, le parentesi fanno lavorare meno chi legge senza cambiare l'intenzione del programma.

L'**associatività** risponde a una domanda diversa dalla precedenza: che cosa accade quando due operatori dello stesso livello compaiono insieme? `20 - 5 - 3` si raggruppa come `(20 - 5) - 3` e vale `12`, non `18`. L'espressione `a = b = 5`, invece, assegna prima `5` a `b` e poi il valore dell'assegnazione ad `a`. È lecita, ma in una spiegazione introduttiva due istruzioni separate rendono più facile vedere quali variabili cambiano. La priorità grammaticale non sostituisce neppure l'ordine di valutazione: Java valuta gli operandi seguendo regole precise, e un metodo chiamato dentro un'espressione potrebbe avere effetti. Per questo è meglio non nascondere più modifiche di stato in un'unica riga aritmetica.

La divisione intera va osservata insieme al resto. `7 / 3` vale `2` e `7 % 3` vale `1`, perché `7` si può scrivere come `3 * 2 + 1`. `-7 / 3` vale `-2`, poiché la divisione intera tronca verso zero; `-7 % 3` vale `-1`. Non dare dunque per scontato che `% 2` restituisca sempre `0` o `1` se l'operando può essere negativo. Con un divisore intero uguale a zero, `/` e `%` sollevano `ArithmeticException` durante l'esecuzione quando l'espressione vi arriva. Una media deve controllare che il numero degli elementi non sia zero prima di dividere, come faremo nell'esempio del capitolo successivo.

Java applica conversioni numeriche in vari contesti. In `long grande = 3;` il valore `int` può essere allargato a `long`. Il percorso opposto può perdere informazione e richiede una conversione esplicita, detta *cast*: `int piccolo = (int) grande;`. Il cast non è una promessa che il valore rientri nell'intervallo. Se la conversione restringe troppo, il risultato può sorprendere; prima di usarla, verifica quale informazione stai accettando di perdere. La specifica delle conversioni distingue i contesti e le regole precise.

Consideriamo un esempio numerico con un risultato che possiamo prevedere. `Long.MAX_VALUE` vale `9_223_372_036_854_775_807`; il cast a `int` conserva soltanto la parte rappresentabile secondo le regole del linguaggio e produce `-1`. Al contrario, assegnare `Integer.MAX_VALUE` a un `long` non perde informazione: `2_147_483_647` rientra nell'intervallo del tipo più ampio. Anche `char lettera = 'A'; int codice = lettera;` è una conversione numerica: in questo caso `codice` vale `65`. La parola *codice* qui indica l'unità UTF-16 di `A`, che coincide con il suo punto di codice Unicode.

```
long grande = Long.MAX_VALUE;
int ristretto = (int) grande;        // -1
long ampliato = Integer.MAX_VALUE;  // 2147483647
int codice = 'A';                    // 65
```

Il frammento mostra tre operazioni, ma non è un programma completo. Nei materiali del capitolo `ConversioniNumeriche.java` contiene anche il metodo `main` e stampa i valori, così puoi controllare la previsione senza affidarti alla sola spiegazione.

> **Nota bene – La somma può cambiare tipo.** Un'operazione aritmetica su `byte` o `short` viene normalmente promossa almeno a `int`. Perciò `byte a = 1; byte b = 2; byte c = a + b;` non compila, benché il risultato matematico sia piccolo. Il compilatore guarda il tipo dell'espressione, non soltanto il valore che ci aspettiamo in questo caso.

Calcoliamo `1_500 * 1_000_000_000`. Se dichiariamo entrambi gli operandi come `int`, il prodotto viene eseguito come `int` e supera il suo limite: assegnarlo *dopo* a un `long` non ripara il valore già calcolato. Se almeno un operando è `long`, la promozione numerica porta il prodotto a `long` prima dell'operazione:

```
int quantita = 1_500;
int prezzoInCentesimi = 1_000_000_000;
long errato = quantita * prezzoInCentesimi;
long corretto = (long) quantita * prezzoInCentesimi;
```

Il nome `errato` esprime l'errore rispetto alla quantità matematica attesa, non un errore di sintassi: entrambe le istruzioni compilano. `corretto` vale `1_500_000_000_000L`; il cast dell'operando sinistro basta perché la moltiplicazione avvenga in `long`. In un programma reale chiederemmo anche se un `long` è abbastanza ampio per tutti i dati previsti. L'overflow intero ordinario non lancia automaticamente un'eccezione; se superare il limite deve essere un errore riconoscibile, le API `Math.multiplyExact` e `Math.addExact` possono segnalarlo.

Una conversione che **allarga** un intero, come da `int` a `long`, conserva il suo valore. Non tutte le conversioni verso un tipo apparentemente «più grande» conservano ogni dettaglio: un `long` molto grande trasformato in `double` può perdere cifre meno significative, perché il formato a virgola mobile ha una precisione finita. Una conversione che **restringe**, come da `long` a `int`, richiede normalmente un cast e può perdere parte del valore. Infine, una costante intera letterale può essere assegnata a `byte`, `short` o `char` se il suo valore rientra nell'intervallo: `byte piccolo = 100;` compila, mentre `byte troppoGrande = 128;` no. La somma di due variabili `byte` resta comunque un'espressione `int`; non estendere la regola delle costanti a qualunque risultato.

> **Concetto chiave – Il tipo dell'espressione nasce prima della destinazione.** In `long risultato = a * b;` il tipo della variabile finale non decide retroattivamente come moltiplicare `a` e `b`. Osserva gli operandi, individua la promozione applicata e solo dopo chiediti se il risultato può essere assegnato. È lo stesso metodo che useremo per la media di un array.

Gli operatori di confronto `<`, `<=`, `>`, `>=`, `==` e `!=` producono un `boolean`. `&&` e `||` combinano condizioni e valutano il secondo operando soltanto quando serve; `!` nega una condizione. Gli operatori `&` e `|` possono anche agire su valori interi bit per bit; usati con valori booleani valutano entrambi gli operandi. Gli spostamenti `<<`, `>>` e `>>>` lavorano sui bit degli interi: sono strumenti utili in protocolli e formati binari, ma non migliorano automaticamente un normale calcolo. L'elenco degli operatori nella specifica è la fonte per precedenza e semantica completa.

La piccola tabella seguente permette di controllare le combinazioni dei due valori booleani. `&&` è vero soltanto quando entrambi sono veri; `||` è vero quando almeno

uno è vero; `^` è vero quando i due valori sono diversi. L'operatore unario `!` inverte un solo valore: `!true` vale `false`.

a	b	a && b	a || b	a ^ b
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

L'effetto di `&&` e `||` si chiama *valutazione corta*, o *short circuit*. Se `testo != null` è falso, in `testo != null && !testo.isEmpty()` Java non valuta la parte destra. Questo evita di chiamare `isEmpty()` attraverso un riferimento nullo. Il vantaggio non è soltanto la velocità: l'ordine delle condizioni esprime una dipendenza logica. Cambiarlo può far fallire il programma.

Questa regola vale anche per `||`, ma con il motivo opposto: se la parte sinistra è già vera, non serve controllare la destra. In `autorizzato || codiceValido()`, il metodo `codiceValido()` non viene chiamato quando `autorizzato` è vero. Se il metodo ha effetti, per esempio registra un tentativo, quei due caratteri `||` cambiano anche il comportamento osservabile del programma. Gli operatori booleani `&` e `|` valutano invece entrambi i lati. Non sceglierli per obbligare una chiamata laterale a essere eseguita: separare la chiamata dalla decisione rende più chiara l'intenzione.

Gli operatori di assegnazione composta meritano una lettura propria. `totale += voto` aggiunge `voto` al totale e conserva il risultato nella stessa variabile; `-=`, `*=`, `/=` e `%=` seguono l'operazione indicata dal primo simbolo. Esistono anche `&=`, `|=`, `^=`, `<<=`, `>>=` e `>>>=`, utili quando lo stato è davvero una maschera di bit. `>>>=` sposta i bit verso destra senza propagare il segno; un operatore `<<<` non esiste in Java. Scrivere `x += y` assomiglia a `x = x + y`, ma la specifica include una conversione implicita verso il tipo della variabile a sinistra. Per esempio, dopo `byte piccolo = 1;`, `piccolo += 2;` compila, mentre `piccolo = piccolo + 2;` richiede una conversione esplicita. La forma breve può quindi nascondere una perdita di informazione: se l'intervallo conta, controlla il risultato invece di fidarti della brevità.

L'operatore condizionale `condizione ? valoreSeVero : valoreSeFalso` sceglie una fra due espressioni. `String esito = voto >= 18 ? "promosso" : "da recuperare";` è leggibile perché entrambi i rami producono un valore semplice. Quando ciascun ramo richiede più istruzioni o un commento per spiegare perché esiste, `if` e `else` del prossimo capitolo offrono più spazio al ragionamento. Non confondere una forma compatta con una scelta automaticamente migliore.

Con valori booleani `^` vale `true` quando esattamente uno dei due operandi è vero. Con interi, invece, `&`, `|`, `^` e `~` agiscono bit per bit. Per esempio, `100 & 125` vale `100`, mentre `100 ^ 125` vale `25`. `~100` vale `-101` perché l'operatore inverte tutti i bit dell'intero con segno, non soltanto quelli che scegliamo di scrivere su carta. Occorre mantenere coerenti operando e risultato: `~125` varrebbe `-126`, non `-101`.

Gli spostamenti richiedono una distinzione simile. `100 << 1` vale `200`; `100 >> 1` vale `50`. Per numeri negativi e casi di overflow, trattare lo shift come una moltiplicazione o divisione ordinaria dà risultati sbagliati. `>>` propaga il bit del

segno, mentre `>>>` inserisce zeri a sinistra. Usa questi operatori quando devi esprimere un'operazione sui bit; non sostituire automaticamente `* 2` o `/ 2` nella speranza di rendere più veloce il programma. Il compilatore e la JVM possono già ottimizzare operazioni semplici, e la misura su un caso reale è l'unica prova di un vantaggio.

Consideriamo il programma `ProdottoDivisione`, perché osservare una **sequenza** insegna più di una singola uguaglianza. Partendo da `100`, tre spostamenti a destra danno `50`, `25` e `12`; tornando a `100`, tre spostamenti a sinistra danno `200`, `400` e `800`. Il passaggio da `25` a `12` mostra subito che non stiamo facendo una divisione fra numeri reali: l'intero scarta la parte frazionaria. In binario, sugli otto bit meno significativi, `100` si scrive `01100100`; uno spostamento a destra lo porta a `00110010`, cioè `50`. La rappresentazione di un `int` Java ha però **32 bit**: il gruppo di otto serve soltanto a vedere il passaggio, non è l'intero valore in memoria.

Il comportamento dei numeri negativi rende visibile la differenza fra i due spostamenti a destra. `-8 >> 1` vale `-4`, mentre `-8 >>> 1` vale `2_147_483_644` per un `int`: nel secondo caso gli zeri entrano da sinistra e il segno non viene conservato. `>>>` è uno spostamento a destra senza estensione del segno; chiamarlo «shift a sinistra senza segno» invertirebbe la direzione dell'operazione. Anche il numero di posizioni va interpretato secondo il tipo dell'operando sinistro: per un `int` Java usa soltanto i cinque bit meno significativi della distanza. Perciò `1 << 32` ha lo stesso risultato di `1 << 0`, ossia `1`; non produce zero e non sposta un bit oltre la parola. È un dettaglio da conoscere quando si leggono protocolli binari, non un trucco da inserire nei calcoli ordinari.

Per controllare `&`, `|` e `^` senza confondere le cifre decimali con i bit, allineiamo due valori su otto posizioni: `100` è `01100100` e `125` è `01111101`. Dove entrambi hanno un `1`, `&` conserva quel bit e otteniamo `01100100`, cioè `100`. `|` conserva ogni posizione in cui almeno un operando ha `1` e dà `01111101`, cioè `125`. `^` conserva le posizioni diverse e produce `00011001`, cioè `25`. Il complemento `~100` inverte invece tutti i 32 bit dell'`int`; leggere soltanto gli otto finali porterebbe all'errata conclusione che il risultato sia `155`. Il valore completo è `-101`. Questi risultati sono raccolti nel programma riproducibile `BitPassoPasso.java`, che accompagna la spiegazione senza fare degli shift una raccomandazione di prestazioni.

> **Nota bene – Maiuscole e maschere di bit.** Trasformare le lettere minuscole in maiuscole azzerando un bit è una scorciatoia con un limite preciso. Funziona soltanto per un sottoinsieme preciso di lettere latine e può cambiare in modo inatteso altri caratteri. Per il testo usa le API di `Character` o `String` con una scelta esplicita sulla lingua quando serve. Le maschere di bit restano un buon esempio per protocolli e formati binari, dove i bit hanno un significato dichiarato.

L'esempio storico `Maiuscolo` merita una prova, perché il suo risultato per `'a'` sembra convincente. Il codice `97 & 223` produce `65`, e questi due numeri corrispondono rispettivamente a `'a'` e `'A'` nel sottoinsieme latino di base. Ma la stessa maschera applicata a un carattere che non sia una lettera minuscola prevista può cambiarlo senza alcun motivo linguistico. Inoltre `args[0].charAt(0)` legge soltanto la prima unità UTF-16 del primo argomento; con un argomento assente fallisce prima ancora di mostrare un risultato. Per una singola lettera semplice `Character.toUpperCase('a')` rende chiara l'intenzione. Per una parola si può usare una trasformazione di `String` con un `Locale` scelto in base alla lingua e

al contesto. Non esiste una sottrazione universale di `32` che trasformi correttamente tutto Unicode: questa è la correzione tecnica, mentre il ragionamento sui bit resta nel paragrafo precedente.

Prendi l'abitudine di separare tre livelli quando un'operazione sorprende: **rappresentazione**, **tipo** e **significato**. La rappresentazione binaria spiega perché `100 & 125` dà `100`; il tipo `int` spiega perché `~100` coinvolge trentadue bit; il significato del dato decide se la maschera è una soluzione sensata. Applicare un operatore bit a bit a un codice di testo senza una regola sul repertorio dei caratteri salta l'ultimo livello. Lo stesso metodo di lettura sarà utile quando lavoreremo con gli array: un indice è un intero, ma il significato di quella posizione appartiene al problema che stiamo modellando.

Anche `++` e `--` meritano attenzione. Se `int i = 0;`, l'espressione `i++` produce il vecchio valore e poi incrementa `i`, mentre `++i` incrementa prima e produce il nuovo valore. Quando l'espressione è usata da sola come istruzione, entrambe portano `i` a `1`; la differenza conta se il valore dell'espressione viene assegnato altrove. Preferisci separare i passaggi se una riga come `int j = i++` costringe a fermarsi per capire l'ordine.

> **Buona pratica – Parentesi quando chiariscono.** La precedenza degli operatori è definita dal linguaggio, ma una persona non dovrebbe doverla ricostruire ogni volta. Scrivi `(somma / numeroVoti) * 100` quando quella sequenza è importante per il ragionamento, e spiega il risultato se il tipo degli operandi cambia il calcolo.

Array: più valori dello stesso tipo

Tre variabili separate bastano per tre voti, ma diventano scomode quando i voti sono trenta. Un array raccoglie un numero fisso di elementi dello stesso tipo e permette di raggiungerli con un indice. L'indice del primo elemento è `0`; l'ultimo indice valido è `length - 1`. In un array di lunghezza tre, quindi, gli indici sono `0`, `1` e `2`.

La forma `int[] voti` mette in evidenza che il tipo è «array di interi». Java ammette anche la forma storica `int voti[]`, ereditata da altri linguaggi, ma negli esempi useremo la prima. Dichiarare `int[] voti;` crea una variabile di riferimento, non ancora l'array. `voti = new int[3];` crea l'oggetto array e ne fissa la lunghezza. Dopo la creazione, la lunghezza non cambia; se ti serve una sequenza che cresce e si riduce, il capitolo sulle collezioni mostrerà altre strutture.



Figura 4.1 – Ogni posizione ha un indice, separato dal valore che contiene. `length` *vale tre; provare a leggere* `voti[3]` *non significa chiedere il terzo elemento, ma una posizione inesistente.*

```
int[] voti = {24, 27, 30};
int primo = voti[0];
int numeroVoti = voti.length;
```

`int[]` è il tipo dell'array; le parentesi graffe forniscono i valori iniziali. Possiamo anche creare `new int[3]` e assegnare poi gli elementi, ma in quel caso ciascuna posizione comincia con il valore predefinito `0`. Se l'indice è fuori dall'intervallo, l'accesso fallisce durante l'esecuzione con `ArrayIndexOutOfBoundsException`. Il compilatore non può sempre conoscere il valore che un indice avrà quando il programma girerà.

Facciamo il percorso inverso rispetto alla dichiarazione compatta. Dopo `int[] voti = new int[3];`, il programma ha un array di lunghezza tre e i valori sono `0`, `0`, `0`. L'istruzione `voti[1] = 27;` cambia **la seconda posizione**, non la ventisettesima e non il numero di elementi: l'array diventa `{0, 27, 0}`. `voti.length` continua a valere 3. Se scriviamo `voti[3] = 30;`, il codice compila perché l'indice è un intero, ma durante l'esecuzione fallisce: le sole posizioni valide sono `0`, `1` e `2`. Non si tratta di un errore della JVM da ignorare; ci sta dicendo che il programma ha fatto un'ipotesi sbagliata sul limite.

La lunghezza può anche essere zero. `new int[0]` crea un array valido che non contiene elementi; non esiste un `voti[0]` da leggere. Un algoritmo generale deve quindi controllare la lunghezza prima di accedere al primo elemento o dividere per il numero degli elementi. Questo spiega perché il programma `MediaVoti` qui sotto funziona con i tre valori stampati, ma non è ancora la versione definitiva: la somma scritta a mano presuppone proprio tre posizioni. Nel capitolo 5 useremo un ciclo e gestiremo esplicitamente il caso vuoto.

> **Concetto chiave – Lunghezza, capacità e numero di valori significativi.** Per un array Java, `length` è il numero di posizioni create e rimane fisso. Non dice quanti elementi abbiamo già riempito con valori utili al problema. Un `int[20]` possiede venti posizioni fin dall'inizio, inizialmente tutte a zero; se abbiamo registrato tre voti, dobbiamo conservare separatamente il fatto che soltanto tre posizioni rappresentano voti reali. Più avanti vedremo strutture che gestiscono la crescita e il conteggio degli elementi per noi.

Ecco un programma completo che usa l'array per calcolare la media:

```
public class MediaVoti {
    public static void main(String[] args) {
        int[] voti = {24, 27, 30};
        int somma = voti[0] + voti[1] + voti[2];
        double media = (double) somma / voti.length;
        System.out.println("Media: " + media);
    }
}
```

La somma vale `81` e la lunghezza dell'array vale `3`, quindi l'output è `Media: 27.0`. Il cast `(double)` converte il valore di `somma` prima della divisione; senza di esso, Java eseguirebbe una divisione fra interi e soltanto dopo convertirebbe il risultato in `double`. Nei capitoli sul controllo di flusso sostituiremo la somma scritta elemento per elemento con un ciclo, così il programma funzionerà per array di lunghezza diversa.

Array con più dimensioni

In Java un array bidimensionale è un array i cui elementi sono a loro volta array. `int[][] caselle = new int[4][4];` crea quattro righe, ciascuna con quattro interi. Per leggere una posizione scriviamo prima l'indice della riga e poi quello della colonna, per esempio `caselle[1][2]`.

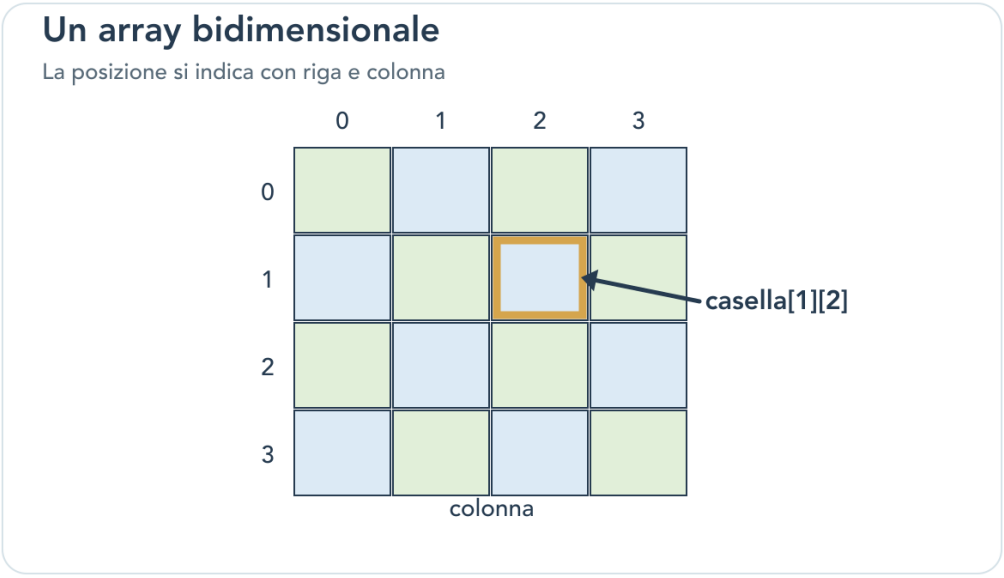


Figura 4.2 – La casella `caselle[1][2]` *è nella seconda riga e nella terza colonna. La figura mostra una griglia regolare; Java permette anche righe di lunghezze diverse.*

La tavola della dama rende concreta la griglia; la figura mette in evidenza gli indici, che sono ciò che dobbiamo imparare a usare. Un array di array può avere righe di lunghezza diversa: per questo, in un programma generale, controlliamo `caselle[riga].length` e non supponiamo che ogni riga abbia la lunghezza della prima.

Vediamo che cosa vuol dire «array di array» senza ricorrere subito alla tavola completa. `int[][] righe = new int[2][];` crea un array esterno con due posizioni; ciascuna contiene inizialmente `null`, perché deve ancora riferirsi a un array interno. Se assegniamo `righe[0] = new int[3];` e `righe[1] = new int[1];`, la prima riga ha tre numeri e la seconda uno soltanto. L'accesso `righe[1][0]` è valido, mentre `righe[1][1]` esce dalla seconda riga anche se `righe[0][1]` esiste. Una «matrice» rettangolare è dunque una scelta del nostro programma, non un vincolo del tipo `int[][]`.

Gli array sono oggetti e le variabili che li nominano contengono riferimenti. Se dopo `int[] primo = {24, 27};` scriviamo `int[] secondo = primo;`, non abbiamo copiato i due voti: `primo` e `secondo` indicano lo **stesso** array. `secondo[0] = 30;` farà leggere `30` anche attraverso `primo[0]`. Quando vogliamo due array indipendenti dobbiamo chiederne una copia, per esempio con `primo.clone()` per questo array semplice di primitivi, e dichiarare perché la copia serve. Questa distinzione prepara il terreno all'incapsulamento: una classe che consegna all'esterno il proprio array modificabile sta consegnando anche la possibilità di alterare il suo stato.

Un array può contenere riferimenti invece di numeri. In `String[] nomi = new String[2];` entrambe le posizioni partono da `null`, non da `""`. Dopo `nomi[0] = "Ada";`, soltanto la prima posizione indica una stringa. Prima di chiamare `nomi[1].length()` bisogna assegnarle un testo oppure trattare esplicitamente l'assenza; altrimenti si incontra una `NullPointerException`. La regola dei valori

iniziali che abbiamo visto per i campi e gli array si applica anche qui, ma il valore predefinito non rende l'elemento pronto per qualsiasi operazione.

> **Nota bene – Confrontare array.** `primo == secondo` controlla se i due riferimenti indicano lo stesso array, non se gli elementi sono uguali. Per confrontare il contenuto di due array a una dimensione useremo `java.util.Arrays.equals`. La classe di utilità offre anche `Arrays.toString`, utile per stampare gli elementi durante un esperimento: la stampa diretta di un array con `System.out.println(primo)` non elenca i voti nel modo che molti principianti si aspettano.

Questo capitolo ha introdotto più di una sintassi. Per affrontare un esercizio sugli array, comincia da tre domande: quale dato rappresenta ogni elemento, quanti elementi esistono e quali indici sono validi? Solo dopo scegli se inizializzare tutti i valori subito o riempirli in passi successivi. Per un esercizio numerico aggiungi una quarta domanda: con quale tipo Java esegue il calcolo *prima* di assegnarne il risultato? Sono domande semplici da porre su carta, e risparmiano tentativi casuali davanti a un risultato inatteso. Nel prossimo capitolo useremo proprio questi dati per decidere quali istruzioni ripetere e quali eseguire soltanto in determinate condizioni.

Per collocare le pedine di una dama useremo una matrice di otto righe. La figura isola la regola degli indici prima dell'applicazione completa. Possiamo iniziare una tavola con `String[][] dama = new String[8][8];` e collocare una pedina con `dama[0][1] = "bianca";`. Le caselle che non abbiamo assegnato contengono inizialmente `null`, non una stringa vuota. Per rappresentare **tutta** la disposizione iniziale useremo un ciclo nel prossimo capitolo, invece di copiare decine di assegnazioni; così potremo verificare che la stessa regola valga per ogni riga. Useremo una sola disposizione iniziale esplicitata e controllata, così figura, codice e conteggio delle pedine descriveranno la stessa tavola.

Per verificare

Prova a cambiare i voti in `{20, 25, 28, 30}`. Prima di avviare il programma, prevedi la media e indica quale riga dovrai modificare affinché la somma includa anche il nuovo elemento. Dopo averlo eseguito, spiega perché nel prossimo capitolo useremo un ciclo. Come seconda prova, scrivi su carta gli indici validi per un array di lunghezza zero e per uno di lunghezza cinque. L'attività è riuscita quando distingui valore, indice e lunghezza e sai motivare il tipo dell'espressione usata nella divisione.

5. Controllare il flusso del programma

Una sequenza che può scegliere

Nel programma della media abbiamo sommato tre elementi scrivendo tre addizioni. Funziona, ma appena il numero dei voti cambia dobbiamo cambiare il codice. Inoltre, che cosa dovrebbe accadere se l'array fosse vuoto? Per rispondere ci servono istruzioni che scelgano un percorso e istruzioni che ripetano un lavoro. Questo è il *controllo di flusso*: decidere quale parte del programma eseguire, quante volte e quando interromperla.

Una condizione in Java deve produrre un valore `boolean`, cioè `true` o `false`. Non è sufficiente un numero diverso da zero, come in altri linguaggi. Le parentesi graffe racchiudono le istruzioni di un ramo. Anche quando un ramo contiene una sola istruzione le useremo: aiutano a vedere i confini e rendono più sicura una modifica successiva.

Il capitolo precedente ci ha dato gli operatori per costruire condizioni, ma un'espressione booleana da sola non cambia il percorso del programma. Se calcoliamo `voti.length == 0`, otteniamo `true` oppure `false`; è l'istruzione `if` a decidere se eseguire il blocco successivo in base a quel valore. Questo è un passaggio importante perché separa due domande che spesso vengono fuse: «la condizione è vera?» e «che cosa faccio se lo è?». Quando una regola diventa complessa, possiamo darle un nome significativo prima di usarla in un ramo; così il lettore controlla la condizione e l'azione in due passaggi.

Le istruzioni possono seguire un percorso **sequenziale**, una **selezione** o una **ripetizione**. La sequenza esegue una istruzione dopo l'altra. La selezione con `if` o `switch` sceglie fra alternative. La ripetizione con `while`, `do-while` o `for` torna su un blocco finché ha senso farlo. Infine `break`, `continue` e `return` modificano il percorso dall'interno di questi costrutti o di un metodo. Non sono categorie decorative: se ti chiedi «voglio scegliere un caso, ripetere un lavoro o terminare?», hai già ristretto molto la scelta della sintassi da usare.

`if`, `else` e decisioni annidate

L'istruzione `if` esegue un blocco soltanto se la condizione è vera:

```
if (voti.length == 0) {
    System.out.println("Nessun voto disponibile");
}
```

Se vogliamo un percorso alternativo aggiungiamo `else`. Per esempio, una classe può distinguere un array vuoto da uno che contiene valori. Una catena `if-else if` permette più scelte, valutate nell'ordine. Quando un ramo viene eseguito, i successivi non vengono considerati. Per questo l'ordine delle soglie conta:

```
if (media >= 27) {
    giudizio = "ottimo";
} else if (media >= 18) {
    giudizio = "sufficiente";
} else {
    giudizio = "da recuperare";
}
```

Il primo ramo include anche i valori maggiori di `30`; se il dominio ammette soltanto voti da `0` a `30`, quel vincolo va controllato nel punto in cui i dati entrano nel programma. Le condizioni non diventano corrette perché sembrano plausibili. In particolare, invertire i primi due rami farebbe classificare `29` come «sufficiente», perché la seconda soglia intercetterebbe il valore prima di arrivare alla prima.

Prova a seguire mentalmente l'esecuzione per `media = 17`, `18`, `26` e `27`. Con `17` i primi due confronti sono falsi e si arriva al ramo finale. Con `18` il primo è falso e il secondo è vero. Con `26` accade lo stesso, benché il numero sia più alto. Con `27` il primo confronto è vero e gli altri rami non vengono nemmeno esaminati. Questo piccolo esercizio è più istruttivo che leggere tre etichette di giudizio: mostra che una catena di condizioni ha un ordine e che i confini degli intervalli devono essere provati. Se cambi `>= 27` in `> 27`, il valore `27` passa al ramo «sufficiente»; un solo carattere ha cambiato la regola.

Quando una decisione ne contiene un'altra, l'indentazione aiuta, ma non sostituisce le parentesi graffe. Se le condizioni annidate diventano molte, chiediti se stai cercando di rappresentare uno stato che merita un nome proprio o un metodo separato. Non c'è un numero magico di `if` consentiti: c'è un limite pratico alla quantità di casi che una persona riesce a tenere in mente mentre legge.

A quale `if` appartiene `else`?

Gli `if` annidati meritano attenzione perché una riga apparentemente innocente può cambiare il risultato. Senza parentesi graffe, un `else` viene associato all'`if` compatibile più vicino. Considera il voto `20`: vogliamo prima verificare che appartenga all'intervallo da `0` a `30`, poi decidere se sia almeno `18`. Se scriviamo i due controlli su righe diverse ma senza blocchi, l'indentazione può farci credere che l'`else` significhi «voto non valido» quando invece appartiene al controllo della promozione. Le graffe rendono visibile la struttura che il compilatore usa davvero.

Una catena `if–else if` non prova automaticamente che i casi siano completi e non sovrapposti. Nel nostro giudizio sui voti, `media >= 27` assorbe anche `31`, mentre il ramo finale assorbe anche `-1`. Se entrambi sono vietati, aggiungiamo un controllo di validità prima della classificazione. Questo separa due domande: «il dato ha senso?» e «in quale fascia cade?». La separazione aiuta a scrivere una prova per ogni confine: `-1`, `0`, `17`, `18`, `26`, `27`, `30`, `31`.

> **Nota bene – Uguaglianza e assegnazione.** In una condizione `==` confronta valori, mentre `=` assegna. Il compilatore intercetta molti scambi accidentali, ma non tutti i problemi logici: `boolean pronto = true; if (pronto = false) { ... }` è sintatticamente possibile e usa il valore assegnato come condizione. Se l'intenzione era confrontare, il programma è sbagliato.

`switch`: scegliere per casi

Quando più rami dipendono dallo stesso valore, `switch` può esprimere la scelta con maggiore chiarezza. La forma classica è un'**istruzione**: esegue azioni. Un esempio semplice usa un numero per rappresentare il giorno della settimana:

Nei frammenti che seguono assumiamo che `giorno` sia un `int` compreso normalmente fra `1` e `7`; il caso fuori intervallo sarà gestito esplicitamente. I frammenti sono parti di un metodo, mentre i programmi completi sono mostrati per intero o disponibili nei materiali del capitolo.

```
switch (giorno) {
    case 6, 7 -> System.out.println("Fine settimana");
    default -> System.out.println("Giorno feriale");
}
```

Le etichette `case` con `->` non cadono automaticamente nel caso successivo. La vecchia forma con i due punti esiste ancora e può essere necessaria per codice che condivide istruzioni fra casi, ma richiede attenzione al *fall through*: senza un'interruzione, l'esecuzione può proseguire nel ramo successivo. Nei nuovi esempi preferiremo le frecce quando esprimono meglio l'intenzione.

Nella forma classica `case 6:` segna il punto da cui cominciare, non un recinto chiuso. Se il codice stampa «sabato» e non incontra `break`, può proseguire nel blocco `case 7:` e stampare anche «domenica». A volte è voluto: più etichette possono condividere una stessa azione. Altre volte è un difetto difficile da vedere, soprattutto quando si aggiunge un nuovo caso fra due esistenti. La freccia `case 6, 7 -> ...` esprime direttamente che entrambi i numeri ricevono la stessa risposta e che quel ramo non prosegue nel successivo.

Per rendere osservabile il *fall through*, immaginiamo `giorno = 6` e due rami classici. Se `case 6:` stampa `sabato` e non termina con `break`, l'esecuzione prosegue nelle istruzioni di `case 7:` e stampa anche `domenica`: il compilatore non rivaluta il selettore fra le due stampe. Inserendo `break` dopo la prima, la seconda non viene eseguita. Un `default` collocato in fondo può a sua volta essere attraversato da un caso precedente che non si interrompe; la sua posizione non crea da sola una barriera. Questa era la parte difficile del secondo programma storico sui numeri in parole: ogni ramo richiedeva un `break` corretto. Con `case numero ->`, ciascun ramo è separato per costruzione.

Le etichette costanti della forma classica devono essere valori accettati dalla grammatica e determinabili in compilazione. `final String cane = "CANE";` può essere usata come etichetta di un `switch` su `String`; una variabile `String gatto = "GATTO";` che può cambiare non è una costante di compilazione, anche se al momento contiene sempre lo stesso testo nei nostri esempi. Le variabili dichiarate dentro un ramo con `case ... :` richiedono attenzione all'ambito condiviso del blocco `switch`: graffe locali attorno al ramo rendono visibile dove nasce e termina il nome. Nella forma a freccia con blocco, quelle graffe sono già il luogo naturale per i dati temporanei del ramo.

`switch` non è sempre preferibile a `if`. Se le decisioni dipendono da condizioni diverse, come `voto >= 18` e `presenze >= 10`, una catena di condizioni può essere più naturale. Se invece stai scegliendo fra valori distinti della stessa espressione, `switch` rende l'elenco dei casi più evidente. In entrambi i costrutti chiediti che cosa

accade per un valore non previsto: un `default` che stampa «valore non valido» è una decisione esplicita, non una toppa da aggiungere meccanicamente.

Trasformiamo un numero nella sua forma scritta in due modi: prima con una catena di `if` ed `else if`, poi con uno `switch`. Considera il valore `3`: nella catena si prova prima se è `0`, poi se è `1`, poi se è `2` e infine si arriva alla condizione per `3`. Nello `switch` si dichiara invece un caso per ogni valore previsto. Il risultato visibile, «tre», può essere identico; cambia il modo in cui il sorgente esprime la relazione fra valore e risposta. Non dedurre dalle righe di un esempio didattico quale forma sia sempre più veloce: per scegliere qui conta soprattutto la leggibilità della regola e la necessità di gestire valori fuori dall'intervallo.

L’intervallo ammesso è una parte essenziale del contratto del programma. Se la conversione è definita soltanto da `0` a `10`, che cosa deve accadere per `-1` o `11`? Lasciare una stringa non inizializzata o non stampare nulla renderebbe l'errore difficile da riconoscere. Possiamo scegliere un messaggio esplicito, rifiutare il dato prima della selezione oppure rappresentare l'assenza di una traduzione: l'importante è dichiarare la scelta. Anche i numeri scritti in lettere portano un piccolo problema di dominio: «uno» e «una» dipendono dal contesto. Un esempio chiamato `InteroComeStringa` deve dire se produce etichette convenzionali o parole adatte a una frase. Questo non cambia la sintassi di `switch`, ma ricorda che un programma corretto rispetto ai casi elencati può ancora essere incompleto rispetto al problema reale.

Le due versioni storiche meritano comunque di essere lette per intero, perché fanno sentire il peso di undici alternative meglio di un esempio con soli due giorni. Il programma seguente conserva gli stessi casi, da zero a dieci, e li mette in due metodi affiancati. `conIf` prova le condizioni dall'alto in basso e restituisce il testo appena ne trova una vera. `conSwitch` presenta gli stessi valori come etichette di un'unica espressione. In entrambi i casi il ramo finale tratta sia i numeri negativi sia quelli maggiori di dieci: l'antico messaggio che parlava solo dei numeri «maggiori» non descriveva il caso `-1`.

```
public class InteroComeStringaComparato {
    private static String conIf(int numero) {
        if (numero == 0) {
            return "zero";
        } else if (numero == 1) {
            return "uno";
        } else if (numero == 2) {
            return "due";
        } else if (numero == 3) {
            return "tre";
        } else if (numero == 4) {
            return "quattro";
        } else if (numero == 5) {
            return "cinque";
        } else if (numero == 6) {
            return "sei";
```

```
        } else if (numero == 7) {
            return "sette";
        } else if (numero == 8) {
            return "otto";
        } else if (numero == 9) {
            return "nove";
        } else if (numero == 10) {
            return "dieci";
        } else {
            return "fuori intervallo";
        }
    }

    private static String conSwitch(int numero) {
        return switch (numero) {
            case 0 -> "zero";


            case 1 -> "uno";
            case 2 -> "due";
            case 3 -> "tre";
            case 4 -> "quattro";
            case 5 -> "cinque";
            case 6 -> "sei";
            case 7 -> "sette";
            case 8 -> "otto";
            case 9 -> "nove";
            case 10 -> "dieci";
            default -> "fuori intervallo";
        };
    }

    public static void main(String[] args) {
        if (args.length != 1) {


            System.out.println("Uso: java InteroComeStringaComparato <intero>");
            return;
        }
        int numero = Integer.parseInt(args[0]);
        System.out.println("if: " + conIf(numero));
        System.out.println("switch: " + conSwitch(numero));
    }
}
```

Esegui `InteroComeStringaComparato.java` con `java InteroComeStringaComparato 3`: le righe stampate sono `if: tre` e `switch: tre`. Ripeti con `0`, `10`, `-1` e `11`, poi controlla che le due righe coincidano ogni volta. Senza argomenti, il programma mostra l'uso previsto anziché tentare di leggere una posizione inesistente dell'array `args`. Una stringa non numerica, invece, provoca `NumberFormatException`: la gestione di un input davvero interattivo richiederebbe una decisione ulteriore, che affronteremo nel capitolo sulle eccezioni. Qui isoliamo il confronto fra le due selezioni.

Il confronto mostra anche che una forma più compatta non cancella il lavoro di definire il dominio. Gli undici valori restano undici; `switch` evita la ripetizione di `numero ==`, ma ogni etichetta richiede ancora la parola corretta. Se domani includiamo `11`, dobbiamo aggiornare entrambe le versioni e il controllo dei confini.

Scrivere prima una tabella di casi attesi e poi implementarla è una buona difesa contro omissioni e refusi, qualunque costrutto scegliamo.

Una **espressione** `switch`, invece, produce un valore. Questo evita di inizializzare una variabile in più punti:

```
String tipoGiorno = switch (giorno) {
    case 6, 7 -> "fine settimana";
    case 1, 2, 3, 4, 5 -> "feriale";
    default -> "valore non valido";
};
```

L'espressione deve coprire tutti i valori possibili del tipo selezionato, altrimenti il compilatore non potrebbe garantire un risultato. Qui `default` si occupa dei numeri fuori dall'intervallo previsto. Se un ramo richiede più istruzioni, si può usare un blocco e `yield` per consegnare il valore dell'espressione; `return` avrebbe un significato diverso, perché uscirebbe dal metodo. La specifica delle istruzioni e dei pattern di Java 25 descrive entrambe le forme.

Per vedere la differenza, immagina un caso che deve prima costruire una spiegazione e poi produrre la stringa finale. Il ramo può contenere un blocco:

```
String descrizione = switch (giorno) {
    case 6, 7 -> {
        String nome = "riposo";
        yield "giorno di " + nome;
    }
    default -> "giorno ordinario";
};
```

`yield` consegna il valore del ramo all'espressione `switch`; la variabile `descrizione` riceve quel valore. Non usarlo in un normale `if` per «uscire dal blocco»: il suo ruolo qui è specifico. Nell'esempio il blocco è più lungo di quanto servirebbe davvero; lo mostriamo soltanto per leggere la sintassi senza aggiungere altri problemi.

> **Concetto chiave – Un valore oppure un effetto.** Usa `switch` come espressione quando devi calcolare un risultato da assegnare o restituire. Usalo come istruzione quando ciascun caso deve compiere un'azione. Scegli la forma che rende evidente lo scopo; la brevità da sola non basta.

Riconoscere il tipo con un pattern

Una forma moderna di `switch` può selezionare il ramo in base al tipo di un oggetto. Considera un valore che può essere testo, intero o qualcos'altro. Questo esempio è un approfondimento perché usa oggetti e tipi che studieremo con più calma, ma mostra perché un `switch` non è più limitato a una lista di numeri:

Il primo passo può essere un controllo con `instanceof`:

```
if (valore instanceof String testo) {
    System.out.println(testo.length());
}
```

Se `valore` è una `String`, la variabile `testo` contiene il riferimento già controllato nel ramo. Non serve ripetere un cast. Il nome non è disponibile ovunque: il compilatore ne determina l'ambito dai percorsi in cui il controllo può essere vero. Un `switch` con pattern estende la stessa idea a più casi:

```
String descrizione = switch (valore) {
    case null -> "assente";
    case String testo when testo.isBlank() -> "testo vuoto";
    case String testo -> "testo: " + testo;
    case Integer numero -> "intero: " + numero;
    default -> "altro valore";
};
```

Il pattern `String testo` controlla il tipo e, se il controllo riesce, rende disponibile il nome `testo` in quel ramo. La guardia `when` aggiunge una condizione. L'ordine è essenziale: il caso più specifico, quello della stringa vuota, viene prima del caso per qualunque stringa. Se li invertissimo, il primo caso assorbirebbe anche il secondo. Il caso `null` è esplicito, così il lettore sa come viene trattato il valore assente. Nel capitolo sui record vedremo anche i *record pattern*, che estraggono componenti da un record. Pattern matching per `switch` e record pattern sono funzionalità stabili nelle versioni di Java trattate qui; le eventuali estensioni preview di Java 25 restano fuori dal percorso principale, come indicato nel riepilogo ufficiale delle modifiche del linguaggio.

Nei due frammenti `valore` è dichiarato come `Object`; `SwitchEPattern.java` mostra il metodo completo e verifica quattro input diversi. Per i frammenti su `yield`, `do-while` e `break` c'è anche `FrammentiControllo.java`, così il codice può essere provato senza indovinare le dichiarazioni mancanti.

> **Approfondimento – Dominanza ed esaustività.** Un pattern più generale può rendere irraggiungibile un caso successivo: `case Object oggetto` prima di `case String testo` intercetterebbe anche le stringhe. In uno `switch` che produce un valore, ogni possibilità deve portare a un risultato; se il compilatore non può dimostrarlo, serve un caso ulteriore. I record pattern seguono lo stesso principio, ma richiedono di conoscere prima che cosa sia un record.

> **Approfondimento Java 25 – Primitivi nei pattern, ancora preview.** I casi precedenti lavorano con forme stabili. Se il valore da classificare è invece un `long` oppure occorre usare un tipo primitivo in un pattern, Java 25 offre un'estensione ancora **preview**, descritta nel riepilogo ufficiale del linguaggio. Il problema che affronta è evitare conversioni o rami separati solo perché certe forme di `switch` e `instanceof` non ammettevano quei primitivi. Non bisogna dedurne che ogni esempio con `switch` del capitolo richieda la preview: quelli già svolti compilano senza opzioni speciali. Per provare un programma che usa questa estensione servono `javac --enable-preview --release 25` e `java --enable-preview`; la sintassi e il contratto possono ancora cambiare. Per un'applicazione che non vuole dipendere dalla preview si conserva una scelta stabile, per esempio una condizione `if` su un `long`. Il criterio di scelta resta il problema da risolvere, non il desiderio di usare l'ultima forma disponibile.

Ripetere con `while` e `do-while`

Un ciclo `while` controlla la condizione **prima** di eseguire il corpo. Se è falsa dall'inizio, il corpo non viene eseguito neppure una volta. La figura segue il percorso della condizione e il ritorno al controllo dopo ogni ripetizione.

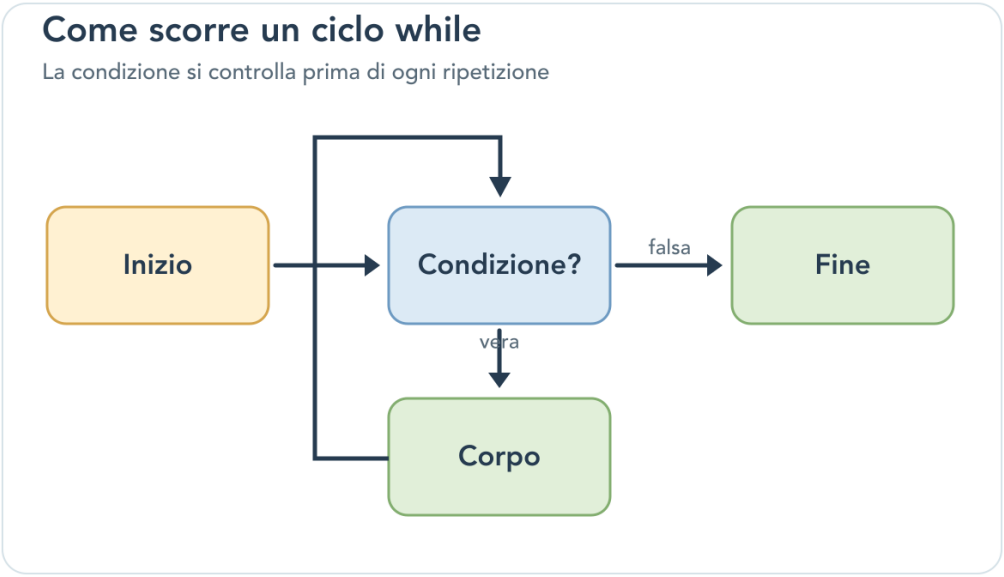


Figura 5.1 – Il corpo viene ripetuto finché la condizione è vera. La freccia di ritorno porta alla verifica, non direttamente al corpo.

```
int indice = 0;
while (indice < voti.length) {
    System.out.println(voti[indice]);
    indice++;
}
```

`indice++` aumenta l'indice di uno. Se dimentichiamo di modificarlo, la condizione può restare vera per sempre; se lo aumentiamo prima della lettura, possiamo saltare il primo voto. Un ciclo va letto come tre domande: qual è lo stato iniziale, che cosa fa terminare la ripetizione e che cosa cambia a ogni passo?

La forma `do-while` controlla la condizione **dopo** il corpo, quindi lo esegue almeno una volta. È utile, per esempio, quando chiediamo un input e poi decidiamo se ripetere la richiesta. Non usarla soltanto per evitare una riga di inizializzazione: scegliendola stai promettendo che la prima esecuzione ha senso anche se la condizione finale è falsa.

Per leggere un ciclo `while` in modo affidabile, individua una proprietà che deve restare vera a ogni giro. Nel ciclo dei voti, prima di accedere a `voti[indice]`, vogliamo `0 <= indice < voti.length`. L'indice parte da zero, la condizione controlla il limite superiore e l'incremento avviene dopo l'accesso: insieme, questi tre pezzi spiegano perché non leggiamo fuori dall'array. Se scambi l'accesso e l'incremento, la prima posizione viene saltata e l'ultima iterazione può tentare un

indice non valido. La prova con un array di un solo elemento rende subito visibile l'errore.

Un ciclo senza fine può essere intenzionale, per esempio un servizio che attende richieste, ma deve avere una strategia per interrompersi o essere arrestato. Nei programmi didattici il caso più frequente è molto più semplice: la variabile usata nella condizione non viene mai aggiornata. Quando un programma sembra bloccato, controlla prima se il corpo del ciclo modifica davvero ciò che potrebbe rendere falsa la condizione. Una stampa temporanea dell'indice può aiutare, purché poi venga tolta o sostituita con una verifica mirata.

```
int tentativi = 0;
do {
    tentativi++;
    System.out.println("Tentativo " + tentativi);
} while (tentativi < 3);
```

Il programma stampa tre righe. Il corpo esegue il primo tentativo prima di valutare `tentativi < 3`; dopo il terzo, la condizione diventa falsa. Un `while` può produrre lo stesso risultato con una diversa disposizione del controllo, ma non offre la garanzia di una prima esecuzione quando la condizione iniziale è falsa.

`for` e `for` esteso

Quando conosciamo un indice che parte da un valore, prosegue finché una condizione è vera e cambia a ogni passo, un `for` riunisce i tre elementi:

```
for (int indice = 0; indice < voti.length; indice++) {
    System.out.println(voti[indice]);
}
```

La variabile `indice` nasce nell'intestazione del ciclo. La condizione `indice < voti.length` evita l'indice fuori intervallo. Se non ci serve conoscere la posizione e vogliamo soltanto ciascun valore, il `for` esteso è più diretto:

```
int somma = 0;
for (int voto : voti) {
    somma += voto;
}
```

La lettura è «per ogni `voto` in `voti`». Il ciclo non crea una copia dell'intero array: a ogni passaggio assegna l'elemento corrente alla variabile locale. Con valori primitivi, modificare `voto` non modifica l'elemento dell'array. Per cambiare una posizione precisa serve l'indice. Il `for` esteso si applica anche a tipi *iterabili*, che incontreremo parlando di collezioni.

I due programmi storici che stampavano i mesi mostravano la stessa sequenza con `for` ordinario e con `for` esteso. Su un array `{"gennaio", "febbraio", "marzo"}`, il primo usa gli indici `0`, `1` e `2` per leggere i valori; il secondo consegna direttamente, un mese alla volta, la variabile `mese`. Se il compito è stampare tutti i nomi, la seconda forma evita un contatore che non aggiunge informazione. Se invece il compito è scrivere «mese 1: gennaio» o sostituire l'elemento in una posizione,

l'indice torna utile. Questo confronto, e non una promessa generale di maggiore velocità, è il criterio per scegliere.

La variabile del `for` esteso va interpretata secondo il tipo dell'elemento. Con `int[] voti`, `voto` riceve una copia del valore primitivo: `voto = 30;` non cambia l'array. Con un array di riferimenti a oggetti modificabili, la variabile riceve una copia del **riferimento**: riassegnarla non sostituisce l'elemento dell'array, ma chiamare un metodo che modifica l'oggetto può cambiare lo stato osservato anche attraverso l'array. Non abbiamo ancora bisogno di usare questa seconda possibilità; serve però a non trasformare la frase «il `for` esteso non modifica gli elementi» in una regola falsa. Anche la rimozione durante l'attraversamento di una collezione dipende dal tipo di collezione e dal suo iteratore: quando arriveremo alle collezioni distingueremo la lettura, la modifica dell'oggetto e la modifica della struttura.

Il `for` ordinario può anche essere letto come un `while` con inizializzazione, controllo e aggiornamento raccolti in testa. Dopo aver inizializzato `indice`, valuta la condizione prima del corpo; dopo il corpo esegue l'aggiornamento e torna al controllo. Se la condizione è falsa dall'inizio, il corpo non parte. Questa corrispondenza aiuta a scegliere la forma senza attribuirle poteri diversi: entrambe possono attraversare un array, ma `for` rende più visibile il contatore quando il numero di passi è determinato dalla lunghezza.

Con un array di array, il ciclo esterno può attraversare le righe e quello interno le posizioni della riga corrente. Il limite corretto per il secondo è `caselle[riga].length`, non necessariamente `caselle[0].length`: Java permette righe di lunghezza diversa. È esattamente ciò che ci servirà nella tavola della dama. La figura del capitolo 4 era rettangolare per chiarezza, ma il codice generale non deve dipendere da quella figura.

Ricostruiamo la traccia di un `while` annidato con due contatori su un caso piccolo, per non perdere il significato in decine di righe stampate. Se `riga` parte da `0` e il ciclo esterno prosegue finché `riga < 2`, mentre a ogni giro `colonna` riparte da `0` e il ciclo interno prosegue finché `colonna < 3`, le coppie visitate sono `(0,0)`, `(0,1)`, `(0,2)`, `(1,0)`, `(1,1)` e `(1,2)`. La riga non cambia durante i tre passi interni; la colonna viene azzerata quando si passa alla riga successiva. Dimenticare quell'azzeramento farebbe saltare tutta la seconda riga. Il programma della dama applica lo stesso ragionamento con due `for` e otto posizioni per riga.

Un `for` può avere anche due contatori nella sua intestazione, come `for (int sinistra = 0, destra = 4; sinistra < destra; sinistra++, destra--)`. Su un array di cinque elementi visita le coppie di indici `(0,4)` e `(1,3)` e poi si ferma quando entrambi arrivano a `2`. Le virgole separano le due dichiarazioni dello stesso tipo e le due espressioni di aggiornamento; la condizione centrale resta una sola espressione booleana. È utile quando un algoritmo guarda contemporaneamente le due estremità di una sequenza, per esempio per controllare se una parola si legge allo stesso modo in entrambi i versi. Se due contatori non condividono una relazione così chiara, spostarli in istruzioni separate evita di trasformare l'intestazione del ciclo in un rompicapo.

La forma `for (;;)` omette inizializzazione, condizione e aggiornamento; non essendoci una condizione falsa che la termini, il ciclo continua finché un `break`, un `return` o un'interruzione eccezionale non cambia il flusso. È lecita, come `while (true)`, ma richiede che il percorso di uscita sia evidente. Omettere soltanto l'aggiornamento e poi dimenticare di modificare il contatore nel corpo è invece un

errore comune. La flessibilità di `for` non la rende automaticamente la scelta migliore in ogni problema: per «ripeti finché arriva un evento» un `while` può raccontare più direttamente l'intenzione.

Ecco il programma della media dopo la revisione. La verifica dell'array vuoto avviene prima della divisione; la somma funziona con qualunque lunghezza positiva:

```
public class MediaVotiConCiclo {
    public static void main(String[] args) {
        int[] voti = {24, 27, 30};
        if (voti.length == 0) {
            System.out.println("Nessun voto disponibile");
            return;
        }

        int somma = 0;
        for (int voto : voti) {
            somma += voto;
        }
        double media = (double) somma / voti.length;
        System.out.println("Media: " + media);
    }
}
```

`return` termina qui il metodo `main` quando non ci sono voti. Il caso normale prosegue fino alla stampa. L'output per i tre voti dell'esempio è ancora `Media: 27.0`; se aggiungi un quarto voto, non devi riscrivere la somma.

Tornare alla tavola della dama

Nel capitolo sugli array avevamo lasciato in sospeso una tavola di otto righe e otto colonne. Le caselle utilizzabili sono alternate; nelle prime tre righe mettiamo le pedine bianche, nelle ultime tre le nere. Con due cicli `for` annidati possiamo applicare la stessa regola a ogni posizione, invece di copiare molte assegnazioni simili. L'operatore `%` ci dice se la somma degli indici è pari o dispari.

```
public class DamaIniziale {
    public static void main(String[] args) {
        String[][] caselle = new String[8][8];
        int bianche = 0;
        int nere = 0;

        for (int riga = 0; riga < caselle.length; riga++) {
            for (int colonna = 0; colonna < caselle[riga].length; colonna++) {
                boolean giocabile = (riga + colonna) % 2 == 1;
                if (giocabile && riga < 3) {
                    caselle[riga][colonna] = "bianca";
                    bianche++;
                } else if (giocabile && riga >= 5) {
                    caselle[riga][colonna] = "nera";
                    nere++;
                }
```

```
            }
        }

        System.out.println("Bianche: " + bianche + ", nere: " + nere);
    }
}
```

Le due righe centrali restano senza pedine; anche le caselle non utilizzabili rimangono `null`. La condizione `giocabile` evita di collocare una pedina in ogni casella della riga. Il risultato atteso è `Bianche: 12, nere: 12`: tre righe per colore, quattro caselle utilizzabili in ogni riga. Questo controllo numerico è piccolo, ma scopre subito un errore nella parità delle caselle o nel confine fra le righe.

`break`, `continue` e `return`

`break` interrompe il ciclo o lo `switch` che lo contiene nella forma in cui è previsto. `continue` salta il resto dell'iterazione corrente e torna al controllo o all'aggiornamento del ciclo. `return` esce dal metodo e, se il metodo produce un risultato, restituisce quel valore. Sono tre trasferimenti di controllo diversi. Prima di usarli, chiediti che cosa deve accadere dopo: ignorare un solo elemento, terminare una ricerca o terminare l'intera operazione?

Per esempio, durante una ricerca in un array un `break` può fermare il ciclo appena troviamo l'elemento richiesto. Se invece vogliamo ignorare i valori negativi e continuare la somma dei successivi, `continue` può esprimere la scelta, anche se talvolta un `if` rende il codice più semplice. Etichette e interruzioni di cicli annidati esistono, ma in un programma molto annidato è spesso più chiaro estrarre un metodo con un nome significativo.

Nella forma classica di `switch`, `break` interrompe lo `switch`; in un ciclo interrompe il ciclo più interno che lo contiene. `continue` non serve a «passare al prossimo case» di uno `switch`: riguarda un ciclo e ne avvia la successiva iterazione secondo le sue regole. `return` termina il metodo corrente, anche se compare dentro un ciclo. Prima di usare uno dei tre, nomina mentalmente il costrutto da cui vuoi uscire. Se la risposta è vaga, la riga sarà difficile da mantenere.

Consideriamo un esempio con `continue` che conta i numeri pari da 0 a 20. Vale la pena verificare i confini, perché un contatore inizializzato a `-1` e incrementato all'inizio del corpo visita prima `0` e poi, all'ultimo giro, anche `20`: i pari contati sono **undici**, non dieci. Se volevamo l'intervallo da 1 a 20, era meglio cominciare da `1` in un `for` con condizione `numero <= 20`; i pari diventano dieci. Una descrizione come «tra 1 e 20» deve quindi specificare se gli estremi sono inclusi e trovare riscontro nel codice. `continue` può saltare l'incremento del contatore se lo collochiamo nel punto sbagliato di un `while`, creando un ciclo senza fine; nel `for` l'aggiornamento dell'intestazione viene eseguito prima del controllo successivo.

Quando un metodo è dichiarato `void`, può terminare con `return;` senza valore, come `main` nel caso dell'array vuoto. Se il metodo promette un risultato, come `int trovaPrimoNegativo(int[] valori)`, i percorsi che terminano normalmente devono restituire un valore compatibile: `return indice;` quando trovano una posizione e `return -1;` al termine della ricerca. Un `break` al posto di quel primo `return` uscirebbe soltanto dal ciclo e richiederebbe ancora un'istruzione per consegnare il risultato al chiamante. Distinguere questi due livelli di uscita rende più semplice

anche la gestione degli errori: un'eccezione non è un sostituto generico di `break` quando vogliamo soltanto smettere di cercare.

```
int sommaNonNegativi = 0;
for (int numero : new int[] {5, -2, 3}) {
    if (numero < 0) {
        continue;
    }
    sommaNonNegativi += numero;
}
System.out.println("Somma non negativi: " + sommaNonNegativi);
```

Il risultato è `8`. Quando `numero` vale `-2`, `continue` salta soltanto l'addizione di quell'iterazione: il ciclo passa poi a `3`. Non termina il ciclo e non esce dal metodo. Il frammento è eseguibile nel file di supporto `FrammentiControllo.java`.

```
int primoNegativo = -1;
for (int indice = 0; indice < valori.length; indice++) {
    if (valori[indice] < 0) {
        primoNegativo = indice;
        break;
    }
}
```

Qui `-1` significa «nessuna posizione trovata», perché gli indici validi partono da zero. Appena il ciclo incontra un valore negativo conserva l'indice e si ferma; se non ne incontra, il valore iniziale resta invariato. Se il ciclo fosse annidato in un altro, questo `break` fermerebbe soltanto il ciclo che lo contiene direttamente. Per uscire da tutto il metodo potremmo progettare un metodo di ricerca che usa `return`, invece di dipendere da un'etichetta difficile da seguire.

Prova tu

Parti da `MediaVotiConCiclo` e prova tre casi: tre voti, quattro voti e un array vuoto. Prima di eseguire il programma scrivi che cosa ti aspetti. Il lavoro è completato quando il caso vuoto non provoca una divisione per zero, gli altri due calcolano la media corretta e sai spiegare quale istruzione impedisce di accedere a una posizione fuori intervallo. Come autoverifica, confronta `while` e `do-while`: quale dei due può non eseguire mai il corpo, e perché?

6. Package e moduli

Dare una casa alle classi

Finora i nostri programmi sono stati piccoli: un file, una classe, una cartella di lavoro. In un'applicazione reale incontrerai più classi con nomi simili, biblioteche di codice e persone che lavorano su parti diverse. Serve un modo per dare a ciascuna classe un posto riconoscibile. I **package** organizzano i nomi; i **moduli** permettono di dichiarare quali package vengono esposti e da quali altri moduli dipende un gruppo di codice. Sono due livelli distinti, e partiremo da quello più semplice.

> **Definizione – Package.** Un package è uno spazio di nomi per classi e interfacce. Il nome completo di una classe unisce il nome del package e quello della classe, per esempio `java.util.ArrayList`. Il punto separa le parti del nome; nei progetti che usano file e cartelle, gli strumenti riflettono di solito questa gerarchia nel percorso dei sorgenti.

Dichiarare e nominare un package

Mettiamo la classe `Catalogo` nel package `javamattone.esercizi.capitolo6`. La prima riga significativa del sorgente è:

```
package javamattone.esercizi.capitolo6;
```

La dichiarazione identifica il package della classe. La cartella che useremo per il sorgente `Catalogo.java` segue la stessa struttura: `javamattone/esercizi/capitolo6/Catalogo.java`. Questo accordo fra nome e percorso rende semplici compilazione, ricerca e manutenzione. Il punto, da solo, non crea cartelle: il filesystem e gli strumenti di build devono trovare i file nella disposizione attesa.

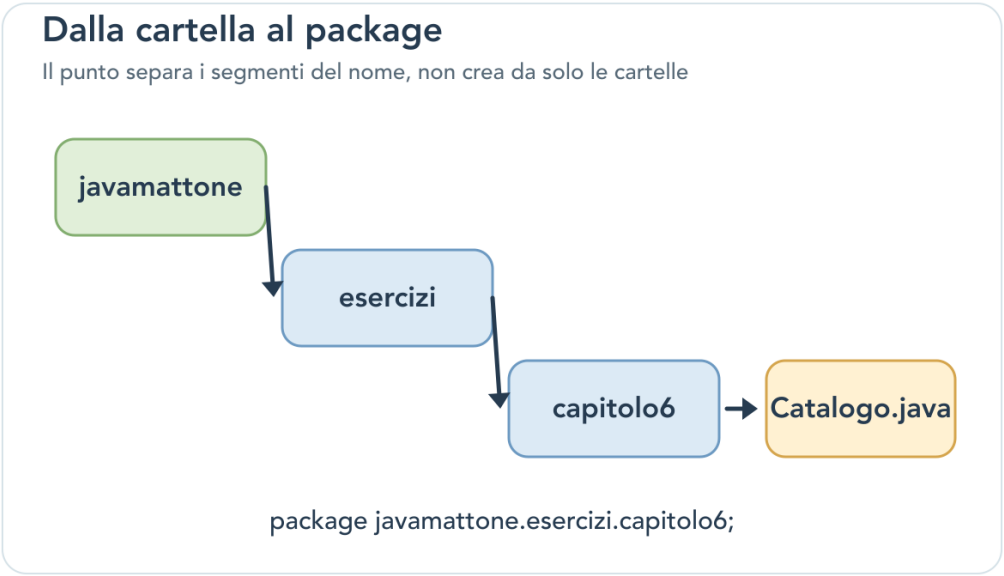


Figura 6.1 – Il nome del package e la gerarchia dei sorgenti si corrispondono nel nostro progetto. La classe `Catalogo` è collocata nella cartella finale.

Un package con un nome generico come `util` rischia di entrare in conflitto con un package di un altro progetto. Per codice destinato a essere distribuito si usa spesso un prefisso legato a un dominio Internet controllato dall'organizzazione, scritto in ordine inverso, per esempio `it.esempio.catalogo`. Non è una verifica automatica della proprietà del dominio: è una convenzione che aiuta a scegliere nomi unici. Dentro un package, usa parole che descrivono una responsabilità; `it.esempio.catalogo.prestiti` dice più di `it.esempio.varie`.

> **Nota bene – Un punto non dà privilegi.** `it.esempio.catalogo` e `it.esempio.catalogo.interno` sono package distinti. Il secondo non ottiene per questo un accesso speciale alle classi del primo. La specifica dei package e dei moduli descrive con precisione nomi e visibilità.

Il nome di package viene scelto una volta per una parte del codice e poi compare in più luoghi: all'inizio del sorgente, negli import di chi lo usa, nei comandi che nominano una classe principale e spesso nelle cartelle. Cambiarlo non equivale a rinominare soltanto una directory. Se `Catalogo.java` dichiara ancora `package javamattone.esercizi.capitolo6;` ma lo sposti sotto `it/esempio/catalogo`, il nome dichiarato della classe resta `javamattone.esercizi.capitolo6.Catalogo`. Il percorso scelto può far fallire o complicare la ricerca dei sorgenti, e il file compilato con `-d` seguirà comunque il package dichiarato.

Un albero di package può aiutare a vedere che il punto separa segmenti del nome. Pensalo come a un indirizzo logico, non come a una relazione di parentela fra package. `java.util` e `java.util.concurrent` condividono un prefisso ma sono due spazi di nomi distinti. Se una classe è accessibile soltanto nel proprio package, aggiungere `.interno` al nome del chiamante non le apre la porta. È lo stesso

principio che abbiamo usato nel capitolo sull'incapsulamento per decidere chi può leggere un membro.

Trovare e distribuire le classi

Per compilare `Catalogo.java` dalla radice dei sorgenti possiamo usare:

```
javac --release 25 -d build javamattone/esercizi/capitolo6/Catalogo.java
java -cp build javamattone.esercizi.capitolo6.Catalogo
```

L'opzione `-d build` chiede al compilatore di mettere i file `.class` sotto `build`, ricreando le sottocartelle del package.

Nell'avvio usiamo il **nome qualificato** della classe, non il percorso con le barre. `-cp build` indica il *class path*: l'insieme delle posizioni da cui il comando `java` può cercare le classi dell'applicazione in questa forma di avvio. Se il comando non trova la classe, controlla insieme dichiarazione `package`, percorso del file prodotto, class path e nome passato a `java`.

Facciamo una diagnosi passo dopo passo, come faremmo davanti a un messaggio d'errore vero. Dopo la compilazione, nella directory `build` deve esserci `javamattone/esercizi/capitolo6/Catalogo.class`. Se invece passi a `java` il percorso `javamattone/esercizi/capitolo6/Catalogo.class`, stai fornendo un nome di file dove il comando si aspetta un nome di classe. Se scrivi soltanto `Catalogo`, il runtime cercherà quel nome senza il package e non troverà la classe appena compilata. Se scrivi il nome completo ma imposti `-cp` sulla cartella `build/javamattone`, la radice della ricerca è troppo in basso. La radice giusta è la directory **sopra** il primo segmento del package, cioè `build` nel nostro esempio. Quattro stringhe possono sembrare quasi uguali e indicare invece quattro cose diverse: percorso del sorgente, percorso del file compilato, radice del class path e nome qualificato.

La variabile d'ambiente `CLASSPATH` può fornire un valore predefinito, ma per imparare è più chiaro passare `-cp` nel comando che stiamo provando: la dipendenza dal percorso resta visibile accanto al nome della classe. Chi riproduce l'esempio non deve indovinare una configurazione fatta mesi prima sul nostro computer. Nei progetti più grandi uno strumento di build costruirà per noi il class path; anche allora, capire quale sia la sua radice aiuta a diagnosticare un `ClassNotFoundException` o un avvio che cerca la classe sbagliata.

Gli archivi JAR permettono di raccogliere classi e risorse in un file. Un JAR può avere un `META-INF/MANIFEST.MF` con informazioni come la classe principale, così da avviare l'archivio con `java -jar`. Il manifest non sostituisce la dichiarazione `package`, non trasforma da solo il programma in un modulo e non modifica la visibilità delle classi. Il manuale di `jar` descrive come creare e ispezionare questi archivi; nel capitolo sugli strumenti prepareremo una distribuzione completa.

Il JAR e il manifest, letti senza magia

Il manifest può indicare al runtime che cosa avviare e contenere altri metadati dell'archivio distribuito. Il punto utile per il nostro esempio è concreto: se l'archivio contiene `javamattone/esercizi/capitolo6/Catalogo.class`, la voce `Main-Class` del manifest deve nominare `javamattone.esercizi.capitolo6.Catalogo`, con i punti, senza `.class`. Se il nome è sbagliato, l'archivio può essere valido ma `java -jar` non trova il punto di ingresso richiesto.

Puoi ispezionare il contenuto di un JAR con `jar --list --file catalogo.jar`. Vedere il percorso della classe nell'archivio aiuta a diagnosticare un errore di avvio. Un JAR può includere anche file di dati, come configurazioni o immagini, che non sono classi Java. L'archivio non incorpora automaticamente tutte le librerie di cui il programma dipende; come procurarle e distribuirle è una scelta di progetto. Non bisogna confondere «sono riuscito a creare un file `.jar`» con «ho preparato una distribuzione completa e riproducibile».

Possiamo completare la prova del nostro `Catalogo` senza lasciare il manifest come promessa astratta. Dopo aver compilato la classe nella directory `build`, eseguiamo dalla stessa directory di lavoro:

```
jar --create --file catalogo.jar --main-class javamattone.esercizi.capitolo6.Catalogo -C build .
jar --list --file catalogo.jar
java -jar catalogo.jar
```

L'opzione `-C build .` fa entrare nell'archivio il contenuto di `build` prendendo quella cartella come radice. Perciò il percorso interno della classe comincia con `javamattone/`, non con `build/`. `--main-class` fa scrivere allo strumento `jar` la voce `Main-Class` nel manifest con il nome qualificato corretto; non occorre inventare a mano gli spazi o la terminazione delle righe del file. La seconda istruzione permette di controllare che `META-INF/MANIFEST.MF` e la classe siano presenti, la terza avvia l'archivio e stampa `Java Mattone dopo Mattone`. Se la directory `build` contenesse classi di altri esperimenti, entrerebbero anch'esse nell'archivio: per una distribuzione reale conviene usare una cartella di compilazione pulita e dichiarare tutte le dipendenze necessarie.

Il manifest può contenere anche informazioni collegate alla sicurezza. Un manifest può trasportare attributi e metadati di firma, ma la presenza di una voce di testo non rende affidabile un archivio. Per controllare provenienza e integrità servono strumenti e verifiche appropriate alla distribuzione scelta. Qui usiamo il manifest per l'avvio e rimandiamo firma e catena di fiducia alla parte sulla distribuzione sicura, dove potremo trattarle senza suggerire che basti scrivere una riga in un file.

`import`: un nome più corto nel sorgente

La classe `ArrayList` appartiene a `java.util`. Possiamo scrivere `java.util.ArrayList` ogni volta che la usiamo oppure aggiungere una dichiarazione `import` dopo `package` e prima della classe:

```
package javamattone.esercizi.capitolo6;

import java.util.ArrayList;

public class Catalogo {
    private Catalogo() {
    }

    public static void main(String[] args) {
        ArrayList<String> titoli = new ArrayList<>();
        titoli.add("Java Mattone dopo Mattone");
        System.out.println(titoli.get(0));
    }
}
```

Questo programma crea una lista di titoli, ne aggiunge uno e lo stampa. Il piccolo costruttore `private Catalogo()` segnala che qui la classe serve soltanto come punto di avvio e non deve essere istanziata; il significato dei costruttori sarà sviluppato nel capitolo sulle classi. `ArrayList<String>` indica che la lista contiene stringhe; il capitolo sulle collezioni spiegherà il tipo parametrizzato e i suoi metodi. Qui ci interessa il nome: grazie a `import java.util.ArrayList;` possiamo scrivere `ArrayList` nel resto del file. La classe rimane nel package `java.util`; l'import non la copia, non la sposta e non la rende pubblica se non lo era già.

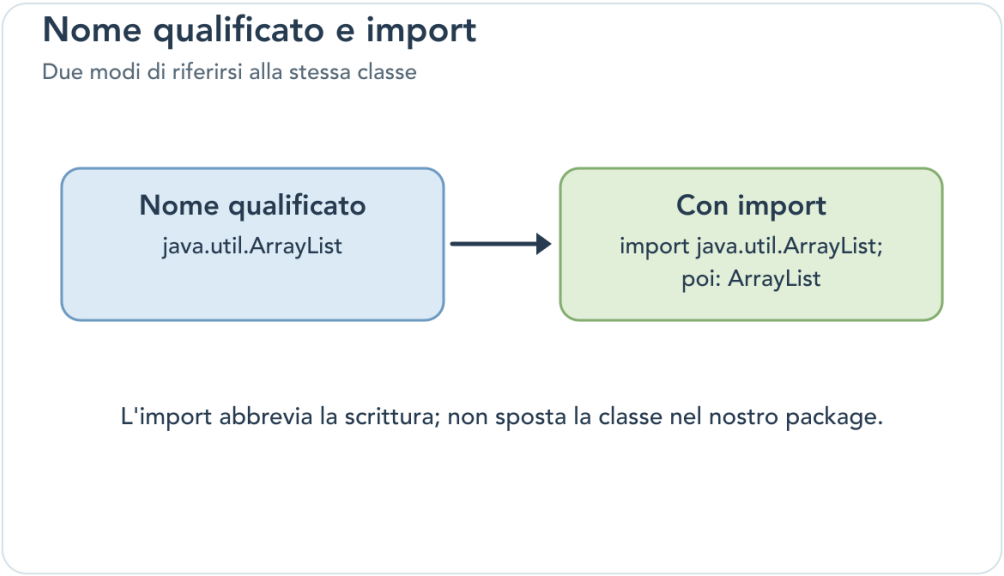


Figura 6.2 – L'import abbrevia il riferimento nel sorgente. Il nome qualificato della classe continua a essere `java.util.ArrayList`.

`import java.util.*;` rende disponibili per nome semplice i tipi accessibili direttamente in quel package, ma non quelli dei suoi sottopackage. Per gli esempi del libro preferiremo import espliciti: chi legge vede subito da dove arriva una classe e un conflitto di nomi è più facile da diagnosticare. I tipi di `java.lang`, come `String`, sono disponibili senza import esplicito.

Un import non equivale nemmeno a un'istruzione di caricamento a runtime. Il compilatore lo usa per risolvere un nome scritto nel sorgente. Se cancelli l'import di `ArrayList` e sostituisci ogni occorrenza con `java.util.ArrayList`, il significato della classe usata non cambia. Se invece mantieni il nome breve senza import e senza una classe omonima disponibile nello stesso package, il compilatore non riesce a determinarne il tipo. È una differenza piccola ma fondamentale quando si leggono messaggi come «cannot find symbol».

Consideriamo due classi, `Autore` e `StampaNomeAutore`: una rappresenta l'autore, l'altra ne stampa il nome. Ci interessa la possibilità di riferirsi alla stessa classe in tre modi. Se `StampaNomeAutore` si trova nello **stesso package** di `Autore`, può usare il nome semplice senza alcun import. Se si trova in un altro package e la classe `Autore` è accessibile, può scrivere il nome qualificato in ogni punto in cui serve. Può infine dichiarare un `import` e usare poi il nome semplice. Nessuna delle tre forme crea una seconda classe `Autore`: cambiano soltanto le regole con cui il compilatore riconosce il nome scritto nel sorgente.

La parola «accessibile» in questo confronto è essenziale. Un import non rende pubblica una classe che non lo è. Una classe dichiarata senza il modificatore `public` può essere usata soltanto secondo le regole di accesso del suo package; spostare `StampaNomeAutore` in un package vicino, anche con lo stesso prefisso, non supera quel confine. Nemmeno il carattere `*` crea un'eccezione: `import it.esempio.catalogo.*;` semplifica alcuni nomi, ma non cambia le dichiarazioni di visibilità e non comprende automaticamente `it.esempio.catalogo.interno`. Ecco perché, quando un import sembra corretto ma il compilatore protesta, controlliamo sia il nome sia il permesso di accesso.

Esiste anche l'import di membri `static`, che permette di usare direttamente il nome di una costante o di un metodo statico. Un esempio comune è importare `Math.PI`; senza quell'import scriviamo `Math.PI`, e il prefisso ricorda immediatamente da dove arriva il valore. Per i primi esempi preferiamo il nome qualificato quando rende la riga più chiara. L'uso di `*` negli import non significa «copia tutto il package» e non importa i sottopackage, sia nella forma normale sia in quella statica.

> **Buona pratica – Se un nome è ambiguo, usa quello completo.** Due package possono contenere classi con lo stesso nome semplice. Un import non cambia la loro identità. In un punto dove serve distinguerle, scrivere il nome qualificato è più chiaro che inventare un trucco per evitare il conflitto.

Un passo più grande: i moduli

Un package organizza nomi; un **modulo** raggruppa package e dichiara le proprie dipendenze e le parti accessibili dall'esterno. Dal JDK 9 il sistema dei moduli fa parte della piattaforma. Non devi trasformare ogni esercizio in un modulo: un piccolo programma sul class path è un punto di partenza sensato. Devi però sapere che i due modi di avvio esistono, perché molti strumenti e librerie moderni li incontrano.

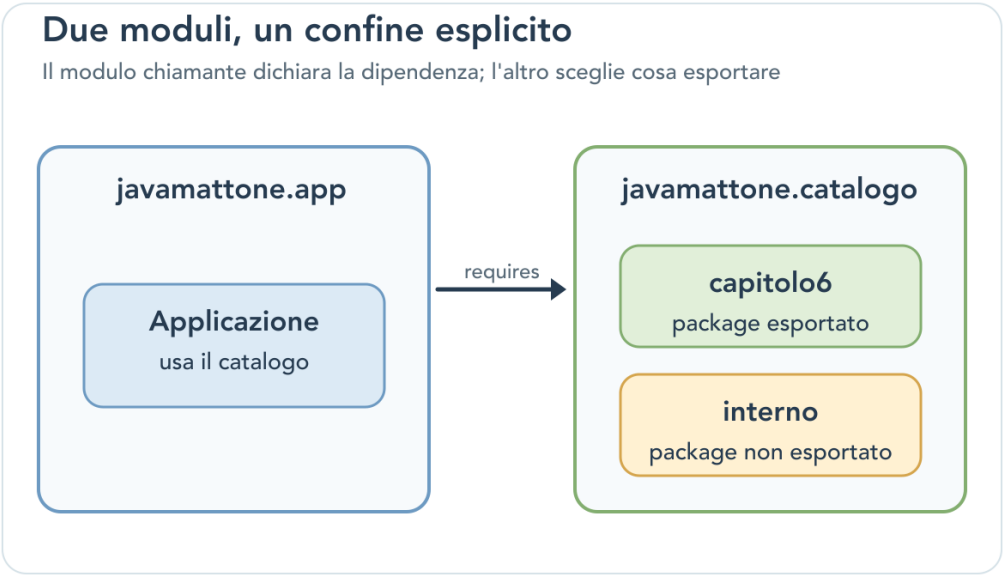


Figura 6.3 – `javamattone.app` legge `javamattone.catalogo`, che esporta il package del capitolo 6. Il nome `capitolo6` nella figura abbrevia `javamattone.esercizi.capitolo6`; il package `interno` rappresenta un'eventuale parte non esportata. Un semplice `import` non renderebbe accessibile quella parte dall'applicazione esterna.

Il descrittore di un modulo si chiama `module-info.java`. Per un modulo di esempio che vuole rendere pubblico il package del catalogo, potrebbe contenere:

```
module javamattone.catalogo {
    exports javamattone.esercizi.capitolo6;
}
```

`exports` rende accessibili a moduli che leggono `javamattone.catalogo` le classi pubbliche di quel package. Non rende pubbliche le classi o i membri privati, e non esporta automaticamente altri package. Se il modulo dipendesse da un modulo diverso da `java.base`, dichiarerebbe la dipendenza con `requires`. `java.base` è letto implicitamente da ogni modulo, quindi non occorre scrivere `requires java.base;` nel nostro esempio.

La compilazione e l'avvio modulari usano un **module path**, separato dal class path. Con i file nella struttura mostrata nei materiali del capitolo, la forma essenziale dell'avvio è `java --module-path mods -m javamattone.catalogo/javamattone.esercizi.capitolo6.Catalogo`: prima il nome del modulo, poi quello qualificato della classe principale. La sezione sui moduli non è un invito a memorizzare subito la riga di comando; serve a capire che il nome di un package e il nome di un modulo rispondono a domande diverse.

> **Concetto chiave – Accessibilità in due passaggi.** Una classe pubblica di un package non è necessariamente accessibile da un altro modulo. Il package deve essere esportato e il modulo chiamante deve leggere quello che lo esporta. Il descrittore rende visibili queste scelte che, con il solo class path, restano più implicite.

Il sistema dei moduli prevede anche servizi: un modulo può dichiarare che usa un servizio e un altro può offrire un'implementazione. L'API `ServiceLoader` permette di trovare implementazioni disponibili senza fissarne il nome nel codice che le usa. È un argomento utile quando progetteremo componenti sostituibili; per ora basta sapere che `requires` ed `exports` non sono le uniche dichiarazioni possibili. Lo strumento `jlink`, inoltre, può costruire un'immagine di runtime con i moduli necessari a un'applicazione: lo incontreremo nella parte dedicata alla distribuzione.

Un import nuovo in Java 25

Java 25 rende stabile anche la dichiarazione `import module`. Per esempio, `import module java.base;` permette di usare tramite nome semplice i tipi dei package esportati da quel modulo, secondo le regole della documentazione ufficiale. È una comodità del **sorgente** per importare tipi; non sostituisce la dichiarazione `requires` nel descrittore di un modulo. Nei programmi piccoli e didattici continueremo spesso a importare una classe per volta, perché rende immediata la provenienza di un nome.

> **Versione Java – `import module`.** La sintassi fa parte del linguaggio stabile in Java 25. Se compili per una release precedente, il file che la usa potrebbe non essere accettato. Gli esempi del percorso principale dichiarano il requisito di versione quando introducono una forma nuova.

Per concludere

Un package dà un nome alle classi correlate, un import abbrevia il modo in cui le nominiamo dentro un file, un JAR può raccogliere file compilati e risorse, un modulo dichiara dipendenze e confini di accesso. Nessuno di questi strumenti ripara automaticamente un modello confuso; servono a rendere leggibili e verificabili le scelte che abbiamo fatto.

Per esercitarti, sposta una copia di `Catalogo.java` nel package `it.esempio.catalogo`, aggiorna cartelle e comandi e verifica che il programma stampi ancora il titolo. L'attività è riuscita quando sai indicare quale riga cambia il nome qualificato della classe, quale opzione indica dove cercare i file compilati e perché la sola dichiarazione `import` non modifica il package di `ArrayList`.

7. Definizione di classi e oggetti

Dal modello al programma

Nel capitolo 2 abbiamo immaginato una biblioteca fatta di libri, lettori e prestiti. Finora erano soprattutto nomi con cui ragionare. Come diventa una di quelle idee un programma Java? Partiamo da un libro che conosce il proprio titolo e sa contare quante volte è stato preso in prestito. È un esempio piccolo: ci permetterà di vedere una classe, creare un oggetto e chiamare un metodo senza confondere le tre operazioni.

Una **classe** è una dichiarazione: descrive quali dati e operazioni avranno gli oggetti creati secondo quella definizione. Un **oggetto**, o *istanza*, è un'entità concreta del programma. Possiamo dichiarare la classe `Libro` una volta e creare più oggetti `Libro`, ciascuno con il proprio titolo e il proprio contatore. La differenza era già comparsa nella definizione del capitolo 2; adesso la vediamo nel codice.

```
public class Libro {
    private final String titolo;
    private int prestiti;

    public Libro(String titolo) {
        this.titolo = titolo;
    }

    public void registraPrestito() {
        prestiti++;
    }

    public String descrizione() {
        return titolo + " - prestiti: " + prestiti;
    }


    public static void main(String[] args) {
        Libro primo = new Libro("Java Mattone dopo Mattone");
        Libro secondo = primo;
        primo.registraPrestito();
        System.out.println(secondo.descrizione());
        System.out.println(primo == secondo);
    }
}
```

Salva il file come `Libro.java`. Nella cartella che lo contiene, `javac --release 25 -d build Libro.java` produce la classe compilata e `java -cp build Libro` la avvia. Le due righe stampate sono `Java Mattone dopo Mattone - prestiti: 1` e `true`. Se il risultato ti sorprende, torna alla riga `Libro secondo = primo;`: è il centro dell'esempio.

Campi e metodi

Dentro `Libro`, `titolo` e `prestiti` sono **campi**: conservano dati associati a ogni istanza. Il titolo viene assegnato quando nasce il libro e, grazie a `final`, non viene riassegnato più tardi. Il contatore invece cambia quando registriamo un prestito. La parola `private` impedisce a codice esterno di accedere direttamente a questi campi; nel prossimo capitolo vedremo perché questo aiuta a mantenere regole valide, e perché da solo non basta.

`registraPrestito` e `descrizione` sono **metodi**. Un metodo dà un nome a un'operazione e contiene le istruzioni necessarie a eseguirla. Il primo non restituisce un valore, perciò dichiara `void`. Il secondo restituisce una `String`: l'istruzione `return` consegna la descrizione al punto in cui il metodo è stato chiamato. In `secondo.descrizione()`, il punto seleziona un metodo dell'oggetto a cui si riferisce `secondo`. La stringa descrive lo stato al momento della chiamata; se registrassimo un altro prestito, una nuova chiamata produrrebbe un testo diverso.

Il testo fra parentesi nella dichiarazione di un metodo è l'elenco dei **parametri formali**. Nel costruttore `Libro(String titolo)`, il parametro formale è il nome locale che riceve il valore passato da `new Libro("Java Mattone dopo Mattone");` quella stringa è l'**argomento** della chiamata. Distinguere i due nomi aiuta a leggere il codice: il parametro appartiene alla dichiarazione del metodo, l'argomento alla singola chiamata.

> **Approfondimento – Un numero variabile di argomenti.** Un parametro scritto `String... titoli` permette di chiamare un metodo con zero o più stringhe. All'interno del metodo, `titoli` si usa come un array: `titoli.length` dice quanti valori sono arrivati. Il parametro variabile deve essere l'ultimo della lista e ce ne può essere soltanto uno. È utile quando il numero dei valori cambia davvero; non serve a evitare di progettare un insieme di dati coerente. La specifica dei metodi definisce questa sintassi.

Il blocco `main` è anch'esso un metodo, ma dichiara `static`: il comando `java` può usarlo come punto di ingresso senza che noi dobbiamo prima creare esplicitamente un `Libro`. Invece `registraPrestito` lavora sul contatore di **un particolare** libro. Non possiamo chiamarlo come `Libro.registraPrestito()` perché non saprebbe quale oggetto aggiornare. La distinzione fra membro di classe e membro di istanza è più importante della parola chiave da memorizzare.

Leggere una dichiarazione di metodo per intero

Prendiamo `public String descrizione()`. `public` indica che il metodo può essere chiamato anche dal codice che usa la classe; `String` indica il tipo del valore restituito; `descrizione` è il nome; le parentesi vuote dicono che la chiamata non richiede argomenti. Il corpo comincia con `{` e termina con `}`. Se promettiamo un risultato di tipo `String`, ogni percorso normale che completa il metodo deve produrre un valore compatibile tramite `return`. Scrivere soltanto `return;` non basterebbe. In `public void registraPrestito()`, invece, `void` dice che non c'è un valore da consegnare.

Nella dichiarazione `Libro(String titolo)` compare un **tipo** seguito dal **nome del parametro**. Nella chiamata `new Libro("Java Mattone dopo Mattone")` compare

l'argomento concreto. Un metodo può ricevere più parametri separati da virgole, per esempio `prestaA(String lettore, int giorni)`. Ogni argomento deve essere compatibile con il parametro che occupa la stessa posizione. Il nome `giorni` non obbliga il chiamante a passare un numero sensato: se i valori negativi non hanno significato, il metodo dovrà controllarli.

Il numero di argomenti richiesti da un metodo è chiamato *arità*. Con `String... argomenti` dichiariamo un metodo a numero variabile di argomenti: il chiamante può passare zero o più stringhe. Dentro il metodo la variabile si usa come un array, ma chi chiama può scrivere gli argomenti separatamente. Il vantaggio è reale quando il numero dei valori cambia; la difficoltà è che una lista lunga e anonima non dice quale ruolo abbia ogni elemento. Per un prestito, un oggetto che raccoglie i dati del lettore e della durata è spesso più chiaro di molti argomenti posizionali.

> **Nota bene – Un campo non è una variabile locale.** `prestiti` appartiene a ogni istanza di `Libro`. Se crei due libri, ciascuno ha il proprio contatore. Una variabile dichiarata nel corpo di `descrizione`, invece, nasce per quella chiamata e non è un campo condiviso con le chiamate successive. Il fatto che entrambi siano scritti con un tipo e un nome non li rende intercambiabili.

Dal metodo isolato alla classe completa

Partiamo dalla forma generale di un metodo, poi collochiamolo nella classe: leggere la dichiarazione è più utile che imparare un elenco di simboli a memoria. In Java una dichiarazione di metodo si trova nel corpo di una classe, di un'interfaccia o di un'altra struttura che può contenerla: non possiamo mettere `descrizione()` da sola, fuori da ogni tipo, come faremmo con una funzione libera in altri linguaggi. Dentro il corpo di `Libro` possono convivere campi, costruttori e metodi. Le parentesi graffe esterne delimitano la classe; quelle interne delimitano il corpo del singolo metodo. Quando il compilatore incontra un'istruzione, il suo posto nel sorgente contribuisce quindi a stabilire quali nomi possa usare.

Una classe è un *modello* comune, ma i valori dei campi di istanza non sono comuni a tutti gli oggetti. Se scriviamo `Libro manuale = new Libro("Java");` e poi `Libro romanzo = new Libro("Il sentiero");`, il metodo `descrizione()` è definito una sola volta nel sorgente, mentre il campo `titolo` di ciascuna istanza ha un valore diverso. È utile chiedersi, per ogni dichiarazione, se descrive ciò che **ogni libro possiede** o ciò che **la classe intera condivide**. Il primo caso porta normalmente a un campo o metodo di istanza; il secondo può giustificare `static`. La differenza non dipende dal numero di variabili che useremo per raggiungere l'oggetto: due variabili possono indicare lo stesso libro.

> **Nota bene – `var` ha un posto preciso.** Nei campi di `Libro` scriviamo il tipo, per esempio `private int prestiti;`: `var` non si usa per dichiarare un campo. In un metodo possiamo invece scrivere `var copia = new Libro("Java");`, perché l'inizializzatore permette di dedurre il tipo della variabile locale. `var assente = null;` non compila: da `null` da solo il compilatore non può inferire il tipo richiesto. Se vogliamo rappresentare l'assenza di un libro, dichiariamo esplicitamente `Libro assente = null;` e poi decidiamo se quel valore è lecito nel contratto. Riunire i campi all'inizio della classe è una convenzione che spesso aiuta la lettura, perché facilita la ricerca dei membri; Java non impone quell'ordine, e la coerenza del progetto conta più di una posizione fissa.

Il valore restituito da un metodo è distinto dai messaggi che il metodo stampa. `descrizione()` consegna una `String` al chiamante: quest'ultimo può stamparla, salvarla, confrontarla o passarla a un altro metodo. Un metodo che eseguisse internamente `System.out.println(...)` produrrebbe invece un effetto sul flusso di uscita, ma non darebbe automaticamente al chiamante il testo da riusare. Quando scriviamo una prima versione del programma le due forme sembrano equivalenti perché vediamo la stessa riga sullo schermo; appena dobbiamo comporre una pagina, fare una prova automatica o mostrare il risultato in un'interfaccia grafica, la differenza diventa decisiva.

> **Per capire – Argomenti e parametri non sono gli stessi nomi.** In `prestaA("Ada", 14)`, `"Ada"` e `14` sono gli argomenti della chiamata. In `prestaA(String lettore, int giorni)`, `lettore` e `giorni` sono i parametri formali disponibili nel corpo. Il chiamante può aver ottenuto quei valori da variabili con nomi del tutto diversi. Prima di leggere il corpo, accoppia ogni argomento al parametro nella stessa posizione e chiediti se il tipo e il significato sono compatibili.

Un metodo a numero variabile di argomenti conserva la medesima logica. Se `stampaTitoli(String... titoli)` riceve tre titoli, nel corpo `titoli` è un array di lunghezza tre; se non ne riceve, la lunghezza è zero. La forma con i puntini permette anche una chiamata con un array già esistente. Il vantaggio è una chiamata comoda per una serie omogenea di valori, non l'eliminazione delle regole sui tipi. Per esempio, `stampaTitoli("Java", 7)` non diventa valida perché il metodo accetta un numero variabile di argomenti: `7` non è una `String`.

Nell'esempio storico `DemoElementiMultipli`, il ciclo `for (String argomento : arg)` stampava ogni elemento ricevuto. Letto con la regola appena vista, il metodo non possiede una magia per leggere argomenti illimitati: riceve un array e lo attraversa. Se la chiamata è `stampaTitoli("Java", "JVM")`, il ciclo compie due iterazioni e stampa due righe; con `stampaTitoli()` non entra nel corpo del ciclo. Se occorre un parametro obbligatorio prima della parte variabile, una dichiarazione come `stampaTitoli(String prefisso, String... titoli)` lo esprime. I puntini devono restare sull'ultimo parametro, così il compilatore sa quali argomenti riunire nell'array. Per una lista da conservare o trasformare in più passaggi, può essere più chiaro passare una raccolta esplicita.

Un libro che si sfoglia

Immaginiamo un `Libro` con un numero di pagine e una pagina corrente. I metodi descrivono azioni dell'oggetto, non soltanto formule che restituiscono numeri. Diamo a queste azioni una regola chiara: le pagine sono numerate da **1** a `numeroPagine`; quando siamo all'ultima, `sfogliaAvanti()` non va oltre, e quando siamo alla prima, `sfogliaIndietro()` non scende a zero. Il contratto ci permette di prevedere il comportamento prima di guardare i dettagli del codice.

Il sorgente completo dichiara due campi e tre operazioni pubbliche. `numeroPagine` viene fissato dal costruttore; `paginaCorrente` può cambiare. La parola `final` sul primo campo impedisce una nuova assegnazione dopo la costruzione, mentre l'assenza di `final` sul secondo consente di avanzare e tornare indietro. Il costruttore rifiuta un libro con zero pagine: non avrebbe una pagina 1 da cui iniziare.

```
public final class LibroDaSfogliare {
    private final int numeroPagine;
    private int paginaCorrente;

    public LibroDaSfogliare(int numeroPagine) {
        if (numeroPagine <= 0) {
            throw new IllegalArgumentException("serve almeno una pagina");
        }
        this.numeroPagine = numeroPagine;
        this.paginaCorrente = 1;
    }

    public int paginaCorrente() {
        return paginaCorrente;
    }

    public void sfogliaAvanti() {
        if (paginaCorrente < numeroPagine) {
            paginaCorrente++;
        }
    }

    public void sfogliaIndietro() {
        if (paginaCorrente > 1) {
            paginaCorrente--;
        }
    }

    public static void main(String[] args) {
        LibroDaSfogliare libro = new LibroDaSfogliare(3);
        System.out.println(libro.paginaCorrente());
        libro.sfogliaAvanti();

        System.out.println(libro.paginaCorrente());
        libro.sfogliaIndietro();
        System.out.println(libro.paginaCorrente());
    }
}
```

Le tre righe stampate sono `1`, `2` e `1`. Il primo `println` osserva lo stato iniziale; la seconda stampa segue un avanzamento; la terza segue il ritorno alla prima pagina. Se chiami `sfogliaIndietro()` ancora una volta, il numero resta 1. Se ripeti `sfogliaAvanti()` fino all'ultima pagina e oltre, resta 3. Questo comportamento è una **scelta del nostro modello**, non una proprietà automatica di tutti i libri Java. Avremmo potuto segnalare un tentativo di uscire dai limiti con un'eccezione, purché il contratto lo dicesse. Nel capitolo 8 vedremo come questi vincoli diventano parte dell'incapsulamento.

La classe contiene un metodo `main` soltanto per renderla eseguibile come esercizio. Le operazioni di sfoglio non dipendono da `main` e possono essere usate da un altro programma che crea un `LibroDaSfogliare`. Questa separazione è utile: il punto di ingresso serve alla dimostrazione, mentre campi e metodi descrivono la responsabilità dell'oggetto. Se avessimo scritto tutti i calcoli direttamente in `main`, avremmo stampato le stesse tre righe senza aver imparato a modellare un libro.

Costruire un'istanza

La riga `new Libro("Java Mattone dopo Mattone")` crea un'istanza e invoca il **costruttore** `Libro(String titolo)`. Un costruttore ha il nome della classe e non dichiara un tipo di ritorno, neppure `void`. Qui riceve il titolo e lo assegna al campo. La variabile `primo` conserva il riferimento all'oggetto creato, non una copia di tutti i suoi campi. Il capitolo 8 approfondirà validazione, costruttori alternativi e invarianti; per ora ci basta vedere come la dichiarazione della classe diventi un'istanza utilizzabile.

Nel costruttore i due nomi `titolo` indicano cose diverse. `this.titolo` è il campo dell'oggetto corrente; il `titolo` a destra è il parametro ricevuto. `this` è il riferimento con cui un metodo o un costruttore indica la propria istanza. Quando non c'è ambiguità, come in `prestiti++`, Java consente di ometterlo. Non è un secondo oggetto nascosto: è un modo esplicito di dire «questo libro».

Nomi, ambito e `this`: seguiamo la ricerca

Immagina una classe `Punto` con i campi `x` e `y` e un metodo `sposta(int x, int y)`. Dentro il metodo, i nomi semplici `x` e `y` indicano i parametri, perché sono i nomi più vicini nel loro ambito. Scrivere `x = x;` assegnerebbe il parametro a sé stesso: il campo resterebbe invariato. Le istruzioni `this.x = x;` e `this.y = y;` indicano invece i campi dell'oggetto corrente a sinistra, e i parametri a destra. È la stessa regola già vista nel costruttore di `Libro`, applicata a due dati.

Una variabile dichiarata dentro un blocco interno è visibile dove le regole di ambito lo consentono, fino alla fine di quel blocco. Un parametro è disponibile nel corpo del metodo che lo dichiara. Un campo è accessibile dai metodi della classe, secondo le regole di accesso e di contesto. Questi sono confini del **nome nel sorgente**: non dicono che la memoria di un oggetto venga liberata appena una parentesi graffa si chiude. Se un riferimento all'oggetto è conservato altrove, l'oggetto può continuare a essere raggiungibile.

Possiamo scrivere `this.descrizione()` da un metodo di istanza di `Libro` oppure, se non c'è ambiguità, `descrizione()`. In entrambi i casi chiediamo l'operazione allo stesso oggetto. `this` può anche essere passato come argomento o restituito da un metodo quando c'è un motivo preciso per consegnare il riferimento all'istanza

corrente. Non si può invece usare `this` nel corpo di un metodo `static`, perché quel metodo non è legato a una particolare istanza. La specifica delle classi distingue questi contesti.

Consideriamo un esperimento in cui un campo e una variabile locale si chiamano entrambi `appoggio`. È un caso utile perché obbliga a leggere i nomi con attenzione. Nel programma `DemoAmbito`, il campo parte da 5, la variabile locale da 3 e il parametro `valore` riceve 2. Prima di eseguirlo, prova a seguire le tre assegnazioni e a scrivere i tre numeri che saranno stampati.

```
public class DemoAmbito {
    private int appoggio = 5;

    public void somma(int valore) {
        int appoggio = 3;
        appoggio = appoggio + valore;
        System.out.println("locale: " + appoggio);

        this.appoggio = this.appoggio + valore;
        System.out.println("campo: " + this.appoggio);

        this.appoggio = this.appoggio + appoggio;
        System.out.println("campo finale: " + this.appoggio);
    }

    public static void main(String[] args) {

        new DemoAmbito().somma(2);
    }
}
```

La prima assegnazione usa il nome semplice `appoggio`, che nel corpo di `somma` indica la variabile locale: 3 più 2 dà 5. Il campo è ancora 5. La seconda assegnazione nomina esplicitamente il campo con `this.appoggio`: 5 più 2 dà 7. La terza somma al campo 7 il valore locale 5 e ottiene 12. L'output è quindi `locale: 5`, `campo: 7`, `campo finale: 12`. Le parole davanti ai numeri non sono decorative: ci impediscono di attribuire per errore una stampa al dato sbagliato.

Se togliamo `this.` dalla seconda assegnazione, non otteniamo un modo più corto di aggiornare il campo. Aggiorniamo ancora la variabile locale e cambiamo il risultato dell'esperimento. La presenza di un campo non rende impossibile usare lo stesso nome per una variabile locale, ma l'ombreggiamento aumenta il carico di lettura. A volte è naturale, come `this.titolo = titolo` in un costruttore; in altri casi è più chiaro scegliere nomi diversi. L'esperimento insegna la regola del linguaggio e, insieme, perché vale la pena non sfruttarla senza motivo.

Il campo continua a esistere nello stato dell'oggetto dopo che `somma` è terminato. La variabile locale `appoggio` non è invece accessibile da una successiva chiamata: ciascuna esecuzione del metodo ha le proprie variabili locali. Se invochiamo `somma(2)` una seconda volta **sulla stessa istanza**, la variabile locale riparte da 3 mentre il campo riparte da 12, il risultato della chiamata precedente. Se creiamo un nuovo `DemoAmbito`, il suo campo parte da 5. Questa variante distingue l'ambito di un nome dalla durata dello stato dell'oggetto, due idee che non dobbiamo sovrapporre.

La figura 7.1 rappresenta il punto più importante del programma. Dopo l'assegnazione, `primo` e `secondo` si riferiscono allo stesso oggetto; chiamare un metodo attraverso uno dei due nomi può quindi produrre un risultato visibile attraverso l'altro.

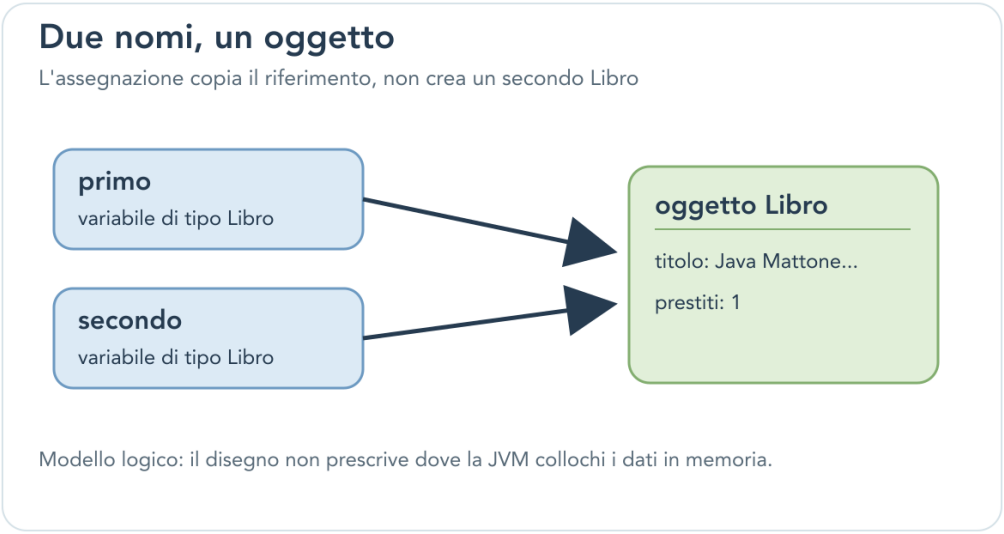


Figura 7.1 – `primo` e `secondo` indicano la stessa istanza di `Libro`. La figura è un modello logico dei riferimenti: non prescrive in quale area fisica della memoria la JVM debba collocare i dati.

`primo == secondo` è `true` perché i due riferimenti indicano **la stessa istanza**. Questa prova riguarda l'identità. Due libri creati con due espressioni `new` diverse possono avere lo stesso titolo ma essere oggetti distinti; nel capitolo 10 useremo `equals` per definire, quando serve, l'uguaglianza del contenuto. Non sostituire automaticamente `==` con `equals` senza chiederti quale domanda vuoi fare.

Una variabile di riferimento può contenere `null`, che significa «nessun oggetto indicato». `Libro assente = null;` è una dichiarazione valida; `assente.descrizione()` fallisce durante l'esecuzione perché non esiste un libro su cui chiamare il metodo. Un **campo** di tipo riferimento non inizializzato esplicitamente riceve il valore iniziale `null`; una **variabile locale** deve invece essere assegnata prima di essere letta. Nel disegno di stack e heap, distinguiamo prima le due regole di inizializzazione: la posizione disegnata per il dato non deve nasconderle. Nel capitolo 12 vedremo come leggere l'eccezione prodotta da un accesso a `null`.

Un array può conservare più riferimenti: `Libro[] scaffale = {primo, secondo};` contiene due elementi, ma nell'esempio entrambi indicano lo stesso libro. La lunghezza dell'array è due; il numero di oggetti `Libro` distinti indicati dai suoi elementi è uno. Per l'uso degli indici e la forma dell'array puoi tornare al capitolo 4.

Quattro dichiarazioni, quattro domande

Confrontiamo `int copie = 2;`, `Libro libro;`, `Libro[] scaffale;` e `Libro[] ripiani = new Libro[3];`. La prima variabile contiene direttamente un valore primitivo. La seconda è una variabile di tipo riferimento, ma la sola dichiarazione locale non crea un `Libro` e non permette ancora di leggere `libro`: prima occorre assegnarle un valore. La terza dichiara un riferimento ad array senza creare l'array. Solo la quarta costruisce un array di tre posti; ciascun elemento di tipo `Libro` inizia con `null`, perché gli elementi dell'array ricevono i valori iniziali del loro tipo. Nessuno di quei tre posti contiene già un libro. Per ottenere tre libri servono tre creazioni, o assegnazioni di riferimenti a libri creati altrove.

La figura 7.2 separa il valore primitivo dal riferimento che permette di raggiungere un oggetto. Le due dichiarazioni possono apparire simili nel sorgente, ma rispondono a domande diverse quando leggiamo il valore di una variabile.



Figura 7.2 – `int` contiene un valore numerico; `Libro` contiene un riferimento. Il modello riguarda i valori osservabili e non stabilisce una collocazione fisica obbligata nella JVM.

Questa lettura corregge un equivoco favorito da alcuni vecchi disegni della memoria. Il tipo `int` ha una dimensione definita dal linguaggio, ma da ciò non segue che la variabile locale `copie` debba trovarsi fisicamente in un'area chiamata heap, né che una variabile di riferimento sia un puntatore manipolabile come in C. La specifica descrive valori e comportamento osservabile; la JVM sceglie come rappresentarli. Per ragionare sul programma ci bastano tre domande: quale valore ha la variabile, quale oggetto può raggiungere attraverso quel valore e quali operazioni il suo tipo permette di scrivere. Una figura che disegna frecce fra nomi e oggetti risponde alla seconda domanda; non è una fotografia della memoria.

La sintassi `new Libro("Java").descrizione()` usa un oggetto senza assegnare il riferimento a una variabile con un nome. L'oggetto esiste e il metodo può essere chiamato; semplicemente, dopo l'espressione non conserviamo in quel punto un riferimento con cui richiamarlo. Al contrario, `new Libro("Java")` da solo come istruzione crea un oggetto che il nostro codice non usa: può avere senso se il

costruttore produce un effetto documentato, ma per il modello `Libro` sarebbe una scelta poco chiara. Non deduciamo da questa osservazione **quando** la JVM recupererà la memoria: la gestione automatica della memoria non promette un istante di raccolta visibile al programma.

L'esempio storico usava anche un oggetto `Integer` appena costruito per trasformare un numero in testo. L'idea di chiamare subito un metodo sull'espressione resta valida, ma oggi, per ottenere il testo di `10`, è più diretto scrivere `Integer.toString(10)` o `String.valueOf(10)`: non dobbiamo creare esplicitamente un wrapper. In questo caso il metodo appartiene alla classe e riceve il valore numerico come argomento. Quando torneremo sui wrapper nel capitolo 10, vedremo perché i loro vecchi costruttori non sono la via ordinaria e perché un `Integer` è comunque un oggetto diverso da un `int`.

> **Nota bene – Valori iniziali e assegnazione certa.** Un campo `int prestiti` inizia da `0` e un campo `Libro collegato` da `null` se non hanno un inizializzatore esplicito. Una variabile locale, invece, deve avere un valore assegnato su ogni percorso che porta alla sua lettura. La distinzione rende visibile una dimenticanza durante la compilazione; non è un invito a lasciare i campi senza una scelta consapevole del loro valore iniziale.

L'array stesso è un oggetto, distinto dagli eventuali oggetti contenuti nei suoi elementi. Se scriviamo `Libro[] primoScaffale = new Libro[2]; Libro[] secondoScaffale = primoScaffale;`, abbiamo **un** array raggiungibile con due variabili. Assegnare `primoScaffale[0] = new Libro("A")` rende quel libro visibile anche attraverso `secondoScaffale[0]`. Se invece scriviamo `secondoScaffale = new Libro[2]`, la seconda variabile comincia a indicare un altro array; il primo continua a contenere il libro. Questa è la stessa regola dei riferimenti applicata a un contenitore. Si possono quindi avere due livelli di condivisione: due variabili che indicano lo stesso array, oppure due array distinti che contengono riferimenti allo stesso `Libro`.

Un array creato con `new Libro[2]` non chiama due volte il costruttore di `Libro`. I suoi elementi partono da `null` e ogni libro va costruito esplicitamente, salvo che si assegnino riferimenti a libri già esistenti. Al contrario, `new int[2]` contiene subito due valori primitivi inizializzati a zero. La differenza non rende gli array di oggetti «meno veri» degli array primitivi: chiarisce che il tipo degli elementi determina **che cosa** si trova in ciascuna posizione. Un disegno dei riferimenti agli array deve mostrare separatamente la variabile, l'oggetto array e gli oggetti a cui gli elementi possono rimandare.

> **Per verificare – Conta oggetti, non nomi.** Dopo `Libro a = new Libro("A"); Libro b = a; Libro[] scaffale = {a, b};`, quante variabili di riferimento abbiamo dichiarato? Quanti oggetti `Libro` sono stati costruiti? E quanti elementi contiene l'array? Le risposte sono tre variabili (`a`, `b`, `scaffale`), un libro e due elementi nell'array. Inoltre è stato creato l'oggetto array. Se cambi solo `b = new Libro("B")`, il secondo elemento di `scaffale` continua a indicare il primo libro: l'assegnazione a `b` non riscrive automaticamente il contenuto dell'array.

La figura 7.3 mostra l'array come un oggetto con due posti. Entrambi possono contenere riferimenti allo stesso libro, e il numero di posti non ci dice quanti libri distinti siano stati creati.

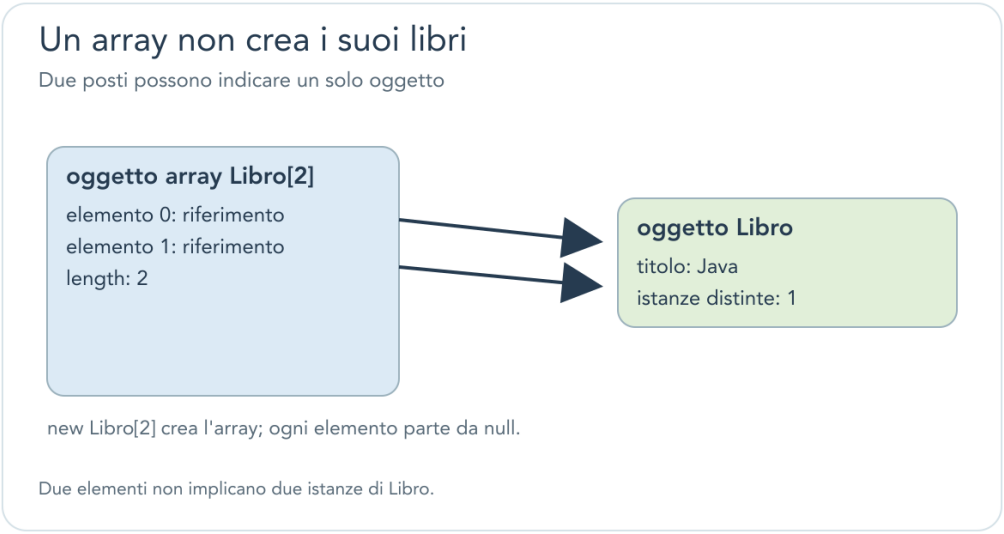


Figura 7.3 – Due elementi dell'array possono indicare un solo oggetto `Libro`*. La costruzione dell'array, da sola, lascia i due elementi a* `null`*; le frecce rappresentano una situazione successiva alle assegnazioni.*

Una **pila** ci mostra che più operazioni possono appartenere allo stesso oggetto. Sviluppiamo l'esempio nel capitolo 8: `inserisci` aggiunge un numero e `estrai` restituisce l'ultimo inserito. Si dice **LIFO**, dall'inglese *last in, first out*. Prima di leggere come è costruita, puoi già descriverne la responsabilità: deve sapere quanti elementi contiene e non deve estrarre da una pila vuota. La stessa pila ci aiuterà a leggere una catena di chiamate nel capitolo 12.

> **Nota bene – Non confondere variabile e oggetto.** Assegnare `secondo = primo` copia il valore del riferimento. Non duplica l'istanza e non invoca il costruttore. Se in seguito scrivi `secondo = null`, l'oggetto può ancora essere raggiunto tramite `primo`: l'assegnazione cambia un nome, non distrugge immediatamente l'oggetto.

Un riferimento passato a un metodo

La distinzione diventa ancora più utile quando passiamo un `Libro` a un altro metodo. Java passa i valori degli argomenti: per un parametro di tipo riferimento, il valore copiato è il riferimento all'oggetto. Se il metodo chiama `libro.registraPrestito()`, modifica lo stato della stessa istanza visibile al chiamante. Se invece assegna `libro = new Libro("Altro titolo")`, cambia soltanto il proprio parametro locale; la variabile del chiamante continua a indicare il libro iniziale. Dire semplicemente «l'oggetto è passato per riferimento» nasconde questa differenza e porta a previsioni sbagliate.

Con un `int` la prova è più intuitiva: il metodo riceve un valore numerico; assegnare un altro numero al parametro non cambia la variabile originale. Nel caso di `Libro`,

il metodo riceve sempre un valore, ma quel valore consente di raggiungere e modificare un oggetto condiviso. Due frasi sono entrambe vere: la riassegnazione del parametro resta locale; la mutazione dell'oggetto può essere osservata fuori dal metodo.

Una chiamata osservata da entrambi i lati

Consideriamo un metodo `presta(Libro ricevuto)` che chiama `ricevuto.registraPrestito()` e poi assegna a `ricevuto` un nuovo `Libro`. Dal lato del chiamante, la variabile `primo` continua a indicare il libro iniziale. Il suo contatore è aumentato perché il metodo ha raggiunto **lo stesso oggetto** tramite una copia del riferimento. La nuova assegnazione, invece, cambia soltanto la variabile locale `ricevuto`; non torna indietro lungo la chiamata per sostituire il valore di `primo`. In termini di comportamento osservabile, il metodo può mutare l'oggetto raggiunto, ma non può riassegnare la variabile locale che il chiamante ha usato come argomento.

Per controllare la previsione senza una figura di memoria, annota due colonne. Prima della chiamata, `primo` e `ricevuto` indicano lo stesso libro solo dopo il passaggio del parametro; il contatore vale zero. Dopo `registraPrestito()`, il contatore di quell'unico libro vale uno. Dopo `ricevuto = new Libro("Altro")`, `ricevuto` indica un secondo libro, mentre `primo` indica ancora il primo, con contatore uno. Se il metodo restituisse `ricevuto` e il chiamante scrivesse `primo = presta(primo)`, sarebbe una **nuova assegnazione esplicita nel chiamante** a cambiare `primo`. Il passaggio degli argomenti, da solo, non l'ha fatto.

> **Per capire – La parola «riferimento».** Un riferimento è un valore che permette di raggiungere un oggetto; non è l'oggetto stesso e Java non offre su di esso l'aritmetica dei puntatori del C. La copia del valore di riferimento spiega perché due variabili possono osservare la stessa mutazione. La distinzione basta per risolvere molti esercizi sulle chiamate, senza attribuire alla specifica una disposizione fisica fissa di stack e heap.

Getter e setter hanno un contratto

Consideriamo una classe `Macchina` con campi come colore e targa e metodi `getTarga()` e `setTarga(...)`. Un *getter* restituisce un'informazione, un *setter* accetta una richiesta di modifica. Avere entrambi per ogni campo, però, non è una regola obbligatoria. Se la targa identifica l'auto e non deve cambiare dopo la creazione, un setter pubblico romperebbe proprio l'invariante che volevamo proteggere. Se il colore può cambiare, un metodo `vernicia(String nuovoColore)` può raccontare l'azione meglio di un generico `setColore`.

Un metodo che modifica lo stato deve anche decidere che cosa fare davanti a un argomento non valido. Per una targa vuota possiamo rifiutare la chiamata, oppure usare una rappresentazione che non possa esprimere quel valore. Non basta spostare l'assegnazione da `macchina.targa = ...` a `macchina.setTarga(...)` se il setter accetta qualunque testo senza controllo. Nel capitolo 8 vedremo come definire le regole dell'oggetto e mantenerle sia nei costruttori sia nei metodi.

Stato, identità e uguaglianza

Lo **stato osservabile** del nostro `Libro` comprende almeno il titolo e il numero di prestiti registrati. Il comportamento dei metodi può dipendere da questi valori: `descrizione()` cambia dopo `registraPrestito()`. Non ogni campo che una classe potrebbe contenere deve entrare nella sua nozione di uguaglianza. Un contatore di accessi usato soltanto per diagnostica, per esempio, può cambiare senza cambiare il significato del libro per il catalogo. La scelta dipende dal contratto della classe, non da una formula universale «due oggetti sono uguali se tutti i campi coincidono».

L'operatore `==` su due riferimenti risponde alla domanda di identità: indicano la stessa istanza? Il metodo `equals` risponde invece secondo il contratto definito dalla classe. L'implementazione ereditata da `Object` usa l'identità; alcune classi, come `String`, ridefiniscono `equals` per confrontare il contenuto. Per questo `new String("Java").equals(new String("Java"))` è vero, mentre il confronto con `==` fra i due oggetti creati con `new` è falso. Non usiamo due letterali identici per dimostrarlo: l'internamento delle stringhe renderebbe l'esempio ingannevole. Quando definiremo l'uguaglianza dei nostri tipi, dovremo anche rispettare il contratto di `hashCode`.

> **Concetto chiave – Il significato prima del confronto.** Un prestito può essere «lo stesso prestito» perché ha un identificatore, anche se lo stato è cambiato dopo la restituzione. Due edizioni con titolo e anno uguali possono invece essere considerate uguali come valori. Prima di scrivere `equals`, scegli la domanda del dominio a cui deve rispondere.

Membri della classe e punto di ingresso

Tutti gli oggetti `Libro` hanno un titolo proprio. Se però ci serve una costante identica per tutta la classe, possiamo dichiararla con `static final`. Un esempio è `static final int MASSIMO_PRESTITI = 20;`: il nome e il valore appartengono alla classe e si richiamano come `Libro.MASSIMO_PRESTITI`, se la visibilità lo consente. Un campo `static` modificabile, invece, contiene uno stato condiviso fra tutte le istanze: usarlo per il numero di prestiti di un singolo libro sarebbe un errore, perché ogni libro aggiornerebbe lo stesso contatore.

Il nome maiuscolo con parole separate da sottolineature segue la convenzione usata per le costanti. Aiuta il lettore a riconoscere un valore condiviso e non riassegnabile, ma non sostituisce i modificatori: `MASSIMO_PRESTITI` scritto in maiuscolo senza `final` potrebbe ancora essere riassegnato. Analogamente `static final` non rende immutabile in profondità un oggetto mutabile raggiunto dal campo. Per un limite numerico come `20`, invece, il valore stesso non ha parti interne modificabili. Distinguere convenzione del nome, regola di assegnazione e mutabilità del valore evita tre promesse diverse nascoste nella stessa parola «costante».

Un metodo `static` può usare direttamente altri membri statici, ma non un campo di istanza senza conoscere **quale** istanza. Può comunque ricevere un `Libro` come parametro e chiamarne i metodi: il divieto non riguarda gli oggetti in generale,

riguarda l'assenza di un `this` implicito. Questa distinzione spiega sia perché `main` è tradizionalmente statico sia perché `main` può creare e usare oggetti.

Il punto di ingresso della forma tradizionale è `public static void main(String[] args)`. Il comando `java` avvia il programma a partire dalla classe che indichiamo; se più classi dichiarano un metodo `main` adatto, scegliamo quale avviare nominando quella classe. L'array `args` contiene le parole passate dopo il nome della classe, come abbiamo verificato nel capitolo 3. `main` non rende l'intera applicazione un oggetto: è un metodo di avvio. Nel seguito incontreremo anche la forma compatta introdotta in Java 25, ma la forma tradizionale rimane importante per leggere progetti esistenti.

Leggere `main` come un normale metodo, con un ruolo speciale

Se avvii `java -cp build Libro Ada 3`, le parole dopo `Libro` diventano elementi dell'array `args`: `args[0]` contiene `"Ada"` e `args[1]` contiene `"3"`. Il secondo elemento è ancora testo; per ottenere un numero, un programma dovrebbe convertirlo e gestire un eventuale input non valido. Se avvii il programma senza argomenti, l'array è vuoto e leggere `args[0]` provocherebbe un errore. Prima di usarlo, controlliamo quindi `args.length`. Questo piccolo passaggio collega la firma del punto di ingresso alle regole sugli array già studiate e mostra che un parametro non crea dati che il chiamante non ha fornito.

`static` non significa che il metodo lavori «prima di ogni oggetto» o che non possa creare istanze. Significa che il metodo è associato alla classe e può essere chiamato senza un ricevente `Libro`. Nel corpo di `main` possiamo scrivere `new Libro(...)`, conservare il riferimento e invocare `registraPrestito()`. Possiamo anche chiamare un altro metodo statico della classe direttamente. Se tentassimo di leggere `titolo` senza indicare un libro, il compilatore ci fermerebbe: di quale delle possibili istanze dovrebbe prendere il titolo? In questo modo il messaggio del compilatore rinforza il modello classe/istanza, anziché sembrare una regola arbitraria sulle parole chiave.

Il nome `args` è convenzionale, ma non è una parola riservata. Possiamo chiamare il parametro `argomenti`; conta il tipo e la forma del punto di ingresso. Nella dichiarazione più nota, `String[]` indica un array di stringhe e `void` indica che il metodo non consegna un valore al proprio chiamante. Stampare una riga e restituire un valore sono ancora operazioni diverse: `main` può produrre messaggi con `System.out`, ma la sua firma non dichiara un risultato del programma. Un eventuale codice di uscita del processo segue un altro meccanismo e non va confuso con `return` di un metodo che restituisce `int`.

`System`: uscita, errori e proprietà

La riga `System.out.println(...)` contiene due passaggi. `System.out` è un campo statico della classe `System` che fornisce un flusso di uscita; `println` è un metodo dell'oggetto flusso. Non è quindi corretto chiamare `println` un metodo statico di `System`. `System.err` è un altro flusso, convenzionalmente usato per i messaggi di errore. Anche se entrambi possono apparire nello stesso terminale, strumenti e script possono indirizzarli in luoghi diversi. Useremo `System.exit(int)` soltanto quando il programma deve terminare il processo con un codice di uscita; chiamarlo da un normale metodo di libreria sorprenderebbe chi usa quella libreria.

Il metodo statico `System.getProperty("java.version")` restituisce una proprietà del runtime, oppure `null` se la chiave richiesta non è definita. Altre chiavi utili sono `os.name` per il nome del sistema, `user.dir` per la directory di lavoro e `line.separator` per il separatore di riga. I valori cambiano da una macchina all'altra: un elenco di percorsi Windows 2000 descriverebbe soltanto la macchina e l'epoca in cui è stato prodotto, non un output da aspettarsi oggi. Prova a stampare soltanto `java.version` e `user.dir`, poi confrontali con `java --version` e con la directory da cui hai avviato il programma.

> **Nota bene – Proprietà non significa variabile d'ambiente.** Le proprietà di sistema sono coppie chiave e valore accessibili dall'API Java. Le variabili d'ambiente sono fornite al processo dal sistema operativo e si leggono con `System.getenv`. Alcune informazioni possono assomigliarsi, ma i due meccanismi non sono identici. La documentazione di `System` elenca metodi e proprietà standard.

Leggere una proprietà senza trasformarla in una promessa

Le proprietà di sistema descrivono l'ambiente in cui il programma viene eseguito; un elenco di valori raccolti su una sola macchina diventa presto una fotografia datata. Se scriviamo `String versione = System.getProperty("java.version");`, la variabile riceve un testo che descrive la versione del runtime con cui il programma sta effettivamente girando. Compilare con `--release 25` e avviare con un altro runtime sono operazioni distinte: la proprietà riguarda l'esecuzione. `System.getProperty("user.dir")` descrive la directory di lavoro del processo, che può essere diversa dalla cartella in cui si trova il sorgente o il file `.class`. È facile verificarlo avviando lo stesso programma da due directory diverse con un classpath adatto.

Un secondo esperimento mette in relazione `System.out` e `System.err`. Stampa `"risultato"` sul primo e `"problema"` sul secondo. In un terminale puoi vedere entrambe le righe, ma non concludere che siano lo stesso flusso: l'ambiente può separarle, come fanno molti strumenti di esecuzione e verifica. La scelta del flusso fa parte del significato del messaggio. L'uscita ordinaria è adatta a un risultato che un altro programma potrebbe leggere; l'uscita degli errori a una diagnosi che non deve confondersi con quel risultato. Nel nostro `Libro` stampiamo una descrizione su `out` perché fa parte dell'esito normale dell'esempio.

> **Approfondimento – Chiavi e valori.** Una proprietà si cerca tramite una chiave testuale. `System.getProperty("java.version")` chiede la versione dell'ambiente Java in esecuzione; se una chiave non è presente, `getProperty` restituisce `null`. Prima di chiamare un metodo sulla stringa ottenuta, decidiamo quindi come gestire l'assenza. Possiamo anche usare la forma con un valore predefinito, come `System.getProperty("applicazione.porta", "8080")`. Il risultato resta una stringa: per usarlo come porta numerica servono conversione e controllo dell'intervallo ammesso. La prova consiste nel leggere una proprietà nota e una chiave assente, verificando il valore e il caso da gestire. Interrogare l'ambiente in questo modo evita di prendere per universali valori che dipendono dall'installazione.

Una forma per dati semplici: i record

Il nostro `Libro` possiede un comportamento che cambia stato. A volte, però, serve soltanto raggruppare dati che vogliamo descrivere in modo trasparente, per esempio il titolo e l'anno di una particolare edizione. Per questo Java offre i **record**, classi speciali che dichiarano i propri componenti nell'intestazione. Un record come `record Edizione(String titolo, int anno) { }` mette a disposizione un costruttore, i metodi `titolo()` e `anno()` e metodi per confronto e rappresentazione testuale. Non dobbiamo scriverli tutti a mano per un semplice aggregato.

```
record Edizione(String titolo, int anno) { }

public class DemoRecord {
    public static void main(String[] args) {
        Edizione prima = new Edizione("Java Mattone dopo Mattone", 2026);
        System.out.println(prima.titolo() + " (" + prima.anno() + ")");
    }
}
```

Il file si chiama `DemoRecord.java` e stampa `Java Mattone dopo Mattone (2026)`. La documentazione dei record descrive i membri generati. I campi dei componenti sono `final`, ma questo non rende automaticamente profondamente immutabile qualsiasi oggetto a cui un componente si riferisca. Torneremo su copie difensive e validazione nel capitolo 8. I record sono stabili da Java 16; non richiedono preview in Java 25.

Decidere quando il record esprime il modello

Per `Edizione`, titolo e anno descrivono un valore che ci interessa leggere e confrontare. Due istanze costruite con gli stessi componenti risultano uguali secondo i metodi generati dal record; due riferimenti a istanze distinte restano diversi secondo ==. Questa è la stessa distinzione che approfondiremo nel capitolo 10, ma qui mostra perché il record è comodo quando il significato del valore corrisponde ai suoi componenti. Se l'identità del libro dipendesse invece da un codice di catalogo assegnato dall'applicazione, o se dovessimo registrare e modificare continuamente prestiti, una classe ordinaria con un contratto esplicito potrebbe raccontare meglio il problema.

La brevità della dichiarazione non elimina la responsabilità di controllare i dati. `new Edizione(null, -3)` è sintatticamente costruibile con il record mostrato: il compilatore non sa che un titolo mancante e un anno negativo non hanno senso per il nostro dominio. Possiamo aggiungere un costruttore che convalidi i componenti. Nel capitolo successivo vedremo come la stessa domanda sugli invarianti si applica a ogni forma di classe. Il criterio per scegliere un record, quindi, non è «questa definizione occupa una riga»; è «i componenti esposti descrivono esattamente il valore che voglio modellare».

Un record può contenere un componente di tipo array, per esempio `record Scaffale(Libro[] libri) {}`. Il campo del componente non può essere riassegnato dall'esterno, ma l'array passato al costruttore e quello restituito dall'accessore possono comunque esporre i propri elementi a modifiche. Inoltre il confronto di due array non diventa automaticamente un confronto elemento per elemento perché si trovano dentro un record. È una ragione concreta per non chiamare

«immutabile» ogni record senza esaminare ciò che custodisce. La domanda su riferimenti e oggetti è la stessa che abbiamo costruito fin dall'inizio del capitolo.

Un programma ancora più piccolo in Java 25

Forse ti sei chiesto perché il primo programma del libro richiedesse classe, `public`, `static` e array di argomenti solo per stampare una riga. Java 25 offre anche un **file sorgente compatto**: si può scrivere un metodo `main` senza una dichiarazione esplicita di classe. Il file `CiaoCompatto.java` dei materiali contiene:

```
void main() {
    IO.println("Ciao da Java 25!");
}
```

Il comando `java CiaoCompatto.java` stampa `Ciao da Java 25!`. In questo caso il linguaggio dichiara implicitamente una classe per il file e il comando avvia un metodo `main` di istanza. `IO.println` è una scorciatoia per l'uscita testuale disponibile in Java 25; il nostro `System.out.println` resta valido. Questa forma è utile per un esperimento breve, ma non elimina le classi dal linguaggio e non è la forma scelta per il modello `Libro`, dove nome, stato e metodi hanno un ruolo didattico preciso. La guida ufficiale sui file compatti ne descrive il comportamento.

> **Versione Java – Due forme stabili.** I record sono permanenti da Java 16; i file sorgente compatti e i metodi `main` di istanza sono permanenti da Java 25. Un lettore con un JDK precedente deve usare la forma supportata dalla propria release. Il riepilogo ufficiale delle modifiche del linguaggio registra introduzione e stato delle funzionalità.

Per verificare

Prima di eseguire il programma `Libro`, prevedi che cosa cambierebbe se la riga `Libro secondo = primo;` diventasse `Libro secondo = new Libro("Java Mattone dopo Mattone");`. Le due variabili indicherebbero ancora la stessa istanza? Quale descrizione stamperebbe `secondo` dopo `primo.registraPrestito()`? Poi prova la modifica e spiega il risultato senza usare la parola «copia» in modo ambiguo. L'attività del capitolo 7 riprende questa prova con un caso da risolvere in autonomia.

8. Incapsulamento

Una regola che deve restare vera

Nel capitolo precedente abbiamo creato un `Libro` con un titolo e un contatore. Il campo `prestiti` è privato e si modifica tramite `registraPrestito()`. Questo evita che un altro pezzo di programma scriva direttamente un numero negativo nel campo, ma non dimostra ancora che l'oggetto sia progettato bene: se il metodo stesso sottraesse uno senza controllo, la regola sarebbe violata lo stesso. **Incapsulare** significa disegnare un confine attorno allo stato e alle operazioni che lo governano, così che dall'esterno si possano compiere soltanto azioni coerenti con il significato dell'oggetto.

Una regola che deve essere vera per tutta la vita utile di un oggetto si chiama **invariante**. In un registro di voti, per esempio, vogliamo almeno un voto, e ogni voto deve stare nell'intervallo da zero a trenta. Se la media è una operazione del registro, non può avere senso chiedergli di dividerla per zero perché qualcuno ha svuotato di nascosto l'array. Il problema concreto, dunque, non è «come nascondo i campi?» ma «come impedisco che si crei o diventi osservabile uno stato impossibile?».

Possiamo spiegare l'occultamento con la leva del cambio di un'automobile. L'analogia è utile finché ricorda che il guidatore usa comandi e non modifica direttamente gli ingranaggi. Una leva, però, non garantisce da sola che il cambio non possa guastarsi: anche l'implementazione deve rispettare le regole. Preferisco quindi mostrare quelle regole nel codice, dove possiamo verificarle.

Pensiamo alla differenza fra chiedere all'auto di **inserire una marcia** e scrivere direttamente il numero della marcia in una variabile. Nel primo caso un'operazione può controllare se il cambio ammette quella richiesta nella condizione corrente; nel secondo chi usa l'auto deve conoscere e rispettare regole che appartengono all'auto stessa. L'analogia ha un limite: il software non impedisce che un programmatore scriva un'implementazione sbagliata del metodo pubblico. Per questo, dopo aver scelto un confine, lo proviamo con chiamate lecite e illecite. La qualità dell'incapsulamento si misura nelle **azioni consentite** e nelle garanzie osservabili, non nel numero di parole `private` presenti nel file.

> **Approfondimento – Rappresentazione e contratto.** La rappresentazione è il modo scelto dalla classe per conservare i dati: `RegistroVoti` usa un array, `Pila` un array e un contatore. Il contratto riguarda invece ciò che chi usa la classe può aspettarsi: quali valori sono ammessi, quali risultati ottiene e che cosa accade nei casi anomali. Se un giorno sostituiamo l'array con un'altra struttura, il codice dei chiamanti dovrebbe continuare a funzionare finché manteniamo il contratto. Questo è il vantaggio pratico del confine.

Un registro che custodisce i propri dati

Il programma seguente è disponibile anche come `RegistroVoti.java`. Salvalo con questo nome, compilalo con `javac --release 25 -Xlint:all -d build RegistroVoti.java` e avvialo con `java -cp build RegistroVoti` dalla cartella del file.

```
public final class RegistroVoti {
    private final String studente;
    private final int[] voti;

    public RegistroVoti(String studente, int[] voti) {
        if (studente == null || studente.isBlank()) {
            throw new IllegalArgumentException("studente mancante");
        }
        if (voti == null || voti.length == 0) {
            throw new IllegalArgumentException("serve almeno un voto");
        }
        for (int voto : voti) {
            if (voto < 0 || voto > 30) {
                throw new IllegalArgumentException("voto fuori intervallo");
            }
        }

        this.studente = studente;
        this.voti = voti.clone();
    }

    public RegistroVoti(String studente, int voto) {
        if (voto < 0 || voto > 30) {
            throw new IllegalArgumentException("voto fuori intervallo");
        }
        this(studente, new int[] {voto});
    }

    public String studente() {
        return studente;
    }

    public int[] voti() {

        return voti.clone();
    }

    public double media() {
        int somma = 0;
        for (int voto : voti) {
            somma += voto;
        }
        return (double) somma / voti.length;
    }

    public static void main(String[] args) {
        int[] originali = {24, 27, 30};
        RegistroVoti registro = new RegistroVoti("Ada", originali);
        originali[0] = 0;
        int[] ricevuti = registro.voti();
```

```
        ricevuti[1] = 0;
        System.out.println(registro.studente() + ": " + registro.media());
    }
}
```

L'output è `Ada: 27.0`. Nel `main` modifichiamo due array fuori dal registro: prima quello consegnato al costruttore, poi quello ottenuto da `voti()`. Se il registro avesse conservato o restituito direttamente il proprio array, la media potrebbe cambiare senza passare da alcuna operazione che controlli i voti. Le due chiamate a `clone()` creano invece copie dell'array; per un array di `int`, copiare gli elementi significa copiare i valori primitivi. La figura 8.1 mostra le due direzioni del confine.



Figura 8.1 – Il registro copia l'array all'ingresso e all'uscita. Chi lo usa può modificare le proprie copie senza cambiare i voti custoditi dall'oggetto. Il disegno descrive il contratto, non la disposizione fisica della memoria.

> **Nota bene – `final` non congela un array.** `private final int[] voti` impedisce al codice della classe di riassegnare il campo a un altro array dopo l'inizializzazione. Non impedisce di cambiare gli elementi dell'array. L'invariante dipende quindi dalla validazione e dalle copie, oltre che dalla visibilità del campo. Se gli elementi fossero oggetti modificabili, `clone()` copierebbe i riferimenti e potrebbe non essere sufficiente: servirebbe valutare copie degli elementi o un diverso modello.

Verificare le due porte del registro

Seguiamo la costruzione una riga per volta. `originali` contiene tre valori validi. Il costruttore verifica prima il nome, poi l'esistenza di almeno un voto, poi l'intervallo di ciascuno. Solo dopo assegna i campi e copia l'array. Se trovasse `31` durante la verifica, interromperebbe la costruzione: il chiamante non riceverebbe un `RegistroVoti` apparentemente utilizzabile ma con un dato illegittimo nascosto al suo interno. Questa distinzione è importante perché un controllo eseguito soltanto da `media()` arriverebbe troppo tardi e lascerebbe altri metodi liberi di osservare uno stato errato.

La prima porta è l'ingresso. Se sostituissimo `this.voti = voti.clone()` con `this.voti = voti`, l'array posseduto dal chiamante e quello conservato dal registro sarebbero **lo stesso oggetto**. La riga `originali[0] = 0` cambierebbe la media da `27.0` a `19.0`, benché nessun metodo del registro abbia approvato la modifica. La seconda porta è l'uscita. Anche lasciando la copia in ingresso, se `voti()` restituisse semplicemente `voti`, la riga `ricevuti[1] = 0` cambierebbe la media da `27.0` a `18.0`. Servono entrambe le copie perché i due percorsi di accesso sono indipendenti. I numeri rendono visibile la violazione: non stiamo facendo una copia «per prudenza» senza sapere quale problema risolva.

Un array di `int` è un buon primo esempio perché i suoi elementi sono valori primitivi. Se il registro conservasse un array di oggetti `Valutazione` modificabili, copiare soltanto l'array creerebbe un nuovo contenitore ma lascerebbe condivise le istanze al suo interno. Un chiamante potrebbe ancora modificare un voto attraverso un riferimento che possiede. La soluzione dipende dal modello: creare copie degli elementi, scegliere oggetti immutabili oppure esporre soltanto viste e operazioni che mantengano il contratto. **Copia difensiva** significa proteggere davvero il confine, non applicare `clone()` automaticamente a qualsiasi struttura.

L'anagrafica clienti

Consideriamo una classe `Cliente` per un negozio, con nome, cognome e telefono privati e getter e setter per ogni campo. Questa forma rende visibile che un campo privato non può essere modificato direttamente dal codice esterno, ma non basta a proteggere uno stato valido. Un cliente appena creato potrebbe esistere senza nome e cognome; un setter potrebbe accettare un testo vuoto. Anche aggiungere un controllo su `null` richiede attenzione: se il setter del cognome controllasse per errore **il nome**, la validazione riguarderebbe il dato sbagliato. Bisogna dare all’oggetto una regola coerente fin dalla costruzione e verificarla in ogni operazione che la potrebbe cambiare.

Riprendiamo lo stesso dominio nel sorgente `Cliente.java`. Un cliente nasce con nome, cognome e telefono; il nome completo è leggibile, mentre il telefono può essere aggiornato attraverso un'operazione che convalida il nuovo dato. In questa piccola versione nome e cognome non cambiano dopo la costruzione. È una scelta del modello, non un limite dei getter e setter. Se nel negozio esistesse una procedura per correggere il cognome, potremmo aggiungere un metodo con le regole di quella procedura.

```
public final class Cliente {
    private final String nome;
    private final String cognome;
    private String numeroDiTelefono;

    public Cliente(String nome, String cognome, String numeroDiTelefono) {
        this.nome = testoObbligatorio(nome, "nome");
        this.cognome = testoObbligatorio(cognome, "cognome");
        aggiornaTelefono(numeroDiTelefono);
    }

    private static String testoObbligatorio(String testo, String campo) {
        if (testo == null || testo.isBlank()) {
            throw new IllegalArgumentException(campo + " mancante");
        }
        return testo.strip();


    }

    public String nomeCompleto() {
        return nome + " " + cognome;
    }

    public String numeroDiTelefono() {
        return numeroDiTelefono;
    }

    public void aggiornaTelefono(String nuovoNumero) {
        this.numeroDiTelefono = testoObbligatorio(nuovoNumero, "telefono");
    }

    public static void main(String[] args) {
        Cliente cliente = new Cliente(" Ada ", " Rossi ", " 123 ");


        System.out.println(cliente.nomeCompleto());
        cliente.aggiornaTelefono(" 456 ");
        System.out.println(cliente.numeroDiTelefono());
    }
}
```

Il programma stampa `Ada Rossi` e `456`. Il metodo privato `testoObbligatorio` concentra il controllo comune: rifiuta `null` e testo formato soltanto da spazi, poi elimina gli spazi ai margini. Il metodo pubblico `aggiornaTelefono` esprime una modifica ammessa dal dominio; non offre al chiamante un accesso senza regole al campo. La classe è `final`, quindi il suo costruttore può richiamare quel metodo senza il rischio che una sottoclasse ne esegua una ridefinizione su un oggetto ancora incompleto. Quando studieremo l'ereditarietà vedremo perché chiamare un metodo ridefinibile da un costruttore è spesso pericoloso.

Un numero di telefono reale richiede regole più attente di «testo non vuoto»: prefissi, spazi, estensioni e convenzioni internazionali dipendono dal contesto. Qui il controllo è volutamente minimo e dichiarato. Inventare una regola che accetta soltanto cifre farebbe rifiutare numeri validi in molti casi. La lezione sull'incapsulamento non è sapere validare ogni numero del mondo, ma mettere la validazione nel punto in cui l'oggetto protegge il proprio significato. Se cambiamo

la regola, chi chiama `aggiornaTelefono` continua a usare la stessa operazione, mentre il suo contratto documentato ci dice quali nuovi input sono ammessi.

Ritrovare la pila, con regole verificabili

La pila incontrata nel capitolo 7 conserva elementi in ordine **LIFO**: l'ultimo numero inserito è il primo estratto. La classe completa `Pila` usa un array privato e un contatore `dimensione`. L'invariante è `0 ≤ dimensione ≤ elementi.length`: il contatore non può essere negativo né superare la capacità. Il costruttore rifiuta una capacità non positiva; `inserisci` controlla che ci sia spazio; `estrai` controlla che esista un elemento. Quando una di queste operazioni non può essere compiuta, segnala il motivo con un'eccezione, che leggeremo nel capitolo 12.

```
public void inserisci(int valore) {
    if (dimensione == elementi.length) {
        throw new IllegalStateException("pila piena");
    }
    elementi[dimensione] = valore;
    dimensione++;
}

public int estrai() {
    if (dimensione == 0) {
        throw new IllegalStateException("pila vuota");
    }
    dimensione--;
    return elementi[dimensione];
}
```

La figura 8.2 ferma la pila dopo due inserimenti in una capacità di tre. Il programma `DemoPila` stampa 2, poi 12 e 7: prima osserva il numero di elementi, poi estrae nell'ordine opposto all'inserimento.



Figura 8.2 – Nella pila di capacità tre, il prossimo valore estratto è 12. Il posto libero resta disponibile per un successivo inserimento; non è un elemento della pila.

> **Concetto chiave – Un contratto, non solo un array privato.** Chi usa `Pila` conosce le operazioni e gli errori previsti, non l'indice interno del prossimo posto libero. L'array privato impedisce modifiche dirette, mentre i controlli mantengono valido il rapporto fra capacità e dimensione. Non restituiamo l'array: un chiamante non deve poter aggirare `inserisci` ed `estrai`.

Seguire la pila senza guardare dentro l'array

Partiamo da una pila nuova di capacità tre. Il suo metodo `dimensione()` restituisce zero. Dopo `inserisci(7)`, la dimensione è uno; dopo `inserisci(12)`, è due. La prima `estrai()` restituisce 12 e la dimensione torna a uno; la seconda restituisce 7 e la dimensione torna a zero. Questa sequenza insegna l'ordine LIFO senza chiedere a chi usa l'oggetto di conoscere gli indici interni. L'invariante `0 ≤ dimensione ≤ capacità` vale dopo ciascuna operazione completata correttamente. È utile fermarsi dopo ogni riga e verificare sia il valore restituito sia lo stato osservabile.

Che cosa accade se chiamiamo ancora `estrai()`? La pila è vuota. Restituire zero sarebbe ambiguo, perché zero è un valore che avremmo potuto inserire legittimamente. La classe sceglie di segnalare `pila vuota`, senza diminuire il contatore sotto zero. Che cosa accade se inseriamo un quarto elemento senza estrarne alcuno? La pila di capacità tre è piena: il metodo rifiuta l'operazione **prima** di scrivere oltre i limiti dell'array. Questi due controlli non servono soltanto a evitare eccezioni prodotte dall'array; esprimono il contratto della struttura dati con messaggi che parlano di pila piena o vuota.

Una `Pila` può essere scritta con nomi e comportamenti diversi. Per non confondere i contratti, seguiamo una storia crescente: nel capitolo 7 abbiamo nominato le operazioni, in questo capitolo fissiamo le regole, nel capitolo 12 osserviamo che cosa succede quando una chiamata non le rispetta. In questo modo il lettore non deve imparare che `pop()` ritorna zero in una pagina e che `estrai()` lancia un'eccezione in un'altra come se fossero la stessa API. Una versione che restituisce zero quando la pila è vuota resta utile come controesempio: ci fa vedere *perché* il nuovo contratto è migliore.

L'eccezione `IllegalArgumentException` dice a chi chiama il costruttore che l'argomento non rispetta il contratto. Qui serve una spiegazione minima: `throw` interrompe la costruzione e segnala il problema, evitando di consegnare un registro non valido. Nel capitolo 12 distingueremo con calma eccezioni di questo tipo, errori e casi in cui un programma può recuperare. Un messaggio come `voto fuori intervallo` è più utile di un'eccezione senza contesto, ma un'applicazione reale potrebbe includere anche il valore rifiutato quando non contiene dati sensibili.

Visibilità e responsabilità

Java offre quattro livelli rilevanti per i membri di una classe. `private` limita l'accesso alla classe che dichiara il membro; senza modificatore, l'accesso è nel package; `protected` comprende il package e, con regole ulteriori, le sottoclassi in altri package; `public` permette l'accesso dove la classe e il suo modulo sono accessibili. La specifica del controllo di accesso precisa i dettagli, soprattutto per `protected`. Un sottopackage non è lo stesso package del nome che lo precede, come abbiamo visto nel capitolo 6.

Dichiarazione del membro	Nella classe	Stesso package	Sottoclasse fuori package	Altri package
`private`	sì	no	no	no
nessun modificatore	sì	sì	no	no
`protected`	sì	sì	sì, secondo le regole dell'accesso tramite sottoclasse	no
`public`	sì	sì	sì	sì, se classe e package sono accessibili

La tabella aiuta a orientarsi, ma non autorizza a dichiarare `public` un campo soltanto perché qualcuno dovrà leggerlo. `studente()` offre il nome, `media()` offre un risultato calcolato, e `voti()` offre una copia: ciascun metodo dichiara che cosa il registro consente di osservare. Un *setter* generico che accetti qualunque array farebbe saltare il controllo se non applicasse di nuovo l'intero contratto. È spesso più chiaro offrire un'operazione del dominio, come `registraVoto`, che verifichi il singolo voto prima di aggiornare uno stato progettato per cambiare.

Che cosa vede davvero chi usa una classe

Prendiamo `private final int[] voti`. Un'altra classe non può scrivere `registro.voti[0]`, perché il nome del campo è privato. Se rendessimo il campo `public`, l'espressione diventerebbe accessibile dove la classe è accessibile, e chiunque potrebbe assegnare `-5` a una posizione: il controllo fatto nel costruttore non servirebbe più a mantenere l'invariante nel tempo. Rendere pubblico un **metodo** è diverso. `registro.media()` espone il risultato di un calcolo, lasciando al registro il controllo della rappresentazione. `registro.voti()` è ancora più delicato: restituisce dati, ma li restituisce come copia proprio per non aprire una via indiretta al campo privato.

Nel package, una dichiarazione priva di modificatore di accesso è raggiungibile dalle altre classi dello stesso package. Il nome `biblioteca.catalogo` e il nome `biblioteca.catalogo.interno` indicano però due package distinti; il secondo non acquisisce automaticamente accesso ai membri del primo. `protected` aggiunge la possibilità d'uso nelle sottoclassi, con regole precise anche quando la sottoclasse

sta in un altro package. Non leggerlo come «pubblico per tutte le sottoclassi e per chiunque le usi»: nel capitolo sull'ereditarietà vedremo perché conta anche l'espressione attraverso cui si tenta l'accesso.

Per le classi di primo livello le possibilità non sono identiche a quelle dei membri: una classe può essere `public` oppure avere accesso di package, ma non si dichiara `private` come classe di primo livello. Una classe *annidata*, invece, è un membro della classe esterna e può essere `private`. Questo dettaglio permette di nascondere anche il tipo di supporto, non soltanto i suoi campi. L'accessibilità del modulo resta un confine ulteriore: una classe pubblica in un package non esportato non diventa accessibile da ogni altro modulo.

Il caso delicato di `protected`

Supponiamo che `biblioteca.base.Registro` dichiari un campo `protected int revisioni` e che `biblioteca.speciale.RegistroStorico` estenda `Registro`. Nel codice della sottoclasse è lecito usare il campo dell'istanza corrente, per esempio `this.revisioni`. Non basta però trovarsi dentro la sottoclasse per leggere `revisioni` attraverso **qualunque** riferimento a `Registro` proveniente dall'altro package. Il linguaggio protegge l'accesso che serve alla sottoclasse per costruire il proprio comportamento; non apre il campo come se fosse pubblico. All'interno dello stesso package della classe base, invece, l'accesso `protected` è disponibile anche a classi che non la estendono. Per questo la frase «visibile ai figli» è troppo corta e può far prevedere erroneamente la compilazione.

La decisione progettuale viene prima della tabella. Se una sottoclasse ha bisogno del numero di revisioni, possiamo offrirle un metodo `protected` che esponga il dato o, meglio, un'operazione che rispetti la regola della classe base. Rendere modificabile un campo per facilitare una sottoclasse lega l'implementazione dei due tipi: una modifica del significato del campo può rompere codice che non controlliamo più. `private` è spesso un buon punto di partenza per i campi, ma la scelta di `public` e `protected` per i metodi richiede di conoscere i veri utilizzatori del contratto.

> **Per verificare – Package e sottopackage.** Disegna tre file: `catalogo.Libro`, `catalogo.Prova` e `catalogo.test.Prova`. Se `Libro` ha un metodo senza modificatore, soltanto `catalogo.Prova` può chiamarlo grazie al package condiviso. La parola `catalogo` nel nome del terzo package non gli conferisce alcun privilegio. Poi cambia il metodo in `public` e osserva che la visibilità del tipo `Libro` e l'eventuale esportazione del package restano condizioni da considerare.

Costruttori e delega

`RegistroVoti` ha due costruttori con parametri diversi: uno riceve un array, l'altro un solo voto. Questa è una forma di **sovraccarico**: il nome resta lo stesso, ma la lista dei parametri permette al compilatore di scegliere quale chiamare. Il secondo usa `this(studente, new int[] {voto})` per delegare la costruzione completa al primo. Evitiamo così due procedure indipendenti che potrebbero divergere nella validazione. I metodi possono essere sovraccaricati allo stesso modo; la scelta dipende dai parametri dichiarati, non dal solo tipo di ritorno.

Prima della delega, il secondo costruttore controlla `voto`. Questa disposizione è permessa dai **corpi di costruttore flessibili**, diventati stabili in Java 25: si possono eseguire istruzioni prima di `this(...)` o `super(...)` rispettando i vincoli sul riferimento all'istanza in costruzione. Nel nostro prologo leggiamo soltanto il parametro; non chiamiamo metodi dell'oggetto ancora incompleto. La guida ufficiale illustra anche casi più avanzati, fra cui l'assegnazione anticipata di un campo prima di `super(...)`. Qui non ne abbiamo bisogno: introdurre una possibilità nuova non rende buona ogni costruzione complicata.

> **Versione Java – Costruttori flessibili.** La forma stabile arriva in Java 25. Se compili il secondo costruttore del nostro esempio per una release precedente, il controllo prima di `this(...)` non è valido. Puoi usare la forma tradizionale spostando la validazione nel costruttore delegato; il comportamento richiesto al registro resta lo stesso. Il riepilogo ufficiale delle modifiche distingue questa forma stabile dalle sue versioni preview.

Seguire `new` senza inventare un modello di memoria

Quando scriviamo `new RegistroVoti("Ada", originali)`, Java crea una nuova istanza e ne esegue l'inizializzazione. Prima che il costruttore sia terminato correttamente, il chiamante non riceve un registro pronto da usare. Se la validazione lancia `IllegalArgumentException`, l'espressione `new` non restituisce l'istanza al chiamante. Questo è il motivo per cui il controllo iniziale è così importante: non vogliamo consegnare un oggetto con un array vuoto e sperare che `media()` si accorga più tardi del problema.

Il paragone fra `new` e `malloc` suggerisce una sequenza concreta di caricamento, allocazione e assegnazione di `this`. Questo può aiutare a intuire che serve spazio per un nuovo oggetto, ma nasconde differenze decisive. `new` invoca le regole Java di creazione e inizializzazione; `malloc` riserva memoria grezza e non conosce i costruttori Java. La posizione fisica dei dati e le ottimizzazioni della JVM non fanno parte del contratto che il nostro programma deve usare. Ci interessa invece una proprietà verificabile: due espressioni `new RegistroVoti(...)` producono due istanze distinte, ciascuna con il proprio stato.

Se non dichiari alcun costruttore, il compilatore ne fornisce uno senza parametri secondo le regole della classe. Ma appena ne dichiari uno tu, non devi aspettarti che esista ancora automaticamente quello senza parametri. Nel nostro registro esistono due costruttori espliciti e `new RegistroVoti()` non compila: mancano i dati necessari per rispettare l'invariante. Un costruttore non ha un tipo di ritorno, nemmeno `void`. Non puoi richiamarlo in seguito come `registro.RegistroVoti(...)`: per cambiare un oggetto già creato serve un normale metodo che dichiari la modifica consentita.

Due modi di inizializzare, un solo contratto

Il costruttore con un singolo voto potrebbe copiare le tre assegnazioni del costruttore con array. Sembra poco lavoro, ma alla prima nuova regola dovremmo ricordarci di modificare entrambi. La delega `this(studente, new int[] {voto})` fa attraversare a entrambe le forme lo stesso controllo. È il motivo per cui la delega migliora l'affidabilità, non soltanto la brevità del file. Il costruttore chiamato dalla delega non crea una seconda istanza: completa la costruzione della stessa.

Nella forma tradizionale di Java la chiamata a `this(...)` o `super(...)` doveva precedere ogni altra istruzione del costruttore. Java 25 permette un prologo entro limiti precisi, come nel controllo del parametro `voto` del nostro esempio. Il costruttore non può usare liberamente l'istanza incompleta in quel prologo. Se stai leggendo un progetto compilato per Java 17 o 21, la forma con controllo prima della delega non è disponibile come sintassi stabile: sposta il controllo nel costruttore delegato oppure in un'espressione di validazione compatibile con la release del progetto.

Perché il costruttore non è un metodo qualsiasi

Un metodo si può chiamare più volte su un oggetto già creato; un costruttore partecipa alla nascita di quell'oggetto. Il suo nome coincide con quello della classe e non compare un tipo di ritorno. Se scrivessimo `void RegistroVoti(...)`, avremmo dichiarato un metodo ordinario con un nome insolito, non un costruttore. La distinzione spiega anche perché l'oggetto non può essere «ricostruito» chiamando di nuovo il costruttore dopo averlo ottenuto: un cambiamento successivo richiede un metodo con un contratto proprio.

Quando due costruttori sono sovraccaricati, la lista dei parametri deve permettere di scegliere la firma. `RegistroVoti(String, int[])` e `RegistroVoti(String, int)` sono distinguibili dal secondo parametro. Cambiare soltanto il nome locale del parametro, oppure aggiungere un diverso tipo di ritorno, non crea una nuova firma valida. Il chiamante sceglie una forma in base ai dati che possiede; la classe deve far rispettare lo stesso invariante in entrambe. La delega `this(...)` è una maniera di garantire questa unità, ma occorre evitare una catena circolare in cui due costruttori si richiamano a vicenda senza arrivare a inizializzare l'oggetto.

L'accesso al costruttore è parte del contratto. Se è `public`, i chiamanti che vedono la classe possono usarlo; se è `private`, soltanto il codice della classe può chiamarlo direttamente. Una fabbrica statica può scegliere quale istanza restituire o validare i dati prima di costruirla, ma il nome «fabbrica» non obbliga a usare un Singleton. Se un dominio richiede un registro per ogni studente, condividere una sola istanza sarebbe un errore di modello anche se la sintassi di un costruttore privato lo permette. Le parole chiave governano l'accesso; il significato dell'oggetto decide la scelta.

Oggetti annidati, blocchi e istanze uniche

Classi interne, classi annidate statiche, classi locali, blocchi di inizializzazione e Singleton richiedono attenzione anche al di fuori dell'esempio del registro. Sono strumenti reali, ma non sono passaggi obbligatori per proteggere l'array del registro. Una **classe annidata** è dichiarata dentro un'altra classe; una classe interna non statica ha un legame con un'istanza della classe esterna, mentre una classe annidata `static` non richiede quell'istanza. La differenza diventa utile quando un tipo di supporto appartiene davvero all'implementazione dell'oggetto esterno. Le classi locali e anonime torneranno nella parte sulle interfacce, dove avranno un problema concreto da risolvere.

Un **blocco di inizializzazione** esegue istruzioni quando viene inizializzata la classe o una sua istanza. Se un costruttore esprime meglio il lavoro, il blocco aggiunge soltanto un passaggio da cercare. L'ordine della costruzione fra classe base e sottoclasse sarà provato nel capitolo 9; i blocchi statici e quelli di istanza richiedono inoltre di distinguere inizializzazione della classe da creazione dell'oggetto.

Un esperimento sull'ordine di inizializzazione

Stampiamo un messaggio da un blocco `static`, uno da un blocco di istanza e uno dal costruttore. Prima di eseguirlo, possiamo prevedere il risultato. All'inizializzazione della classe, i suoi inizializzatori statici vengono eseguiti nell'ordine previsto dal sorgente; in un semplice avvio con `main`, ciò avviene prima di costruire le istanze. Ogni volta che creiamo un oggetto, gli inizializzatori di istanza della sua classe partecipano alla costruzione prima del corpo del costruttore. Con due oggetti della stessa classe vedremo quindi il messaggio statico una volta, e i messaggi di istanza e del costruttore due volte ciascuno.

Non diciamo che il blocco statico venga eseguito «ogni volta che la classe viene caricata» come se caricamento e inizializzazione fossero lo stesso evento. La JVM distingue queste fasi, e una classe può essere caricata prima che sia inizializzata. Per il programma didattico ci basta osservare che il codice statico non è legato a una singola istanza. Se puoi inizializzare un campo con una semplice espressione o nel costruttore, preferisci la forma che rende più facile seguire il valore; un blocco separato si giustifica quando ha un lavoro riconoscibile.

Il programma `DemoBlocchi` propone una prova con **due** blocchi statici, **due** blocchi di istanza e due creazioni. Prima di eseguirlo, separa su carta ciò che appartiene alla classe da ciò che appartiene a ciascun oggetto.

```
public class DemoBlocchi {
    static {
        System.out.println("statico 1");
    }

    static {
        System.out.println("statico 2");
    }

    {
        System.out.println("istanza 1");
    }

    {
        System.out.println("istanza 2");
    }

    public DemoBlocchi() {
        System.out.println("costruttore");
    }

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

L'output comincia con `statico 1` e `statico 2`, una sola volta nell'avvio mostrato. Segue `istanza 1`, `istanza 2`, `costruttore` per il primo oggetto; la stessa terna torna per il secondo. Non leggiamo il file semplicemente dall'alto verso il basso come una lista di istruzioni che viene eseguita una volta. Distinguiamo l'inizializzazione della classe, che precede l'uso attivo nel nostro caso, dalla costruzione ripetuta delle istanze. I blocchi di uno stesso tipo seguono l'ordine del sorgente insieme alle inizializzazioni dei campi pertinenti. Se una classe estende un'altra, interviene anche la sequenza di inizializzazione della base; il capitolo 9 la renderà osservabile con un esempio separato.

I blocchi di inizializzazione non sono una novità di Java 11: erano disponibili già prima di quella release. Saperlo è importante perché chi mantiene programmi precedenti a Java 11 potrebbe altrimenti credere di non poterli usare. Più importante della data è però il loro scopo. Se l'inizializzazione è una semplice assegnazione, scriverla accanto al campo rende il codice più facile da seguire. Un blocco con effetti complessi richiede una ragione che il lettore del file possa riconoscere.

Un costruttore `private` può impedire ad altro codice di creare direttamente istanze. È uno degli ingredienti del pattern *Singleton*, che offre una sola istanza condivisa. Nel registro dei voti sarebbe sbagliato: ogni studente deve avere il proprio registro. Una sola istanza non rende automaticamente sicuro o facile da verificare uno stato globale; useremo questa soluzione soltanto quando il problema la richiederà.

Il Singleton storico e la domanda sulla condivisione

Un Singleton può usare un campo statico inizialmente `null` e un metodo `getInstance()` che costruisce l'oggetto alla prima chiamata. In un programma eseguito da un solo thread la sequenza sembra chiara: la prima chiamata crea l'istanza, le successive la restituiscono. Con più thread, però, due chiamate possono osservare entrambe il campo ancora nullo e costruire due istanze. Il problema non si risolve rendendo soltanto `private` il costruttore: quella parola limita chi può invocarlo, non coordina le esecuzioni concorrenti. Il capitolo sulla concorrenza spiegherà gli strumenti per ragionare sull'ordine fra thread; qui evitiamo di stampare l'implementazione ingenua come modello sicuro da copiare.

> **Approfondimento – Che cosa vuol dire atomico.** Un'operazione è *atomica* rispetto agli altri thread quando il suo effetto viene osservato come un solo passaggio indivisibile. Nel Singleton ingenuo, «controlla se il campo è `null`» e «assegna l'istanza appena costruita» sono due passaggi distinti: un altro thread può intervenire fra i due. Il costruttore privato non rende atomica la coppia. Per progettare l'accesso concorrente occorrono regole di sincronizzazione o una forma di inizializzazione che offra già la garanzia necessaria; la sola lettura della sequenza nel sorgente non basta a provarla.

Se il dominio richiede davvero un'unica istanza di un servizio senza parametri di configurazione per ogni uso, un enum a una costante può rappresentare quella scelta con una forma semplice. Anche in quel caso, il servizio può avere **stato modificabile**, e l'unicità dell'istanza non rende automaticamente corrette le modifiche concorrenti. Un'altra possibilità è creare esplicitamente un oggetto nel punto di avvio dell'applicazione e passarlo ai componenti che ne hanno bisogno. Chi legge la firma dei costruttori vede così la dipendenza, e una prova può consegnare un'istanza diversa. Questa pratica è spesso chiamata *iniezione delle dipendenze*: il termine indica qui il passaggio esplicito di un collaboratore, senza richiedere alcun framework.

> **Concetto chiave – Una sola istanza non è sempre un'invariante.** Il registro di un cliente, la pila di un calcolo e il libro sfogliato nel capitolo precedente devono poter esistere più volte. Prima di scegliere un Singleton, formula la regola del dominio che richiederebbe **esattamente una** istanza e chiediti se deve valere per il processo, per una sessione, per una richiesta o per un utente. Se la risposta cambia con il contesto, una variabile globale nascosta dietro `getInstance()` rende il modello meno chiaro.

Un costruttore `protected` è ancora un'altra scelta: permette l'uso nel package e nelle sottoclassi secondo le regole di accesso, senza aprire la costruzione indiscriminatamente. Serve quando il tipo è progettato per essere esteso e la sottoclasse deve inizializzare la parte ereditata. Non scegliamo il modificatore per imitare il Singleton; lo scegliamo dopo aver deciso **chi** può creare o estendere il tipo. Nel prossimo capitolo vedremo questa decisione nella classe base `Veicolo`.

Tre forme di classe annidata, tre legami diversi

Una **classe interna membro** non statica è associata a un'istanza della classe esterna. Immagina una `Biblioteca` che definisca al proprio interno `Scansione`: ogni oggetto `Scansione` potrebbe riferirsi alla biblioteca a cui appartiene. Dentro `Scansione`, `this` indica la scansione corrente, mentre `Biblioteca.this` indica l'istanza esterna associata. Questa distinzione fra i due `this` è essenziale per leggere le autoreferenze. Se due biblioteche creano ciascuna una scansione, i riferimenti esterni sono diversi; non esiste una sola biblioteca globale implicita.

Il legame di un'**istanza** interna con quella esterna non vieta di dichiarare membri `static` nel tipo interno. Il divieto di dichiarare metodi statici in una classe interna apparteneva alle versioni anteriori a Java 16, non a Java 25. Un metodo `static` dichiarato nella classe interna appartiene al tipo e non può usare implicitamente `Biblioteca.this` come farebbe un metodo di istanza. La JLS §8.1.3 distingue precisamente le due cose. Nel nostro esempio `Dettaglio` usa l'istanza esterna; `Formato` è invece un tipo annidato statico perché non ha bisogno di quel legame.

La figura 8.3 ridisegna quel rapporto con due istanze esterne. Ogni dettaglio conosce la biblioteca che lo ha creato; non c'è una sola biblioteca condivisa da tutti i dettagli.



Figura 8.3 – Ogni istanza interna `Dettaglio` è legata alla propria istanza esterna. Una classe annidata `static` non conserva questo legame implicito.

Una **classe annidata statica** appartiene alla struttura del sorgente della classe esterna. Le sue istanze non ricevono un riferimento implicito a un oggetto esterno. Nel programma che segue, `Formato` è dichiarata dentro la classe della biblioteca ma lavora soltanto sui dati che riceve. Il legame fra i due tipi è visibile nel codice, senza obbligare ogni `Formato` a ricordare una particolare biblioteca. Possiamo creare più oggetti di quel tipo: `static` qui non significa «istanza unica».

Una **classe locale** viene dichiarata dentro un blocco, per esempio nel corpo di un metodo. Il suo nome è disponibile nell'ambito di quel blocco. Se usa una variabile locale del metodo che la contiene, quella variabile deve essere `final` oppure

effettivamente final: può cioè essere assegnata una volta e non modificata dopo. Java accetta una variabile non modificata anche senza il modificatore `final`: ciò che conta è che sia effettivamente final. Le classi anonime sono un'altra forma locale che useremo quando incontreremo le interfacce e avremo un esempio che ne motivi l'uso.

Il criterio di scelta non è «annidare rende sempre il file più corto». Una classe di supporto privata e piccola, usata soltanto dall'esterna, può rendere più chiaro il confine. Una classe che cresce, ha un contratto autonomo o viene usata da più parti del programma può meritare un file proprio. Anche le classi annidate partecipano alle regole di accesso e possono accedere ai membri privati dell'esterna secondo la loro forma; questo non autorizza a creare dipendenze difficili da seguire.

Per vedere i tre legami in esecuzione, il file `DemoClassiAnnidate.java` contiene un caso completo. I nomi sono volutamente semplici: qui osserviamo la relazione fra istanze, non stiamo ancora progettando una vera biblioteca.

```
public class DemoClassiAnnidate {
    private final int id;

    public DemoClassiAnnidate(int id) {
        this.id = id;
    }

    public class Dettaglio {
        public String descrizione() {
            return "biblioteca: " + DemoClassiAnnidate.this.id;
        }
    }

    public static class Formato {
        public String nome() {
            return "catalogo";


        }
    }

    public String etichettaLocale() {
        String prefisso = "id=";
        class Etichetta {
            String testo() {
                return prefisso + id;
            }
        }
        return new Etichetta().testo();
    }

    public static void main(String[] args) {
        DemoClassiAnnidate biblioteca = new DemoClassiAnnidate(7);
        Dettaglio dettaglio = biblioteca.new Dettaglio();
```

```
        Formato formato = new Formato();
        System.out.println(dettaglio.descrizione());
        System.out.println(formato.nome());
        System.out.println(biblioteca.etichettaLocale());
    }
}
```

La chiamata `biblioteca.new Dettaglio()` associa il nuovo `Dettaglio` a quella particolare `biblioteca`. Dentro la classe interna, `DemoClassiAnnidate.this.id` indica senza ambiguità il campo dell'istanza esterna. `new Formato()` non ha bisogno della biblioteca: `Formato` è annidata staticamente e il suo metodo non legge lo stato di una `DemoClassiAnnidate`. La classe locale `Etichetta` è visibile soltanto nel metodo `etichettaLocale`; legge `prefisso`, che non viene riassegnato ed è quindi effettivamente `final`. Il programma stampa, nell'ordine, `biblioteca: 7`, `catalogo` e `id=7`.

Nel disegno storico delle classi interne comparivano due istanze della classe esterna, ciascuna con il proprio oggetto interno. Possiamo ricostruire la situazione senza immaginare una variabile globale nascosta: se creiamo `prima = new DemoClassiAnnidate(7)` e `seconda = new DemoClassiAnnidate(9)`, allora `prima.new Dettaglio()` produce un dettaglio che, tramite `DemoClassiAnnidate.this`, legge 7; `seconda.new Dettaglio()` ne produce uno che legge 9. La classe `Dettaglio` è una sola dichiarazione nel sorgente, ma ogni sua istanza interna è associata a una particolare istanza esterna. È proprio questa associazione a giustificare la forma non statica quando il comportamento deve usare lo stato esterno.

Se `Formato` deve soltanto restituire la parola `catalogo`, non ha ragione di portare con sé un legame implicito a una biblioteca. La forma annidata statica esprime meglio questa indipendenza. Non confondiamo però il modificatore con i campi statici studiati nel capitolo 7: possiamo creare molti oggetti `Formato`, ciascuno con il proprio eventuale stato di istanza. `static` qui descrive il rapporto della **dichiarazione annidata** con la classe esterna, non limita a uno il numero di oggetti.

La classe locale `Etichetta` risponde a una terza esigenza: serve soltanto mentre eseguiamo `etichettaLocale()`, quindi il suo nome rimane nel blocco del metodo. Il valore `prefisso` usato al suo interno è una variabile locale del metodo che la circonda. Java richiede che non venga riassegnata dopo la sua inizializzazione; altrimenti non potremmo leggere quel valore come una cattura stabile. Non occorre scrivere la parola `final` se il codice rispetta già la regola. Una classe locale molto lunga o riutilizzata in più metodi, tuttavia, rende difficile leggere il metodo: in quel caso vale la pena valutare un tipo membro o un file separato. La forma più interna non è automaticamente la più incapsulata nel senso utile al lettore.

Una piccola prova modifica solo `prefisso` dopo la dichiarazione, aggiungendo per esempio `prefisso = "numero=";` prima della classe locale. Il programma non compila più, perché la classe locale cattura una variabile che non è effettivamente finale. Se invece cambi l'identificatore passato al costruttore della biblioteca, cambiano le righe che leggono lo stato dell'istanza esterna, ma non `catalogo`. Questa differenza è il motivo per scegliere con attenzione fra classe interna e classe annidata statica.

Per verificare

Riprendi prima `DemoClassiAnnidate`: crea due biblioteche con identificatori diversi e un `Dettaglio` per ciascuna. Prevedi quali righe cambiano e spiega perché `Formato` continua a stampare `catalogo`. Poi riassegna `prefisso` prima della dichiarazione della classe locale e verifica che il compilatore rifiuti la cattura. Qui la prova deve distinguere il legame con l'istanza esterna da quello con una variabile locale.

Torna quindi alla copia difensiva degli array: togli una chiamata a `clone()` per volta e prevedi la media stampata dopo le modifiche agli array esterni. Quale direzione di copia manca in ciascun caso? L'attività del capitolo 8 propone i comandi e i risultati con cui controllare la risposta. I due esperimenti richiedono lo stesso metodo di lettura: individua quale oggetto o valore resta raggiungibile, poi verifica la previsione eseguendo il programma.

9. Ereditarietà, composizione e gerarchie sealed

Quando «è un» significa qualcosa

Nel capitolo 2 abbiamo provato a leggere una relazione fra classi come una frase: «un'auto è un veicolo» suona sensato; «un'auto è un motore» no, perché il motore è una parte dell'auto. Questa distinzione guida la scelta fra **ereditarietà** e **composizione**. La prima permette a una classe di specializzarne un'altra e di essere usata dove è atteso il tipo generale. La seconda costruisce un oggetto collegando altri oggetti con responsabilità proprie. Nessuna delle due è una scorciatoia garantita per scrivere meno codice.

La **classe base**, detta anche *superclasse*, stabilisce un contratto che le **sottoclassi** devono rispettare. Se una funzione accetta un `Veicolo`, deve poter lavorare con un'`Auto` o una `Bicicletta` senza verificare in continuazione quale delle due sia arrivata. Quando una sottoclasse cambia il significato atteso di un'operazione, la gerarchia diventa fragile anche se compila. Riuso e manutenzione locale sono benefici possibili, ma vanno dimostrati caso per caso.

Nel sorgente completo, `Veicolo` dichiara il metodo `descrizione()`. `Auto` e `Bicicletta` lo realizzano secondo il proprio significato; `Auto` **ha un** `Motore`, rappresentato come record. La figura 9.1 separa visivamente i due rapporti prima di guardare le parole chiave.

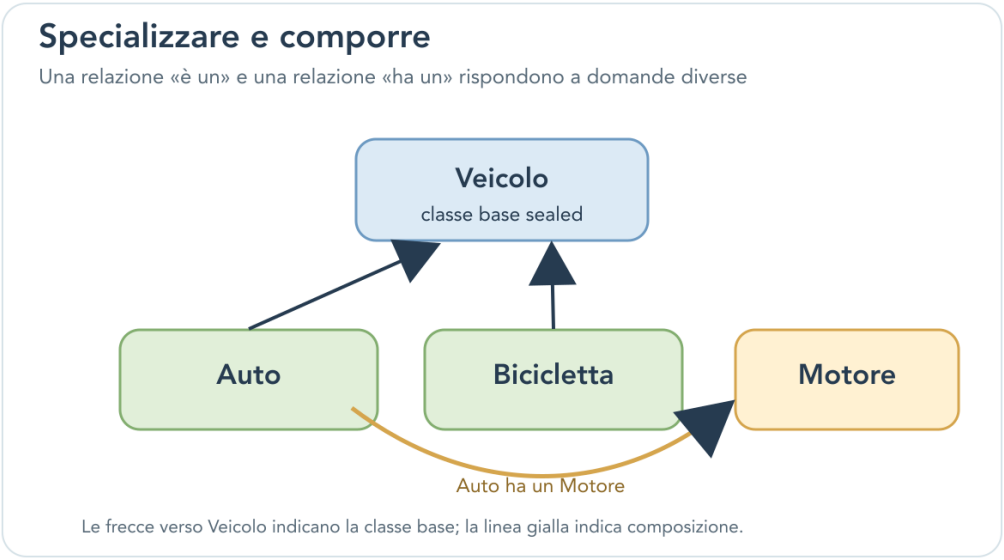


Figura 9.1 – `Auto` e `Bicicletta` sono specializzazioni di `Veicolo`; `Auto` contiene un `Motore`. Le frecce delle sottoclassi puntano alla classe base. Il collegamento con il motore indica una parte, non un altro tipo di veicolo.

Dichiarare la relazione

Il nucleo della dichiarazione è questo. `abstract` dice che `Veicolo` definisce un modello incompleto: non possiamo costruire un `new Veicolo(...)` perché il metodo astratto non ha ancora un corpo. `extends`, usato dalle sottoclassi nel programma completo, indica la classe base. Ogni sottoclasse concreta fornisce la propria implementazione del metodo.

```
sealed abstract class Veicolo permits Auto, Bicicletta {
    private final String nome;
    protected Veicolo(String nome) {
        this.nome = nome;
    }
    public String nome() {
        return nome;
    }
    public abstract String descrizione();
}
```

`Auto` non eredita da `Motore`: lo conserva in un campo. Se cambiassimo il modo di descrivere l'alimentazione, il motore rimarrebbe una responsabilità distinta. Nel sorgente completo il costruttore di `Auto` chiama `super(nome)` per far inizializzare alla classe base la propria parte dello stato, poi assegna il campo `motore`. Chiamare un metodo ridefinibile dal costruttore della base può essere pericoloso, perché la sottoclasse potrebbe non aver completato l'inizializzazione; qui il costruttore della base conserva soltanto il nome.

Il modificatore `protected` del costruttore consente alle sottoclassi di invocarlo. Per i membri in generale, `protected` comprende anche l'accesso dallo stesso package e impone regole particolari fuori package: la tabella del capitolo 8 è un orientamento, mentre la specifica di accesso dà la regola completa. Un sottopackage, da solo, non ottiene i privilegi del package che lo precede nel nome.

Il breve programma sull'ordine dei costruttori stampa prima `base` e poi `figlio: 7`. La costruzione della parte ereditata precede il completamento del costruttore della sottoclasse. Le inizializzazioni dei campi e i blocchi di inizializzazione aggiungono passaggi a quest'ordine: non confondere l'ordine in cui il sorgente mostra i campi con una chiamata sicura a metodi ridefinibili dal costruttore della base. La specifica sulla creazione delle istanze descrive la sequenza completa.

La ruota e il raggio: salvare la lezione, correggere il modello

L'esempio con `Circonferenza` e `Ruota` mostra che il costruttore della sottoclasse non può assegnare direttamente un campo privato della base. La lezione sull'accesso è valida; il modello «una ruota è una circonferenza» è meno convincente, perché una ruota è un oggetto fisico e una circonferenza una misura o una figura geometrica. Il piccolo esperimento è una prova della sintassi, con questo limite del modello. In un'applicazione su veicoli reali sarebbe spesso più naturale che `Ruota` **abbia un** valore per il raggio e usi una funzione geometrica, senza ereditare da `Cerchio`.

Nel programma `DemoRuota`, la classe base conserva `raggio` come campo privato e lo convalida nel proprio costruttore. La sottoclasse chiama `super(raggio)` invece di provare ad assegnare il campo altrui. Il codice è completo e consente di seguire la

costruzione senza lasciare una riga intenzionalmente non compilabile dentro l'esempio positivo.

```
class Cerchio {
    private final double raggio;

    protected Cerchio(double raggio) {
        if (!Double.isFinite(raggio) || raggio <= 0) {
            throw new IllegalArgumentException("raggio non valido");
        }
        this.raggio = raggio;
    }

    public final double raggio() {
        return raggio;
    }

    public double circonferenza() {
        return 2 * Math.PI * raggio;


    }
}

class Ruota extends Cerchio {
    Ruota(double raggio) {
        super(raggio);
    }

    public double diametro() {
        return 2 * raggio();
    }
}

public class DemoRuota {
    public static void main(String[] args) {
        Ruota ruota = new Ruota(5);

        System.out.println("raggio: " + ruota.raggio());
        System.out.println("diametro: " + ruota.diametro());
        System.out.println("circonferenza > diametro: "
                + (ruota.circonferenza() > ruota.diametro()));
    }
}
```

L'output è `raggio: 5.0`, `diametro: 10.0`, `circonferenza > diametro: true`. Il diametro è **due volte il raggio**; con raggio 5 otteniamo 10. La lunghezza della circonferenza è `2 × π × raggio`, dove π è il rapporto costante fra lunghezza della circonferenza e diametro; per questo è maggiore del diametro positivo. Usiamo `Math.PI` invece di arrotondare π a `3.14` nello stato di ogni oggetto. Il programma stampa un confronto booleano, non una lunga sequenza di cifre decimali che distrarrebbe dalla relazione fra base e sottoclasse.

Il controllo `Double.isFinite(raggio)` rifiuta anche `NaN` e gli infiniti: limitarsi a `raggio <= 0` non rifiuterebbe `NaN`, perché i confronti ordinati con quel valore sono falsi. Non è necessario conoscere tutti i dettagli della virgola mobile per leggere l'esempio; basta osservare che la classe base protegge il proprio dato prima di

consegnare alla sottoclasse un oggetto completato. La chiamata `raggio()` è lecita perché il metodo è pubblico, mentre `Ruota` non può scrivere `this.raggio = ...` perché il campo appartiene all'implementazione privata di `Cerchio`.

> **Nota bene – Un esempio sintattico non è sempre un buon dominio.** Il codice di `DemoRuota` insegna `super(...)` e l'accesso ai membri; non invita a derivare ogni ruota da una figura geometrica. Separare le due valutazioni evita di trasformare un espediente didattico in una regola di progettazione. Quando la relazione «è un» non regge, conserva il calcolo ma cambia il rapporto fra le classi.

> **Nota bene – Una sola classe base.** Una classe Java può estendere direttamente una sola classe. Questo non significa che possa svolgere un solo ruolo: nel capitolo 13 incontreremo le interfacce. Comporre oggetti resta un'altra possibilità, spesso più semplice quando il rapporto naturale è «ha un». Due diagrammi di ereditarietà singola e multipla mostrano la forma delle relazioni, ma non bastano a scegliere un modello adatto al dominio.

L'ordine completo di inizializzazione, osservato due volte

Il programma `DemoOrdine` mostrava soltanto i due costruttori. Aggiungiamo blocchi statici e di istanza nella base e nella sottoclasse. Seguiamo la prova nel sorgente `DemoOrdineCompleto`, creando **due** oggetti: il secondo ci permette di separare ciò che avviene una volta per le classi da ciò che avviene per ogni istanza.

```
class BaseOrdineCompleto {
    static {
        System.out.println("base statica");
    }

    {
        System.out.println("base istanza");
    }

    BaseOrdineCompleto() {
        System.out.println("base costruttore");
    }
}

class FigliaOrdineCompleto extends BaseOrdineCompleto {
    static {
```

```
        System.out.println("figlia statica");
    }

    {
        System.out.println("figlia istanza");
    }

    FigliaOrdineCompleto() {
        System.out.println("figlia costruttore");
    }
}

public class DemoOrdineCompleto {
    public static void main(String[] args) {
        new FigliaOrdineCompleto();
        new FigliaOrdineCompleto();


    }
}
```

All'avvio della prima costruzione compaiono `base statica` e `figlia statica`. Segue la terna della parte base: `base istanza`, `base costruttore`. Poi la terna della sottoclasse, qui composta da `figlia istanza` e `figlia costruttore`. Con il secondo `new` le due righe statiche **non** tornano: si ripetono le due righe di istanza e i due costruttori, nello stesso ordine relativo. Il blocco di istanza della base precede il corpo del suo costruttore; quello della figlia precede il corpo del costruttore della figlia. Se guardassimo soltanto l'ordine di apparizione delle classi nel file, non potremmo dedurre correttamente questo risultato.

La prova mostra l'inizializzazione delle classi nel percorso di avvio scelto, senza equipararla al semplice caricamento dei file `.class` da parte della JVM. Mostra inoltre che la parte base viene costruita prima della parte specifica della figlia. Non dobbiamo usare un blocco statico per inizializzare un campo di istanza, né aspettarci che un costruttore della base conosca già i campi che il costruttore della figlia assegnerà dopo. Per un programma reale con molte classi e dipendenze, tenere semplice l'inizializzazione riduce le sorprese; qui l'abbiamo resa volutamente verbosa per vedere ogni passaggio.

Progettare prima la classe base

Prima di costruire una classe `Veicolo` e specializzarla, consideriamo il problema che deve risolvere. Che cosa deve poter chiedere il codice a **qualunque** veicolo? Un nome e una descrizione possono appartenere al contratto comune. Il numero di porte non può: una bicicletta non ne ha, e aggiungerle un valore fittizio soltanto per soddisfare la classe base produrrebbe un modello ingannevole. Se un'operazione non ha un significato coerente per tutte le sottoclassi, probabilmente non appartiene alla base.

Una classe base può offrire un comportamento già completo oppure dichiarare un metodo astratto che ogni sottoclasse concreta deve realizzare. `Veicolo.nome()` restituisce il dato comune; `descrizione()` è astratto perché il tipo concreto decide che cosa dire. Non ogni differenza richiede però un metodo astratto. Se il programma deve solo conservare nome e categoria, una classe semplice o un

record possono bastare. L'ereditarietà si giustifica quando un codice client ha davvero bisogno di lavorare con il tipo comune e con varianti sostituibili.

Confrontiamo ereditarietà multipla e singola. Java permette a una classe di estendere direttamente una sola superclasse, evitando fra l'altro l'ambiguità di scegliere fra due implementazioni ereditate dello stesso membro da due classi. Questo non impedisce di riunire capacità: un'`Auto` può *avere* un `Motore` e più avanti potrà implementare più interfacce. La domanda progettuale rimane quale relazione racconti meglio il problema.

Dalla freccia del diagramma al contratto del codice

La freccia dell'ereditarietà è un aiuto visivo, ma per usarla bene occorre trasformarla in una frase verificabile. Se `Auto extends Veicolo`, ogni punto del programma che richiede un `Veicolo` può ricevere un'`Auto`. Questa frase ha una conseguenza concreta: il codice che lavora sul tipo base non dovrebbe conoscere il numero di ruote, il carburante o altri dettagli della sottoclasse per svolgere un'operazione promessa da `Veicolo`. Se lo costringiamo a un cast prima di ogni chiamata, abbiamo disegnato una freccia senza aver costruito un vero comportamento comune.

Viceversa, la relazione «ha un» non rende intercambiabili i tipi. Un'`Auto` possiede un `Motore`, ma non possiamo passare l'auto a un metodo che richiede un `Motore`. Possiamo invece chiedere all'auto un'operazione che usa il motore, oppure passarle un `Motore` nel costruttore. Questa separazione protegge le responsabilità: se domani il modo di produrre energia cambia, il contratto generale di `Veicolo` non deve necessariamente cambiare insieme alla rappresentazione del motore. La composizione non è sempre superiore; qui è la relazione che descrive correttamente l'oggetto.

> **Per capire – Una prova di sostituibilità.** Immagina un metodo `stampaScheda(Veicolo v)` che chiama `v.nome()` e `v.descrizione()`. Provalo mentalmente con un'auto e una bicicletta. Se per farlo funzionare devi chiedere se `v` è un'auto prima di stampare la descrizione, rivedi la responsabilità di `descrizione()`. Se invece devi mostrare la cilindrata, quel dato appartiene davvero a tutti i veicoli? Una domanda sul dominio evita di aggiungere alla base metodi privi di senso per alcune sottoclassi.

Sovraccarico e ridefinizione: una prova con le chiamate

Supponiamo che `Veicolo` dichiari `descrizione()` e `descrizione(boolean dettagliata)`. Sono due metodi **sovraccaricati**: hanno lo stesso nome ma parametri diversi. Se `Auto` dichiara `descrizione()` con una firma compatibile, **ridefinisce** il primo. Può continuare a ereditare il secondo, oppure ridefinire anche quello. Una chiamata `mezzo.descrizione(true)` viene selezionata dal compilatore in base ai tipi degli argomenti e ai metodi disponibili nel tipo dichiarato; solo dopo, se il metodo di istanza selezionato è ridefinito, il runtime sceglie il corpo associato all'oggetto effettivo.

Questa separazione evita un equivoco comune: l'oggetto effettivo non fa comparire magicamente nel tipo `Veicolo` tutti i metodi che `Auto` ha aggiunto. Se `Auto` dichiara `apriBagagliaio()`, `Veicolo mezzo = new Auto(...)` non consente

`mezzo.apriBagagliaio()` finché il compilatore vede soltanto `Veicolo`. Possiamo cambiare il tipo del riferimento, verificare che l'oggetto sia un'`Auto`, oppure, se l'operazione deve valere per tutti, ripensare il contratto comune. Il cast non cambia l'oggetto: cambia il tipo attraverso cui chiediamo al compilatore di trattare quel riferimento, e può essere controllato a runtime.

Un esempio può far partire un veicolo a `1 km/h` e poi aggiungere un'altra versione di `muovi` che riceve la velocità. Seguiamo quella progressione in un programma completo, con i nomi necessari a distinguerlo dall'altro esempio del capitolo:

```
class MezzoInMovimento {
    private int velocita;

    public void muovi() {
        muovi(1);
    }

    public void muovi(int nuovaVelocita) {
        if (nuovaVelocita < 0) {
            throw new IllegalArgumentException("velocita negativa");
        }
        velocita = nuovaVelocita;
        System.out.println(descrizione() + ": " + velocita);
    }

    public void ferma() {


        muovi(0);
    }

    public String descrizione() {
        return "Veicolo";
    }
}

class AutoInMovimento extends MezzoInMovimento {
    @Override
    public String descrizione() {
        return "Auto";
    }
}

public class DemoMovimento {


    public static void main(String[] args) {
        MezzoInMovimento mezzo = new AutoInMovimento();
        mezzo.muovi();
        mezzo.muovi(10);
        mezzo.ferma();
    }
}
```

Il metodo senza argomenti delega a quello con un `int`, quindi la regola che aggiorna la velocità sta in un solo punto. `ferma()` usa la stessa regola passando zero. Il controllo su una velocità negativa protegge il significato del dato: un campo privato senza controllo non basta a dare un significato al dato. Il `main` conserva

un'`AutoInMovimento` in una variabile di tipo base e chiama prima `muovi()`, poi `muovi(10)` e infine `ferma()`. L'output è `Auto: 1`, `Auto: 10`, `Auto: 0`. La scelta fra le due firme `muovi` dipende dagli argomenti scritti; la parola `Auto` nelle tre righe dipende invece dalla ridefinizione di `descrizione()` nell'oggetto effettivo. Nello stesso programma vediamo dunque i due meccanismi senza confonderli.

Prova a sostituire `new AutoInMovimento()` con `new MezzoInMovimento()`. Le chiamate restano lecite, ma le tre righe cominciano con `Veicolo`: il tipo dichiarato della variabile non è cambiato, è cambiato l'oggetto creato. È la prova concreta della differenza fra selezione del metodo e scelta del corpo eseguito. Il file usa `System.out` per osservare lo stato; in un'applicazione reale potremmo anche restituire la velocità e lasciare la presentazione a un altro componente.

Leggiamo le tre chiamate con due domande separate. In `mezzo.muovi()`, il compilatore trova la firma senza parametri nel tipo `MezzoInMovimento`; quel corpo delega a `muovi(1)`. In `mezzo.muovi(10)`, trova la firma con un `int`. In `mezzo.ferma()`, trova `ferma()` e il corpo chiama `muovi(0)`. Tutte e tre, a un certo punto, invocano `descrizione()`. Qui entra la ridefinizione: il metodo di istanza eseguito per `descrizione()` dipende dall'oggetto effettivo `AutoInMovimento`, quindi la parola stampata è `Auto`. Il risultato non nasce dal fatto che il compilatore abbia cambiato la firma di `muovi` durante l'esecuzione.

Sovraccaricare un metodo richiede parametri distinguibili; cambiare soltanto il tipo restituito non basta. Due dichiarazioni `int misura()` e `double misura()` nella stessa classe non offrono al chiamante una scelta affidabile basata sul contesto di ritorno e non sono un sovraccarico valido. Una chiamata senza argomenti e una con un `int` sono invece distinguibili. Il sovraccarico dei **costruttori** segue lo stesso principio delle liste di parametri: `RegistroVoti(String, int)` e `RegistroVoti(String, int[])` del capitolo 8 scelgono due ingressi per costruire lo stesso tipo, senza ereditare costruttori come se fossero metodi.

La ridefinizione, al contrario, richiede una firma compatibile con un metodo ereditato e riguarda il corpo selezionato per l'istanza. `@Override` rende controllabile questa intenzione. Se per errore scrivessimo `descrizione(String dettaglio)` in `AutoInMovimento`, avremmo aggiunto un nuovo sovraccarico; la chiamata `descrizione()` continuerebbe a usare il corpo della base. Con `@Override` sopra la firma sbagliata il compilatore interrompe il lavoro e ci costringe a correggere il programma. È una piccola verifica che evita un difetto difficile da riconoscere guardando soltanto un output diverso da quello atteso.

> **Approfondimento – Metodi `static` e campi.** Il dispatch dinamico descritto qui vale per i metodi di istanza ridefinibili. Un metodo `static` dichiarato con lo stesso nome in base e sottoclasse viene nascosto, non ridefinito nello stesso senso; un campo con lo stesso nome segue ancora le regole del nascondimento. Per questo non deduciamo il comportamento di ogni membro dalla prova con `descrizione()`. La distinzione diventerà più importante quando leggeremo API con factory statiche e gerarchie estese.

Il tipo scritto e l'oggetto reale

Nel `main` del programma troviamo:

```
Veicolo mezzo = new Auto("A", new Motore("elettrico"));
System.out.println(mezzo.descrizione());
```

Il **tipo dichiarato** della variabile `mezzo` è `Veicolo`. L'oggetto creato è un'`Auto`, quindi la sua **classe effettiva** è `Auto`. Il compilatore permette di chiamare `descrizione()` perché il metodo fa parte del contratto di `Veicolo`. Durante l'esecuzione viene scelto il corpo ridefinito in `Auto`, e la riga stampata è `Auto A con motore elettrico`. Questa scelta del metodo di istanza in base all'oggetto effettivo è il **dispatch dinamico**. Non implica che la variabile cambi tipo dichiarato, né che tutti i metodi si comportino allo stesso modo: i metodi `static`, per esempio, non sono ridefiniti con questo meccanismo.

> **Nota bene – Il caso dei primitivi.** La distinzione fra tipo dichiarato del riferimento e classe effettiva dell'oggetto serve quando una variabile può indicare istanze di sottoclassi diverse. Una variabile `int velocita` non contiene un riferimento a un oggetto con una classe più specifica nascosta: contiene un valore del tipo primitivo `int`. Una conversione numerica può produrre un valore di un altro tipo, e il boxing può creare o ottenere un wrapper, ma non c'è un `int` che diventi in esecuzione la sottoclasse di un altro `int`. Per questo i diagrammi di `Veicolo` e `Auto` non vanno applicati meccanicamente ai primitivi del capitolo 4.

La figura 9.2 conserva insieme le due informazioni che gli schemi storici distribuivano su pagine diverse: il tipo disponibile al compilatore e la classe dell'oggetto effettivo. La freccia indica il riferimento, non un cast che abbia cambiato l'oggetto.



Figura 9.2 – `mezzo` è dichiarato `Veicolo` e indica un'`Auto`. `descrizione()` è nel contratto comune; `alimentazione()` richiede un riferimento noto come `Auto`, per esempio dopo un controllo `instanceof`.

`@Override` fa chiedere al compilatore di controllare che `descrizione()` ridefinisca davvero un metodo della classe base. Se sbagliamo nome o parametri, la verifica

segnala il problema. La **ridefinizione** (*overriding*) riguarda un metodo ereditato e il comportamento scelto per l'oggetto. Il **sovraccarico** (*overloading*) dichiara invece più metodi con lo stesso nome e parametri diversi, come i due costruttori di `RegistroVoti` del capitolo 8. Confonderli rende difficile prevedere quale corpo verrà eseguito.

Che cosa si eredita e che cosa resta protetto

Una sottoclasse eredita membri secondo le regole del linguaggio, ma questo non significa che ogni campo della base diventi accessibile per nome nel suo codice. Il campo `nome` di `Veicolo` è `private`: `Auto` non può leggerlo direttamente. Usa `nome()`, il metodo che la base espone, e così dipende dalla promessa «fornisci il nome» invece dalla posizione o dal nome del campo. La base può cambiare in seguito il modo in cui conserva il nome senza obbligare la sottoclasse a conoscere la rappresentazione interna.

Un membro `protected` richiede un ragionamento diverso. Nello stesso package è accessibile secondo le regole del package; fuori package, la sottoclasse può accedervi nel contesto consentito dalle regole di ereditarietà. Per questo il costruttore protetto di `Veicolo` può essere chiamato da `Auto` con `super(nome)`, ma non è un costruttore a disposizione di qualunque chiamante esterno. La scelta di `protected` va motivata da un contratto per gli estensori della classe: una volta esposto un membro alle sottoclassi, cambiare quel contratto diventa più costoso.

La parola `super` ha due usi che vedremo spesso. `super(nome)` invoca un costruttore della classe base durante la costruzione della sottoclasse. `super.descrizione()` può invocare un'implementazione ereditata di un metodo quando la sottoclasse la vuole ampliare, purché quella implementazione esista e sia accessibile. Un metodo astratto non possiede un corpo da richiamare con `super`. In entrambi i casi `super` non crea un secondo oggetto base: la parte base e la parte specifica appartengono alla stessa istanza di `Auto`.

Nel programma `DemoSuper` vediamo le due forme nello stesso oggetto. Il costruttore della sottoclasse affida il nome a `super(nome)` e conserva l'alimentazione nel proprio campo. La ridefinizione di `descrizione()` parte dal testo della base con `super.descrizione()` e vi aggiunge il dettaglio dell'auto. È diverso dal copiare il corpo della base: se la base cambierà la forma della descrizione, la sottoclasse continuerà a riusarne il risultato. Questa dipendenza è utile soltanto se la base promette un risultato adatto a essere ampliato; altrimenti una composizione separata potrebbe essere più chiara.

```
class MezzoBase {
    private final String nome;

    MezzoBase(String nome) {
        this.nome = nome;
    }

    public String descrizione() {
        return "veicolo " + nome;
    }
}

class AutoDescritta extends MezzoBase {
    private final String alimentazione;

    AutoDescritta(String nome, String alimentazione) {

        super(nome);
        this.alimentazione = alimentazione;
    }

    @Override
    public String descrizione() {
        return super.descrizione() + " con motore " + alimentazione;
    }
}

public class DemoSuper {
    public static void main(String[] args) {
        MezzoBase mezzo = new AutoDescritta("A", "elettrico");
        System.out.println(mezzo.descrizione());
    }
}
```

L'output è `veicolo A con motore elettrico`. `mezzo` è dichiarato come `MezzoBase`, quindi il compilatore permette la chiamata a `descrizione()` perché è presente nella base. L'oggetto è un'`AutoDescritta`, quindi il corpo scelto durante l'esecuzione è quello ridefinito; **dentro** quel corpo la chiamata qualificata con `super` sceglie il corpo della base. Il testo finale è composto da entrambi. Se togliamo `super.` e scriviamo soltanto `descrizione()` dentro la ridefinizione, chiamiamo di nuovo il medesimo metodo dell'auto e otteniamo una ricorsione senza termine utile, non un accesso implicito alla base.

> **Per capire – Costruttore e metodo.** `super(nome)` non usa il punto perché invoca un costruttore in una posizione governata dalle regole di costruzione. `super.descrizione()` usa il punto perché invoca un metodo della base sull'istanza corrente. La grafia simile non deve farci chiamare un costruttore «metodo ereditato»: i costruttori non si ereditano come metodi ordinari.

Costruire la sottoclasse senza chiamarla troppo presto

Seguiamo `new Auto("A", motore)` dall'esterno. Si crea una nuova istanza di `Auto` e il suo costruttore deve assicurare l'inizializzazione della parte ereditata. La chiamata `super(nome)` demanda alla classe base la scelta di come custodire il nome. Solo dopo il completamento della costruzione della base il costruttore della sottoclasse può terminare la propria parte, assegnando il motore. Non sono due oggetti accostati: un riferimento all'`Auto` raggiunge un'unica istanza che possiede anche lo stato definito dalla base. La composizione con `Motore`, invece, coinvolge un altro oggetto raggiunto tramite un campo.

Se `Veicolo` chiamasse `descrizione()` dal proprio costruttore, il dispatch dinamico potrebbe eseguire la versione ridefinita in `Auto` prima che il campo `motore` sia pronto. Il risultato può essere un testo incompleto o un errore. La scelta robusta è inizializzare nella base soltanto ciò che la base conosce, senza delegare a un comportamento ridefinibile mentre l'oggetto è ancora in costruzione. Questo esempio spiega perché l'ordine dei costruttori è una regola progettuale e non soltanto una curiosità da indovinare nelle stampe.

Non si eredita il **costruttore** della base come se fosse un normale metodo pubblico. La sottoclasse dichiara i propri costruttori e può invocare un costruttore accessibile della base con `super(...)`; se omette la chiamata esplicita, valgono le regole per l'invocazione implicita del costruttore senza argomenti. Se la base non ne possiede uno accessibile, il codice della sottoclasse deve scegliere esplicitamente un'altra firma. In altre parole, aggiungere un costruttore alla base può obbligare le sottoclassi a cambiare: anche questa è una dipendenza da valutare quando disegniamo una gerarchia.

> **Nota bene – Visibilità della classe e visibilità del membro.** Anche un metodo dichiarato `public` non diventa utilizzabile da qualunque codice se la classe che lo dichiara, il suo package o il suo modulo non sono accessibili dal chiamante. Leggere un solo modificatore senza il contesto può dare una conclusione errata.

`protected` fra package e sottoclassi: la prova completa

L'esperimento con `Produttore` e `Consumatore` ci permette di osservare un caso spesso spiegato troppo in fretta. Un membro `protected` è accessibile alle classi dello stesso package anche se non sono sottoclassi. Una sottoclasse in un altro package ha accesso secondo le regole della sottoclasse, ma non può usare un riferimento arbitrario alla base come scorciatoia per leggere qualsiasi istanza. I quattro sorgenti dell'esperimento distinguono le due situazioni. Li teniamo in file diversi perché il **nome del package** è parte della prova, non un'etichetta decorativa.

Nel primo file, `Produttore` dichiara un campo e un metodo `protected`. Nel medesimo package, `Consumatore` non estende `Produttore`, ma può invocare `produttore.leggiDato()` su un riferimento ricevuto. La ragione è il package condiviso. In `accesso.esterno`, `Derivato` estende `Produttore` e può chiamare `this.leggiDato()` per leggere il proprio dato ereditato. Il programma principale stampa due volte `dato protetto`: una volta attraverso il privilegio del package e una volta attraverso il rapporto di sottoclasse.

```
package accesso.base;

public class Produttore {
    protected String dato = "dato protetto";

    protected String leggiDato() {
        return dato;
    }
}
```

```
package accesso.base;

public class Consumatore {
    public String leggi(Produttore produttore) {
        return produttore.leggiDato();
    }
}
```

```
package accesso.esterno;

import accesso.base.Produttore;

public class Derivato extends Produttore {
    public String leggiIlProprioDato() {
        return this.leggiDato();
    }
}
```

La variante negativa consiste nell'aggiungere a `Derivato` un metodo `String leggiAltro(Produttore altro) { return altro.leggiDato(); }`. Poiché `Derivato` è fuori dal package `accesso.base`, quell'accesso attraverso un riferimento dichiarato `Produttore` viene rifiutato dal compilatore. Non basta dire «il metodo è visibile alla sottoclasse»: occorre guardare **attraverso quale espressione** la sottoclasse tenta di usarlo. Se fosse un metodo `public`, l'accesso dipenderebbe ancora dalla visibilità della classe e del package o modulo, ma non da questa regola specifica di `protected`.

Per riprodurre la prova, compila insieme i quattro file con `javac --release 25 -d build accesso/base/Produttore.java accesso/base/Consumatore.java accesso/esterno/Derivato.java accesso/DemoAccesso.java` dalla cartella che li contiene ed esegui `java -cp build accesso.DemoAccesso`. Il comando nomina il **nome completo** della classe di avvio, perché questa volta il package è dichiarato. L'esperimento collega i capitoli 6, 8 e 9: package, accesso e sottoclasse agiscono insieme. Se tutto il codice fosse nello stesso package, non potremmo osservare il limite che vogliamo insegnare.

> **Buona pratica – Esponi un'operazione, non un campo da sorvegliare.** `Produttore` rende `dato` protetto soltanto per isolare la regola in una prova. In una classe destinata a evolvere, un campo `private` e metodi accessibili con un contratto stabile riducono il legame fra base e sottoclasse. `protected` è uno strumento per un'estensione progettata consapevolmente; usarlo su tutti i campi non sostituisce un'interfaccia per gli estensori.

Quando ci serve un'operazione propria di `Auto`, il solo tipo dichiarato `Veicolo` non basta per chiamarla. Il nostro programma usa un pattern `instanceof`:

```
if (mezzo instanceof Auto auto) {
    System.out.println(auto.alimentazione());
}
```

Il controllo accerta che l'oggetto sia un'`Auto` e, nello stesso ramo, introduce la variabile `auto` del tipo più specifico. L'output è `elettrico`. Un cast diretto senza controllo potrebbe fallire durante l'esecuzione se il veicolo fosse una `Bicicletta`. Neppure il pattern deve diventare il modo abituale di aggirare una classe base mal progettata: se ogni chiamante deve riconoscere la sottoclasse prima di operare, chiediti se il comportamento comune appartenga davvero al contratto della base.

Assegnare, convertire e riconoscere un riferimento

Un riferimento di tipo `Auto` può essere assegnato a una variabile `Veicolo` perché ogni `Auto` è un `Veicolo`: è una conversione verso un tipo più generale. Il percorso inverso non è garantito. Se una variabile `Veicolo` indica in realtà una `Bicicletta`, scrivere `(Auto) mezzo` compila in un contesto in cui il cast è ammesso, ma fallisce quando il programma tenta la conversione durante l'esecuzione. Il pattern `instanceof Auto auto` unisce il controllo e la dichiarazione della variabile specifica, evitando di ripetere il cast.

Facciamo una prova mentale con due righe: `Veicolo primo = new Auto(...);` `Veicolo secondo = new Bicicletta(...);`. Il tipo dichiarato delle variabili è uguale. `primo.descrizione()` e `secondo.descrizione()` sono entrambi ammessi dal compilatore, perché `Veicolo` promette quel metodo. Il corpo scelto in esecuzione è diverso perché i due oggetti effettivi sono diversi. `primo.alimentazione()` non è ammesso, benché l'oggetto sia un'`Auto`, perché quella capacità non compare nel tipo dichiarato. Dopo un controllo `instanceof`, possiamo usare il nome introdotto dal pattern per chiedere l'alimentazione.

Nel programma `DemoTipi` la stessa variabile `mezzo` indica prima un'auto e poi una bici. La sua dichiarazione non cambia: rimane `MezzoTipizzato`. Cambia il valore di riferimento che vi assegniamo e, quindi, la classe effettiva dell'oggetto raggiunto. Usiamo nomi leggermente diversi da quelli del programma sealed per isolare questa prova: qui vogliamo capire assegnazioni e cast, non l'esaustività di uno `switch`.

```
class MezzoTipizzato {
    public String descrizione() {
        return "mezzo";
    }
}

class AutoTipizzata extends MezzoTipizzato {
    @Override
    public String descrizione() {
        return "auto";
    }

    public int porte() {
        return 4;
    }
}

class BiciTipizzata extends MezzoTipizzato {
    @Override
    public String descrizione() {
        return "bici";
    }
}

public class DemoTipi {
    public static void main(String[] args) {
        MezzoTipizzato mezzo = new AutoTipizzata();
        System.out.println(mezzo.descrizione());
        if (mezzo instanceof AutoTipizzata auto) {
            System.out.println("porte: " + auto.porte());
        }

        mezzo = new BiciTipizzata();
        System.out.println(mezzo.descrizione());
        System.out.println(mezzo instanceof AutoTipizzata);

        MezzoTipizzato assente = null;
        System.out.println(assente instanceof AutoTipizzata);
        try {
            AutoTipizzata sbagliata = (AutoTipizzata) mezzo;
            System.out.println(sbagliata.porte());
        } catch (ClassCastException errore) {
            System.out.println("cast non valido");
        }
    }
}
```

L'output è `auto`, `porte: 4`, `bici`, `false`, `false`, `cast non valido`. La prima chiamata a `descrizione()` esegue la versione dell'auto; dopo la nuova assegnazione esegue quella della bici. `instanceof AutoTipizzata auto` introduce `auto` solo nel ramo in cui la verifica è vera. Dopo che `mezzo` indica una bici, il controllo restituisce falso. Restituisce falso anche per `null`: il valore nullo non è un'istanza della sottoclasse. Infine il cast diretto compila perché il tipo dichiarato potrebbe contenere un'auto, ma fallisce durante l'esecuzione dato che in quel momento contiene una bici.

Il `catch` finale è presente per rendere osservabile il fallimento senza interrompere la dimostrazione; **non** è una strategia consigliata per chiamare normalmente

`porte()`. Se sappiamo che il valore può indicare tipi diversi, il pattern `instanceof` è più chiaro quando l'operazione specifica è davvero necessaria. Se un algoritmo deve chiedere `porte()` a ogni mezzo, possiamo valutare un metodo nel contratto comune soltanto se la domanda ha senso per tutti. Una bici non ha quattro porte nascoste: aggiungere `porte()` alla base e restituire zero per ogni bici potrebbe essere un segnale di modellazione forzata.

L'ambito del nome introdotto dal pattern

L'operatore `instanceof` tradizionale restituiva un booleano; il programmatore doveva poi scrivere un cast per ottenere un nome del tipo più specifico. Con `instanceof AutoTipizzata auto` la verifica introduce anche la variabile `auto`, ma soltanto nei percorsi in cui il compilatore sa che il controllo è riuscito. Nel ramo vero di `if (mezzo instanceof AutoTipizzata auto)`, il nome è disponibile. Nel ramo `else` dello stesso `if` non possiamo usarlo, perché quel ramo significa proprio che `mezzo` non è stato riconosciuto come auto.

La negazione ribalta i percorsi. In `if (!(mezzo instanceof AutoTipizzata auto)) { ... } else { System.out.println(auto.porte()); }`, il nome `auto` è disponibile nell'`else`, dove la verifica originaria deve essere vera. Non è disponibile nel primo ramo, dove sappiamo soltanto che il valore non è un'auto. La regola è legata al **flusso del controllo**, non alla vicinanza grafica delle parentesi. Questo spiega anche perché, con espressioni booleane complicate, l'ambito di una variabile di pattern può richiedere più attenzione di un cast esplicito ben circoscritto. Se il lettore deve percorrere mentalmente troppi `&&`, `||` e `!` per sapere se il nome esiste, dividere la condizione in passaggi più chiari migliora il codice.

> **Nota bene – `null` non entra nel ramo vero.** Se il riferimento vale `null`, `mezzo instanceof AutoTipizzata auto` è falso e non viene introdotto alcun valore `auto` nel ramo vero. Il controllo evita quindi anche un cast su `null`, ma non decide da solo se `null` sia un input lecito. Quel significato appartiene al contratto del metodo che riceve `mezzo`.

> **Concetto chiave – Due controlli in due momenti.** In compilazione Java verifica che una chiamata sia lecita per il tipo disponibile nel sorgente. In esecuzione, per i metodi di istanza ridefinibili, sceglie l'implementazione legata all'oggetto effettivo. Il cast controllato appartiene ancora a un'altra domanda: l'oggetto effettivo è compatibile con il tipo più specifico richiesto?

La compatibilità si può leggere come un cammino nella gerarchia. Un oggetto `Auto` può essere trattato attraverso un riferimento `Auto`, attraverso un riferimento `Veicolo` e, più in generale, attraverso un riferimento `Object`, perché ogni classe partecipa alla catena che arriva a `Object`. Ogni passaggio verso un tipo più generale lascia l'oggetto invariato ma riduce ciò che il compilatore può promettere sulle chiamate scritte attraverso quel riferimento. `Object` offre metodi comuni come `toString()`, ma non conosce `descrizione()` di `Veicolo`: se dichiariamo `Object elemento = new Auto(...)`, la chiamata `elemento.descrizione()` non compila. Possiamo recuperare una capacità più specifica soltanto verificando o conoscendo con certezza il tipo effettivo, non immaginando che il compilatore guardi il `new` lontano dalla chiamata in ogni situazione.

Questo spiega anche perché una conversione verso la classe base è ordinaria e non richiede un cast scritto, mentre una conversione verso una sottoclasse può richiedere un controllo durante l'esecuzione. Se usiamo un riferimento `Auto` come `Veicolo`, sappiamo già dalla dichiarazione che l'auto soddisfa il contratto del veicolo. Nel verso opposto, un riferimento dichiarato `Veicolo` potrebbe indicare una bicicletta: scrivere il cast a `Auto` non la trasforma in un'automobile. Il compilatore ammette alcuni cast che **potrebbero** riuscire; se durante l'esecuzione l'oggetto non è compatibile, si ottiene `ClassCastException`. Seguiamo dunque due cose separate: il tipo con cui il codice usa il riferimento e l'oggetto raggiunto da quel riferimento. Il cast può cambiare le operazioni che il compilatore permette attraverso il riferimento, ma non cambia l'identità o la classe dell'oggetto.

> **Approfondimento – Metodi ponte.** Quando incontreremo i generics, vedremo casi in cui il compilatore produce un piccolo metodo aggiuntivo, detto *bridge method*, per conservare la ridefinizione dopo la cancellazione di alcune informazioni di tipo nel bytecode. Non è un metodo che l'autore deve scrivere a mano nel normale sorgente di `Auto`. Per ora basta la regola osservabile: una chiamata lecita attraverso il tipo base deve continuare a raggiungere l'implementazione corretta della sottoclasse. Il capitolo 15 mostrerà perché l'implementazione di quella promessa richiede talvolta un ponte.

La tabella segue lo stesso riferimento attraverso quattro istruzioni. In ogni riga il valore di `mezzo` indica un oggetto `Auto`, ma il tipo disponibile per scrivere la chiamata cambia secondo l'espressione usata.

Espressione	Che cosa sa il compilatore	Esito
`Veicolo mezzo = new Auto(...);`	`Auto` è compatibile con `Veicolo`	assegnazione lecita
`mezzo.descrizione()`	`Veicolo` dichiara il metodo	esegue il corpo di `Auto`
`mezzo.alimentazione()`	`Veicolo` non dichiara quel metodo	errore di compilazione
`if (mezzo instanceof Auto auto) auto.alimentazione();`	nel ramo vero `auto` ha tipo `Auto`	chiamata lecita

Il cast `(Auto) mezzo` sarebbe un'affermazione del programmatore sottoposta a controllo durante l'esecuzione. Non modifica l'oggetto e non rende vere ipotesi sbagliate: se `mezzo` indica una `Bicicletta`, il cast fallisce. Per questo il pattern `instanceof` è utile quando il caso specifico è davvero necessario. Se la stessa verifica compare in molti punti del programma, la ripetizione è un indizio che il contratto comune o la distribuzione delle responsabilità meritano una revisione.

Sostituire davvero, senza rompere la promessa

La prova più utile per una gerarchia non è «compila?» ma «un chiamante che conosce soltanto `Veicolo` può continuare a ragionare allo stesso modo?». Se la base promette che `descrizione()` restituisce sempre un testo non vuoto, una sottoclasse che restituisce `null` rompe la promessa. Se la base permette di fermare qualunque veicolo, una sottoclasse che lancia un'eccezione per l'operazione normale di fermata rende la sostituzione discutibile. Questi sono problemi del **contratto comportamentale**, che la sola parola `extends` non controlla.

Un'altra trappola è usare l'ereditarietà soltanto per recuperare due righe già scritte. Se `Auto` e `Bicicletta` condividono una funzione di formattazione, possiamo spostare la funzione in una classe di supporto o comporre un oggetto che se ne occupi. Creare una superclasse con un nome artificiale per evitare una piccola duplicazione può introdurre un legame permanente più costoso della duplicazione stessa. Il riuso è un effetto possibile di una relazione ben progettata, non la prova che la relazione sia giusta.

Rileggere la prima classe `Veicolo` come revisori del codice

Riprendiamo la classe `Veicolo` per esaminare la qualità del codice. Immaginiamo una prima forma che espone pubblicamente `tipo`, `velocita` e `direzione`, così la sottoclasse `Macchina` può assegnarli direttamente. È comodo per far muovere subito l'esempio, ma distribuisce la responsabilità delle regole su ogni sottoclasse e su ogni chiamante. Che cosa impedisce di assegnare una velocità negativa? Che cosa significa una direzione intera diversa dalle tre costanti previste? Se una nuova sottoclasse dimentica di inizializzare `tipo`, la base può trovarsi con uno stato che i suoi metodi non sanno descrivere.

Rendere privati i campi e aggiungere getter e setter risolve parte del problema. Questo è un passo, ma un `setVelocita(int)` che accetta qualunque intero riapre la stessa falla attraverso una porta pubblica. Nel nostro `DemoMovimento`, `muovi(int nuovaVelocita)` controlla il valore prima di modificare il campo e `ferma()` usa la stessa operazione passando zero. Il nome del metodo dichiara una richiesta del dominio; la validazione resta in un punto. Se il modello dovesse distinguere «fermo», «in movimento» e «guasto», potremmo introdurre un tipo adatto invece di moltiplicare interi speciali sparsi nel programma.

> **Approfondimento – Effetto collaterale.** Quando un metodo modifica uno stato osservabile oltre al valore che restituisce, quella modifica è un *effetto collaterale*. `muovi(30)` cambia la velocità del veicolo; una successiva chiamata a `descrizione()` può dunque produrre un testo diverso. Un effetto collaterale non è necessariamente un difetto: serve per modellare un cambiamento reale. Diventa difficile da controllare quando una chiamata dal nome apparentemente innocuo modifica un oggetto condiviso, oppure quando base e sottoclasse aggiornano lo stesso stato attraverso vie diverse. Nominare l'effetto nel contratto aiuta a prevedere il comportamento dopo una chiamata e a scrivere una prova che lo osservi.

Anche l'ordine nel file aiuta chi dovrà mantenere la classe. Una convenzione utile mette le costanti all'inizio e i getter e setter alla fine: è una disposizione possibile, perché rende facili da trovare le dichiarazioni condivise. Non è però una regola del

linguaggio, né il modo migliore per ogni classe. Se un metodo di lettura è l'unica operazione pubblica legata a una regola del dominio, lasciarlo vicino a quella regola può essere più chiaro che spedirlo in fondo a una lunga serie di accessori. Nel nostro `Veicolo`, raggruppiamo ciò che descrive il contratto comune e teniamo vicini i metodi che modificano lo stesso stato; poi manteniamo coerente quella scelta nel progetto.

Le tre direzioni `DRITTO`, `SINISTRA` e `DESTRA` dell'esempio sono costanti `static final int`. Scrivere `Veicolo.SINISTRA` è più leggibile di un `-1` isolato, ma una variabile `int direzione` può ancora ricevere `27`. Un enum `Direzione` renderebbe il dominio più esplicito, come vedremo nel capitolo 10. La scelta va graduata: una breve dimostrazione sul sovraccarico può restare con numeri semplici; un tipo destinato a essere mantenuto e usato da molti componenti merita un contratto più forte. Introduciamo un nuovo costrutto quando risolve un difetto concreto del programma e possiamo verificarne l'effetto.

> **Buona pratica – Una regola posseduta da una sola classe.** La base deve proteggere i propri dati; la sottoclasse deve aggiungere il comportamento che la specializza, senza duplicare controlli interni della base. Se ogni `Macchina` deve ricordare di mettere `velocita = 0` nel proprio costruttore, una futura sottoclasse può dimenticarlo. Un costruttore della base che stabilisce lo stato comune rende più facile mantenere l'invariante lungo tutta la gerarchia.

Quando una sottoclasse aggiunge e quando cambia

Una sottoclasse può aggiungere un metodo che la base non possiede. `Auto` potrebbe offrire `apriBagagliaio()`: chi dispone di un riferimento dichiarato `Auto` può chiamarlo, mentre chi dispone soltanto di `Veicolo` no. Questo è coerente se il bagagliaio non fa parte del contratto di ogni veicolo. Se invece la sottoclasse ridefinisce `descrizione()`, sta cambiando il corpo di un'operazione già nota alla base. La chiamata tramite `Veicolo` continua a compilare, ma durante l'esecuzione raggiunge il corpo della sottoclasse. Sono due modi diversi di specializzare una classe e producono effetti diversi su chi la usa.

Un metodo dichiarato `final` nella base non può essere ridefinito dalla sottoclasse. Può essere una scelta sensata se la base deve mantenere un passaggio invariabile, per esempio un controllo che precede un'operazione più specifica. Dichiarare `final` ogni metodo per paura delle sottoclassi, però, rende inutile promettere un punto di estensione. Viceversa, lasciare ridefinibile un metodo chiamato dal costruttore espone la base al rischio di eseguire codice di una sottoclasse ancora incompleta. La scelta di estensione va documentata nel contratto: quali metodi possono essere personalizzati, quali invarianti devono mantenere e in quale momento possono essere chiamati.

I campi non seguono il dispatch dinamico dei metodi ridefiniti. Se una sottoclasse dichiara un campo con lo stesso nome di uno della base, il nuovo campo **nasconde** quello ereditato nel contesto appropriato; non diventa una versione alternativa dello stesso dato scelta in base all'oggetto effettivo. Questa situazione rende il codice difficile da leggere: due espressioni apparentemente simili possono riferirsi a campi diversi in base al tipo con cui sono scritte. Per rappresentare uno stato comune, è più chiaro lasciare alla base il controllo dei propri campi e offrire metodi

con un significato preciso. `super.nome()` può richiamare un metodo accessibile della base; duplicare un campo `nome` nella sottoclasse creerebbe invece due fonti di verità.

> **Nota bene – `@Override` protegge un'intenzione.** Scrivilo quando intendi ridefinire un metodo: se sbagli la firma, il compilatore ti avvisa. Senza l'annotazione, un metodo con un parametro diverso potrebbe diventare un sovraccarico, e il programma continuerebbe a compilare eseguendo il corpo della base nelle chiamate che volevi cambiare. L'annotazione non rende corretta la nuova implementazione sul piano del contratto, ma intercetta un errore di forma molto comune.

Un caso in cui la composizione rende la modifica locale

Immaginiamo che la descrizione di un veicolo debba includere una politica di tariffazione. Un'auto e una bicicletta possono essere entrambe veicoli, ma la tariffa dipende anche da luogo, giorno e promozioni. Se mettiamo tutte queste regole in `Veicolo` e poi le ridefiniamo in ogni sottoclasse, ogni nuova politica rischia di obbligarci a modificare l'intera gerarchia. Possiamo invece far collaborare il veicolo con un oggetto `Tariffa` che riceve i dati necessari e calcola il prezzo. Il veicolo mantiene la propria identità e il proprio comportamento di veicolo; la politica ha una responsabilità separata che può essere verificata da sola.

Questo non significa introdurre un oggetto di supporto per ogni riga di codice. Significa riconoscere due assi di variazione diversi: il **tipo di veicolo** e la **regola di prezzo**. L'ereditarietà modella bene una relazione «è un» quando il contratto comune è stabile. La composizione aiuta quando un comportamento può cambiare indipendentemente da quella classificazione. Il criterio si applica anche all'esempio del motore: `Auto` è un `Veicolo` e **ha un** `Motore`; una futura auto con una diversa alimentazione non richiede una nuova superclasse solo per descrivere il componente.

Chiudere una gerarchia quando il dominio lo richiede

La parola `sealed` in `Veicolo` limita le sottoclassi dirette ammesse a quelle indicate da `permits`: qui `Auto` e `Bicicletta`. Entrambe sono `final`, quindi non possono essere ulteriormente estese. Una sottoclasse ammessa potrebbe anche essere `sealed`, continuando a fissare un elenco, oppure `non-sealed`, riaprendo l'estensione. Questa scelta va fatta sul modello: se sappiamo che il dominio ha un insieme chiuso di casi, il compilatore può aiutarci a controllare che siano tutti trattati. Se invece una libreria deve consentire estensioni sconosciute, chiuderla può essere un ostacolo.

Disegnare davvero una gerarchia chiusa

Un tipo `sealed` non è una classe `final` con una grafia più lunga. `final` impedisce ogni ulteriore sottoclasse; `sealed` ammette precisamente le sottoclassi dirette previste dalla dichiarazione. Se `Veicolo` permette `Auto` e `Bicicletta`, una nuova classe `Monopattino extends Veicolo` non può essere aggiunta in un altro file senza modificare la gerarchia autorizzata. La regola è intenzionale: quando il dominio ha un insieme noto di alternative, l'aggiunta di un'alternativa deve diventare un cambiamento visibile per chi mantiene gli algoritmi che le esaminano.

Ogni sottoclasse diretta ammessa deve dichiarare come prosegue la propria storia: `final` la chiude, `sealed` elenca a sua volta un insieme di discendenti ammessi, `non-sealed` riapre l'estensione. Se dichiariamo `non-sealed Auto`, un'altra classe può estendere `Auto` secondo le normali regole di accesso; il tipo base conosce ancora la variante diretta `Auto`, ma non può più elencare tutte le sue possibili sottoclassi concrete. La scelta non riguarda soltanto il compilatore: racconta ai lettori del modello quali confini sono stabiliti e quali punti sono offerti per nuove specializzazioni.

Nel caso dei nostri tre tipi nello stesso file, `permits Auto, Bicicletta` rende il rapporto esplicito. In alcune condizioni il compilatore può inferire le sottoclassi permesse dichiarate nello stesso file, ma per il lettore che sta imparando la gerarchia l'elenco scritto ha valore didattico. In un progetto modulare le sottoclassi dirette ammesse appartengono allo stesso modulo nominato della base; fuori da un modulo nominato devono appartenere allo stesso package. Non si può quindi usare `permits` come un registro aperto di plugin distribuiti in moduli arbitrari. Una API progettata per estensioni esterne ha un'esigenza diversa da una gerarchia chiusa.

> **Approfondimento – Chiusura e manutenzione.** Limitare la gerarchia non impedisce errori nei corpi dei metodi delle sottoclassi. Aiuta però a sapere dove cercare le varianti e a individuare alcuni algoritmi da aggiornare quando se ne aggiunge una. Se un nuovo tipo `Monopattino` entra nel dominio, il `switch` esaustivo che oggi tratta auto e bicicletta deve essere riesaminato. Questo lavoro imposto dal compilatore è utile quando la classificazione è davvero chiusa; è un costo inutile quando la piattaforma deve ricevere implementazioni esterne senza cambiare il modulo della base.

Il `switch` finale usa proprio quell'insieme noto:

```
String categoria = switch (mezzo) {
    case Auto auto -> "auto";
    case Bicicletta bicicletta -> "bicicletta";
};
```

Non compare `default`: per una gerarchia sealed di cui il compilatore conosce tutti i sottotipi diretti e concreti, i due casi coprono i valori non null ammessi. Se `mezzo` fosse `null`, questo `switch` lancerebbe comunque un'eccezione durante l'esecuzione; quando `null` è un valore previsto dal contratto, va trattato esplicitamente come abbiamo fatto nel capitolo 5. Togliendo un caso, il programma non compila perché l'espressione non è più esaustiva. Il riepilogo ufficiale distingue le classi sealed, stabili da Java 17, dal pattern matching per `switch`, stabile da Java 21; entrambi sono ordinari in Java 25.

Il `switch` risponde a una domanda diversa dal metodo virtuale `descrizione()`. Quando chiediamo a ogni veicolo di descrivere sé stesso, il comportamento appartiene alla variante e il dispatch dinamico sceglie il corpo. Quando un algoritmo esterno deve produrre una categoria amministrativa per ogni variante nota, uno `switch` esaustivo può rendere visibile che tutti i casi sono stati considerati. Non trasformiamo ogni metodo polimorfico in uno `switch`, né nascondiamo ogni decisione esterna dentro le sottoclassi per principio. La posizione del comportamento dipende da chi possiede la regola e da quanto spesso cambiano i tipi e le operazioni.

> **Concetto chiave – Sostituibilità prima della sintassi.** `extends` è facile da scrivere; il lavoro vero è assicurare che ogni sottoclasse rispetti ciò che un chiamante può ragionevolmente attendersi dalla classe base. Una gerarchia chiusa rende esplicito chi può estenderla, ma non rende automaticamente buono il suo contratto.

La parola `final` assume significati collegati ma distinti secondo la dichiarazione: su una classe impedisce sottoclassi; su un metodo impedisce la ridefinizione; su un campo impedisce una nuova assegnazione dopo l'inizializzazione consentita. Non garantisce da sola che gli oggetti raggiungibili da quel campo siano immutabili. Tenere separati questi tre casi aiuta a leggere `final class Auto`, `final String nome` e `final String descrizione()` senza attribuire a tutti la stessa promessa.

Gerarchie chiuse, dati immutabili e record

La classe `Veicolo` è `sealed` perché in questo esempio vogliamo conoscere le varianti dirette. Una classe `final` chiude invece l'estensione completamente. Un record è implicitamente finale e può essere un buon modo di rappresentare un valore trasparente come `Motore(String alimentazione)`. Nessuna di queste parole rende **immutabile in profondità** tutto ciò che l'oggetto può raggiungere. Se un record contiene un array, il riferimento al componente non può essere cambiato attraverso un normale setter, ma gli elementi dell'array possono essere modificati da chi possiede quel riferimento. Le copie difensive del capitolo 8 restano pertinenti.

Due oggetti `PersonaFisica` con gli stessi dati mostrano che i record confrontano i componenti. È un buon esempio se i componenti sono valori il cui `equals` rappresenta il confronto desiderato. Con un componente array, invece, l'uguaglianza generata usa l'uguaglianza di quell'array, che normalmente è basata sull'identità, non su una comparazione elemento per elemento. La parola «record» non risolve da sola ogni decisione sul significato dell'uguaglianza.

La persona dell'esempio storico e il significato di «uguale»

Nel record `PersonaFisica(nome, cognome, eta, altezza, peso)`, due istanze create separatamente con gli stessi componenti risultano uguali secondo `equals`. Il risultato ci permette di vedere che il compilatore genera un confronto coerente con la dichiarazione dei componenti. Non dimostra però che una persona, nel dominio di un'anagrafica reale, sia identificata da quei cinque dati. Due persone possono avere nomi e misure uguali; una stessa persona può cambiare età o peso. Il record rappresenta bene una **fotografia di dati** quando quel confronto per componenti è la domanda che vogliamo fare. Per un'identità che persiste mentre i dati cambiano, serve una decisione esplicita, per esempio un identificatore stabile custodito da una classe con un contratto appropriato.

L'immutabilità va letta allo stesso modo, a partire dall'osservazione. Un record non offre setter per i componenti, e i campi corrispondenti non possono essere riassegnati dopo la costruzione. Se un componente è una `String`, il valore del testo è immutabile; se un componente è un array, il riferimento non viene riassegnato ma gli elementi possono cambiare. Questa è la differenza fra immutabilità **superficiale** e protezione dell'intero grafo di oggetti raggiungibili. Il `RegistroVoti` del capitolo 8 usava copie in ingresso e in uscita proprio per impedire modifiche indirette. Non promettiamo thread safety o sicurezza in generale soltanto perché una classe è `final` o un record: occorre esaminare tutti gli oggetti mutabili che espone e il modo in cui sono pubblicati fra thread.

Nel capitolo 7 abbiamo presentato `Edizione(String titolo, int anno)` come valore trasparente. Qui il record `Motore(String alimentazione)` è un componente dell'auto. Entrambi aiutano a ridurre codice ripetitivo quando la dichiarazione dei componenti coincide con il significato del valore. La classe `Veicolo`, al contrario, definisce un contratto di comportamento per varianti sostituibili. Record, `final` e `sealed` rispondono a domande diverse: come rappresento un valore, se permetto ulteriori sottoclassi e quali sottoclassi dirette autorizzo. È possibile usarli insieme in un progetto, ma non vanno trattati come tre versioni della stessa soluzione.

Anche l'ordine di inizializzazione diventa più interessante con una gerarchia. Prima di eseguire il corpo del costruttore di `Auto`, deve essere costruita la sua parte `Veicolo`; inizializzatori e costruttori partecipano a una sequenza definita. Per questo `Veicolo` non dovrebbe chiamare nel proprio costruttore un metodo che `Auto` può ridefinire: il corpo di `Auto` potrebbe non avere ancora assegnato `motore`, e la ridefinizione leggerebbe uno stato incompleto. Il piccolo programma `DemoOrdine` mostra il caso senza blocchi; la specifica di creazione delle istanze descrive la sequenza completa. Quando l'ordine conta, lo verifichiamo con un caso ridotto invece di affidarci all'impressione visiva delle righe.

Per verificare

Compila il programma completo con `javac --release 25 -Xlint:all -d build DemoVeicoli.java`, poi esegui `java -cp build DemoVeicoli` dalla cartella del file. Prima di avviarlo, prevedi le tre righe stampate. Poi sostituisci la costruzione dell'`Auto` con `new Bicicletta("B")`: quale ramo del `switch` verrà scelto? Che cosa dovrai fare del controllo `instanceof`? L'attività del capitolo 9 separa queste due osservazioni.

Per una seconda verifica, tieni fissa la dichiarazione `Veicolo mezzo` e cambia soltanto l'oggetto assegnato: prima `Auto`, poi `Bicicletta`. Prevedi separatamente tre cose: quali metodi il compilatore permette di nominare, quale corpo di `descrizione()` viene eseguito e quale caso del `switch` riconosce l'oggetto. Se cambi soltanto il tipo dichiarato del riferimento mantenendo la stessa istanza, la risposta alla prima domanda può cambiare, mentre l'oggetto non si trasforma. Questa separazione è il criterio con cui controllare anche gli esempi più lunghi.

Infine prova a spiegare senza codice perché `Auto` può avere un `Motore` senza estenderlo, perché `Ruota` può chiamare `super(raggio)` senza poter assegnare direttamente il campo privato della base, e perché un `catch` di `ClassCastException` non rende una conversione sbagliata una buona scelta progettuale. Se una risposta dipende dalla sola parola chiave, ritorna alla relazione fra contratti e oggetti: la sintassi serve a esprimere quel rapporto, non a sostituirlo.

10. Object, classi wrapper ed enumerazioni

Un antenato comune, tre domande diverse

Nel capitolo 7 abbiamo visto che due variabili possono riferirsi allo stesso oggetto. Nel capitolo 9 abbiamo introdotto una gerarchia di classi. Alla radice delle classi Java c'è `Object`: anche quando non scriviamo `extends Object`, una classe senza altra superclasse lo estende implicitamente. Questo dà a ogni istanza metodi come `equals`, `hashCode` e `toString`. Non significa però che questi metodi rispondano sempre alla domanda che interessa al nostro programma.

`==` fra riferimenti chiede se identificano **lo stesso oggetto**. `equals` chiede se due oggetti sono **uguali secondo il criterio della loro classe**. Il comportamento ereditato direttamente da `Object` considera uguali soltanto i riferimenti allo stesso oggetto; una classe può ridefinirlo. `toString` produce una rappresentazione testuale utile a leggere un oggetto. La sua implementazione di base non è una descrizione del suo contenuto: se vogliamo mostrarlo con un significato chiaro, la ridefiniamo.

Nel programma completo modelliamo una piccola `Etichetta` che vale per il testo che contiene. Ne costruiamo due con il medesimo testo, ma in due espressioni `new` distinte. Il confronto di identità è `false`; il confronto di valore è `true`. La figura 10.1 lega le tre domande alle tre espressioni, senza suggerire che un hash sia un'identità.



Figura 10.1 – Due etichette diverse possono avere lo stesso valore. Un codice hash uguale è richiesto per oggetti uguali; da solo non prova che gli oggetti siano uguali.

Dare un significato a equals e hashCode

L'esempio usa una classe immutabile: dopo la costruzione, il suo campo `testo` non cambia. Questo rende più facile rispettare il contratto di uguaglianza. Il metodo `equals` controlla prima l'identità, poi verifica che l'altro oggetto sia un'`Etichetta`; infine confronta i testi con `String.equals`.

```
@Override
public boolean equals(Object altro) {
    if (this == altro) return true;
    if (!(altro instanceof Etichetta etichetta)) return false;
    return testo.equals(etichetta.testo);
}

@Override
public int hashCode() {
    return testo.hashCode();
}
```

La firma riceve `Object` perché è quella del metodo che ridefiniamo. Il pattern `instanceof`, incontrato nel capitolo 9, evita un cast separato. Due `Etichetta` con lo stesso testo ottengono lo stesso hash perché lo calcoliamo dal medesimo dato usato da `equals`. Se ridefinissimo soltanto `equals`, una collezione basata su hash potrebbe cercare oggetti uguali in posizioni diverse e comportarsi in modo inatteso.

> **Concetto chiave – Il contratto di uguaglianza.** `equals` deve essere riflessivo, simmetrico, transitivo e coerente finché i dati confrontati non cambiano. Se `a.equals(b)` è vero, `a.hashCode()` e `b.hashCode()` devono essere uguali. L'inverso non è richiesto: due oggetti diversi possono produrre lo stesso numero hash. La specifica di `Object` espone il contratto completo.

Due punti con le stesse coordinate

Costruiamo `new Punto(10, 10)` due volte. Prima di ridefinire `equals`, i due punti sono istanze distinte e il metodo ereditato da `Object` restituisce `false`. Dopo una ridefinizione che confronta `x` e `y`, il risultato diventa `true`. A questo punto dobbiamo considerare `hashCode`: se i due punti sono uguali secondo `equals`, devono produrre lo stesso codice hash. La nostra `Etichetta` segue esattamente questa progressione, usando una sola componente per rendere visibile il contratto.

Il codice hash non è un numero identificativo univoco dell'oggetto. Restituisce un `int`, quindi molti valori possibili del mondo reale devono condividere qualche risultato, e due oggetti diversi possono avere una **collisione**. Una struttura basata su hash usa il numero per restringere la ricerca e poi deve verificare l'uguaglianza. Anche la rappresentazione predefinita di `toString`, che mostra il nome della classe seguito da `@` e da una forma esadecimale del codice hash, non dimostra che due oggetti siano identici o diversi. La sequenza non va interpretata come l'indirizzo di memoria dell'istanza.

La mutabilità rende più difficile questa promessa. Se inseriamo un oggetto in una collezione basata su hash e poi cambiamo un campo usato sia da `equals` sia da

`hashCode`, la ricerca potrebbe non trovarlo più dove lo aveva collocato. Per questo `Etichetta` conserva un testo immutabile. Non è l'unica soluzione possibile, ma riduce un rischio concreto che un esempio troppo piccolo potrebbe nascondere.

Per classi con più campi, `Objects.equals` e `Objects.hash` della libreria standard possono semplificare il codice. Non aggiungiamo automaticamente tutti i campi: scegliamo prima che cosa rende due istanze equivalenti nel dominio. Se l'uguaglianza deve basarsi su tutti i componenti di un semplice valore immutabile, un `record` come quello del capitolo 7 fornisce già `equals`, `hashCode` e `toString` coerenti con i componenti. Se un componente è un array, il confronto generato dal record usa comunque l'uguaglianza di quel componente: la progettazione del valore resta responsabilità nostra.

`toString` nell'esempio restituisce `Etichetta[Java]`, così una stampa o un messaggio diagnostico sono leggibili. Non dovrebbe rivelare dati riservati: il testo stampato da un oggetto può finire in log e segnalazioni d'errore.

`getClass()` restituisce un oggetto `Class` che descrive la classe effettiva dell'istanza durante l'esecuzione. Può servire per diagnostica o per API che esaminano tipi, ma la normale logica del programma dovrebbe preferire contratti, metodi e interfacce alla continua interrogazione della classe concreta. `clone()` esiste in `Object`, ma la sua combinazione con `Cloneable` e con oggetti che contengono altri oggetti richiede attenzione: non è un comando universale per ottenere una copia indipendente. Per un valore semplice è spesso più chiaro costruire un nuovo oggetto o usare un metodo di copia con un contratto esplicito.

Sull'oggetto `Class` ottenuto da un `Punto` possiamo chiamare `getSimpleName()`, `getName()` e `getPackageName()`. La differenza si vede con una classe nel package `esempi.geometria`: il nome semplice può essere `Punto`, mentre il nome completo comprende il package. Queste informazioni possono aiutare un sistema che registra eventi o carica componenti per nome, ma non dicono quali valori contenga una particolare istanza di `Punto`. La *reflection* permette di esaminare membri e, nei limiti consentiti, invocare operazioni; non riscrive a piacere il codice sorgente o l'interfaccia pubblica di una classe già caricata, come si potrebbe credere da una descrizione troppo generica. La sua potenza va accompagnata da regole di accesso, moduli e controlli sugli errori.

`Object` dichiara anche metodi legati alla coordinazione fra thread, come `wait` e `notify`. Non li useremo per imparare l'uguaglianza: appartengono al tema della concorrenza e richiedono di comprendere monitor, possesso del lock e condizioni di attesa. Il fatto che un metodo sia ereditato da ogni oggetto non significa che sia opportuno chiamarlo in qualsiasi capitolo.

Tre metodi, tre prove diverse

Consideriamo due punti con coordinate `(10, 10)`. Senza una ridefinizione, `primo.equals(secondo)` eredita il comportamento di `Object` e restituisce `false` se abbiamo usato due istruzioni `new`, benché le coordinate coincidano. Dopo una ridefinizione che considera soltanto `x` e `y`, il risultato diventa `true`. A quel punto dobbiamo ridefinire anche `hashCode` usando gli stessi dati. Non serve ottenere un numero hash «bello» o diverso per ogni punto: serve garantire che due punti considerati uguali producano lo stesso numero. Se questa regola manca, il difetto potrebbe rimanere invisibile finché non inseriamo i punti in una raccolta basata su hash.

La terza prova riguarda `toString`. Stampare `System.out.println(primo)` invoca una rappresentazione testuale dell'oggetto. Possiamo decidere di restituire `Punto[x=10, y=10]`, che aiuta la diagnosi; questa scelta non modifica né l'identità né l'uguaglianza. Due punti possono avere lo stesso `toString` e restare distinti, oppure essere uguali secondo il contratto e avere una rappresentazione destinata a cambiare per ragioni editoriali. Non scriviamo un programma che confronta oggetti attraverso la loro stringa di stampa: le tre domande devono rimanere separate.

`getClass()` risponde a una quarta domanda: qual è la classe effettiva dell'istanza che stiamo osservando? Se una variabile ha tipo `Object` ma indica un `Punto`, `getClass()` descrive `Punto`. Il risultato è un oggetto `Class`, non una stringa da confrontare a mano. `getSimpleName()` può produrre il solo nome semplice e `getName()` un nome più completo, utile nei messaggi di diagnostica; nessuno dei due ci dice se due punti abbiano coordinate uguali. Nel capitolo sull'ereditarietà abbiamo visto che lavorare attraverso un tipo generale è normale: ricorrere a `getClass()` per ogni decisione del dominio significherebbe spesso perdere il beneficio dei metodi comuni e della ridefinizione.

Un'ultima distinzione riguarda il codice hash. Il metodo `hashCode()` non promette di restituire l'indirizzo dell'oggetto e non promette un numero diverso per ogni istanza. Un `HashSet` può collocare due valori diversi nella stessa zona della propria struttura e poi distinguerli con `equals`. Il contratto obbliga soltanto gli oggetti uguali ad avere hash uguali durante il periodo in cui i dati rilevanti non cambiano. Per questo il nostro esempio evita di stampare numeri hash specifici come output atteso: il lettore deve verificare la **relazione** fra i valori, non imparare un numero prodotto da una particolare esecuzione.

> **Approfondimento – `Objects.equals` e il caso `null`.** Se due campi possono essere `null`, `Objects.equals(a, b)` restituisce `true` quando sono entrambi nulli, `false` quando uno solo lo è, e altrimenti chiama `a.equals(b)`. Riduce un controllo ripetitivo, ma non decide *quali* campi appartengano al significato del valore. `Objects.hash(...)` costruisce un hash dai valori forniti; anche qui la scelta dei valori deve restare coerente con `equals`. Il nome simile non autorizza a usare `Objects.equals` in `equals` e un campo diverso in `hashCode`.

Quando un primitivo serve come oggetto

I tipi primitivi, per esempio `int`, non sono istanze di `Object`. Le **classi wrapper** (`Integer`, `Double`, `Boolean` e le altre) rappresentano quei valori come oggetti. Nell'esempio, `Integer.valueOf(42)` produce un `Integer`; l'assegnazione a `int valore` estrae il primitivo attraverso l'**unboxing**. L'operazione inversa, dal primitivo al wrapper, si chiama **boxing**. Il compilatore può inserirle automaticamente quando il contesto lo richiede.

```
Integer numero = Integer.valueOf(42);
int valore = numero;
System.out.println(valore + 1);
```

La riga stampa 43. Se `numero` fosse `null`, l'unboxing non avrebbe un valore da estrarre e produrrebbe `NullPointerException`. Inoltre, `==` applicato a due

riferimenti wrapper confronta l'identità e può sembrare dare risultati diversi per valori diversi a causa delle regole di caching. Per confrontare il valore, usa `equals` oppure, quando i valori sono validi e la conversione è appropriata, confronta i primitivi. La documentazione di `java.lang` elenca i wrapper; le collezioni che vedremo più avanti ne sono un caso d'uso comune.

> **Nota bene – Confronti misti.** In `numero == 42`, uno dei due operandi è primitivo e Java può estrarre il valore dal wrapper; in `numero == unAltroInteger`, se entrambi sono riferimenti `Integer`, il confronto è d'identità. Scrivere tipi e conversioni esplicitamente mentre si impara evita di attribuire un significato unico a == in contesti differenti.

Perché esistono i wrapper

Una collezione come `List<Integer>` lavora con oggetti, mentre un array `int[]` conserva direttamente valori primitivi. I wrapper permettono di usare valori numerici nei contesti che richiedono riferimenti. La corrispondenza è `byte`/`Byte`, `short`/`Short`, `int`/`Integer`, `long`/`Long`, `float`/`Float`, `double`/`Double`, `char`/`Character` e `boolean`/`Boolean`. I nomi `Integer` e `Character` non si ottengono aggiungendo semplicemente una maiuscola all'inizio del primitivo: è bene riconoscerli quando compaiono nei messaggi del compilatore.

Il valore rappresentato da un wrapper non cambia dopo la sua creazione. Se scriviamo `Integer numero = 10; numero = numero + 1;`, la seconda istruzione non modifica internamente l'oggetto che rappresentava `10`: calcola un nuovo valore e assegna alla variabile un riferimento appropriato per `11`. Il compilatore può nascondere boxing e unboxing in una sintassi breve, ma le conversioni esistono e possono avere costi o fallire se un wrapper da estrarre vale `null`. Per la conversione esplicita da testo a numero useremo metodi come `Integer.parseInt("42")`, distinguendo l'errore di testo non numerico dalla semplice presenza di un oggetto wrapper.

La tabella dei wrapper rende evidente la corrispondenza, senza suggerire che `Boolean` e `Character` appartengano alla gerarchia numerica.

Valore primitivo	Oggetto wrapper	Un uso possibile
`byte`, `short`, `int`, `long`	`Byte`, `Short`, `Integer`, `Long`	valore intero in un'API che richiede oggetti
`float`, `double`	`Float`, `Double`	valore in virgola mobile in una raccolta
`char`	`Character`	unità UTF-16 trattata come oggetto
`boolean`	`Boolean`	esito logico in un contesto che ammette `null`

La tabella accorpa le coppie per funzione, ma ciascun primitivo ha il **proprio** wrapper. `Integer` non contiene un `int` da cambiare tramite setter. Per ottenere un `int` da un testo usiamo `Integer.parseInt`, che restituisce un primitivo;

`Integer.valueOf` restituisce invece un oggetto `Integer`. Entrambi possono rifiutare un testo non numerico con `NumberFormatException`. Se dobbiamo rappresentare l'assenza di un valore, `Integer` può contenere `null` come riferimento, ma questo impone una scelta esplicita prima dell'unboxing. Un valore mancante non dovrebbe essere confuso automaticamente con zero.

La figura 10.2 mostra la gerarchia dei wrapper: i wrapper dei numeri fanno capo a `Number`, mentre `Boolean` e `Character` sono separati. Il disegno non dice che tutti i valori numerici siano intercambiabili senza conversioni; mostra soltanto il rapporto fra le classi wrapper.

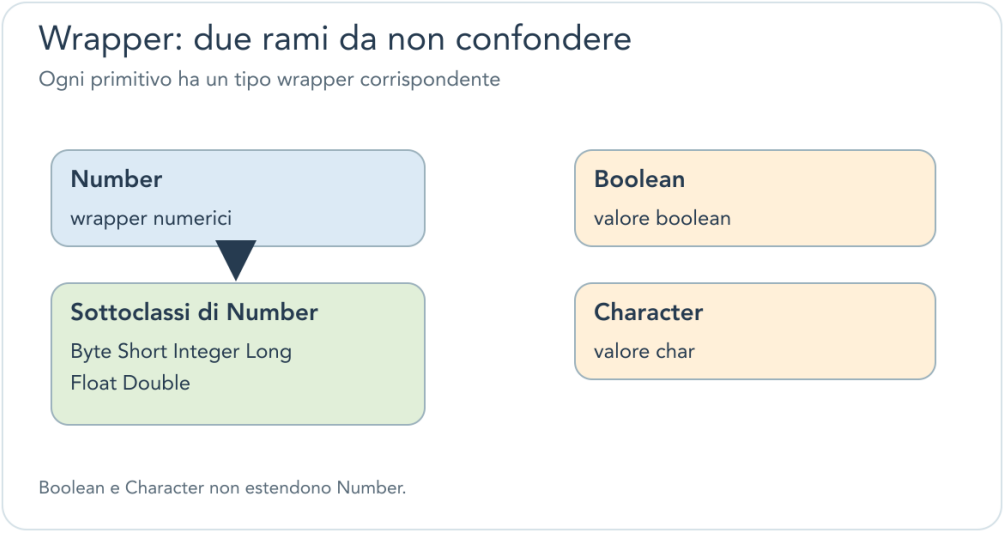


Figura 10.2 – `Boolean` e `Character` non sono sottoclassi di `Number`. Ogni primitivo ha un wrapper, ma la gerarchia delle classi non coincide con una graduatoria di conversioni numeriche.

> **Per capire – Un errore nascosto dalla sintassi.** Considera `Integer voto = null; int prossimo = voto + 1;`. La seconda riga sembra una normale addizione, ma richiede prima di estrarre un `int` da `voto`. Poiché non c'è un oggetto da cui estrarre il valore, si ottiene `NullPointerException`. Il messaggio non dice che l'operatore `+` sia difettoso: ci invita a risalire alla provenienza del riferimento nullo e a decidere che significato abbia nel contratto.

Seguire boxing e unboxing senza magie

Immagina di aggiungere un voto a una `List<Integer>`. La raccolta conserva oggetti, mentre il valore che hai calcolato può essere un `int`. In una chiamata come `voti.add(27)`, il compilatore inserisce il boxing necessario per ottenere un `Integer`. Quando leggi `int primo = voti.get(0);`, inserisce l'unboxing. Il codice è breve, ma una lettura accurata ricorda che `get(0)` restituisce un riferimento e che quel riferimento potrebbe essere nullo se la raccolta ammette e contiene `null`. Il fatto che il compilatore faccia una conversione non cambia il contratto dei dati che hai scelto di memorizzare.

Il confronto è un altro punto in cui la sintassi corta inganna. `Integer a = 100; Integer b = 100;` può far apparire vero `a == b`, perché alcune istanze sono condivise secondo le regole dell'API. Cambiare i valori può far apparire falso lo stesso confronto fra variabili che contengono numeri uguali. Un libro non dovrebbe insegnare il confine della cache come regola per confrontare numeri: dovrebbe insegnare a porre la domanda giusta. Se vuoi sapere se due valori wrapper rappresentano lo stesso numero, usa `a.equals(b)` dopo aver definito come trattare `null`. Se vuoi lavorare solo con numeri presenti e scegliere un confronto primitivo, estrai consapevolmente i valori. == fra due riferimenti rimane un confronto di identità.

Anche `Double` e `Float` richiedono attenzione diversa dagli interi: la virgola mobile ha valori speciali e arrotondamenti propri. Un wrapper non rende il calcolo esatto per magia. Quando ci occuperemo di importi di denaro, non basterà sostituire `double` con `Double`; dovremo scegliere una rappresentazione coerente con le regole dell'importo. Qui ci basta riconoscere che il wrapper risolve il problema del **tipo di oggetto**, non ogni problema matematico del valore primitivo.

> **Per verificare – Due errori di origine diversa.** `Integer.parseInt("venti")` fallisce perché il testo non rappresenta un intero valido. `Integer numero = null; int valore = numero;` fallisce perché manca l'oggetto da cui estrarre il valore. Le due istruzioni non hanno lo stesso problema: nel primo caso controlliamo il formato dell'input, nel secondo la presenza del valore. Scrivi prima questa diagnosi, poi osserva i tipi delle eccezioni durante l'esecuzione.

Un insieme dichiarato di valori

Un'**enumerazione**, dichiarata con `enum`, dà un nome e un tipo a un insieme finito di costanti. Nel programma `Stato` ammette `NUOVO` e `IN_PRESTITO`; `Stato.IN_PRESTITO` è un valore di quel tipo, non una stringa libera soggetta a refusi. Possiamo usarlo in un `switch` e, se il dominio lo richiede, aggiungere campi, costruttori e metodi all'enumerazione. Qui `consentePrestito()` risponde `true` solo per `NUOVO`: la regola resta vicina ai valori che governa. Non va scelto per elenchi che cambiano secondo i dati caricati dall'esterno.

```
enum Stato {
    NUOVO, IN_PRESTITO;

    boolean consentePrestito() {
        return this == NUOVO;
    }
}
```

Le costanti enum sono istanze definite dalla dichiarazione. Per confrontarle si usa normalmente ==, perché ciascuna costante rappresenta un'identità unica nel proprio tipo. Questo caso non smentisce il criterio spiegato per le nostre `Etichetta`: sono modelli diversi. Il nome di una costante può essere letto con `name()`, ma non conviene usarlo come testo destinato all'interfaccia se quella descrizione potrà cambiare o essere tradotta.

Dai numeri dei mesi a un tipo che si controlla

Consideriamo dodici costanti intere per i mesi. Una variabile `int mese` poteva ricevere anche `20`: il nome della costante aiutava a scrivere il valore corretto, ma non impediva di assegnarne uno fuori elenco. Con `enum Mese { GENNAIO, FEBBRAIO, ... }`, la variabile `Mese mese` può contenere una delle costanti dichiarate oppure `null`; un intero qualsiasi non è un valore di quel tipo. Non basta, però, trasformare in enum qualsiasi insieme di parole: se i valori arrivano da un archivio che l'utente può ampliare, un elenco fissato nel sorgente non descrive bene il problema.

Ogni enum dispone di `values()`, che restituisce un array delle costanti nell'ordine dichiarato. Possiamo attraversarlo con un `for` esteso, come nell'esempio dei mesi, ma non dovremmo salvare in un database la posizione numerica ottenuta da `ordinal()`: riordinare o inserire una costante cambierebbe quei numeri. Per un formato esterno serve un codice stabile deciso dal dominio. Il confronto `mese == Mese.GENNAIO` è sicuro anche se `mese` è `null`: restituisce `false`. `mese.equals(Mese.GENNAIO)`, invece, lancerebbe `NullPointerException` in quel caso.

Un enum può avere un costruttore e campi per dati che appartengono stabilmente a ciascuna costante. Nel nostro `Stato`, il metodo `consentePrestito()` è una regola semplice associata ai valori. Un campo booleano potrebbe distinguere mesi «invernali»: quell'esempio va maneggiato con cautela, perché la stagione dipende anche dall'emisfero e dal criterio meteorologico o astronomico adottato. Un campo su ogni costante può essere corretto quando la classificazione è parte esplicita del contratto del programma; se dipende da luogo e data, serve un'operazione che riceva anche quel contesto.

Quando incontreremo le raccolte, `EnumSet` ci darà un insieme specializzato di costanti dello stesso enum, mentre `EnumMap` permetterà di associare un valore a ciascuna costante. Non servono per il primo `switch` del capitolo: ricordiamo soltanto che l'enum è un tipo vero, usabile anche dalle API, e non un elenco di numeri a cui abbiamo applicato nomi più leggibili.

Un'enumerazione è più di un insieme di nomi stampabili. Se dichiariamo `enum Mese { GENNAIO, FEBBRAIO, MARZO }` per un esperimento ridotto, `Mese.GENNAIO` ha tipo `Mese`: non si può assegnare direttamente alla variabile un `int` o la stringa `"GENNAIO"`. Per trasformare un testo in una costante esiste `Mese.valueOf(testo)`, ma il metodo richiede un nome esatto e segnala un nome sconosciuto con `IllegalArgumentException`. Se il testo proviene da una persona, dobbiamo decidere come gestire maiuscole, spazi e traduzioni prima della conversione. La sicurezza del tipo nel programma non rende automaticamente validi i dati esterni.

Quando un enum possiede dati e metodi, il suo costruttore non è un invito a creare nuove costanti durante l'esecuzione. L'elenco resta quello della dichiarazione. Per un mese potremmo conservare un numero stabile deciso dal dominio e offrire un metodo `numero()`, distinguendolo da `ordinal()`, che segue semplicemente l'ordine nel sorgente. Se l'ordine delle costanti cambia per rendere il codice più leggibile, il numero di dominio deve restare quello promesso ai dati già salvati. Questa distinzione è preziosa ogni volta che il programma scambia valori con file o database.

Un enum non può estendere liberamente una nostra classe base, ma può **implementare interfacce** e quindi rispettare un contratto condiviso con altri tipi. Lo vedremo più avanti quando le interfacce avranno un ruolo concreto nel programma. Per i nomi delle costanti si usano spesso lettere maiuscole, per esempio `GENNAIO`: è una convenzione riconoscibile anche oggi, non una regola che il compilatore impone. Scegliere nomi stabili e leggibili conta soprattutto se quei nomi sono serializzati o letti da altri componenti: rinominare una costante è allora anche una modifica del contratto esterno.

Un enum come piccolo tipo del dominio

Torniamo al prestito di un libro. Con una stringa libera potremmo scrivere `"IN_PRESTITO"`, `"in prestito"` o `"prestato"` e poi ricordarci in ogni metodo che tutti e tre i testi significano la stessa cosa. Con `enum Stato { DISPONIBILE, IN_PRESTITO }`, il compilatore conosce i due valori ammessi dal programma. Un metodo `puoEsserePrestato(Stato stato)` può ricevere soltanto un valore di quel tipo o `null`; non può ricevere accidentalmente una `String`. Il controllo su `null` resta una nostra decisione: il tipo enum riduce gli stati rappresentabili, ma non elimina il riferimento nullo.

Supponiamo che `Stato` offra `consentePrestito()`. Quando il libro è `DISPONIBILE`, il metodo restituisce vero; quando è `IN_PRESTITO`, falso. Abbiamo messo la regola accanto al tipo che la descrive, ma la transizione del libro richiede ancora un'operazione: `presta()` può controllare lo stato corrente e cambiarlo. Avere un enum non impedisce a due parti del programma di modificare direttamente un campo pubblico `stato` senza rispettare le regole del prestito. L'incapsulamento del capitolo 8 e il tipo enum risolvono **problemi diversi** e lavorano bene insieme.

Possiamo attraversare i mesi con un `for` esteso, scritto come `for (Mese mese : Mese.values()) { ... }`: `values()` restituisce un array con le costanti nell'ordine in cui sono state dichiarate. È utile per costruire una scelta o verificare tutti i casi, ma non garantisce che quell'ordine sia un codice stabile per dati esterni. `toString()` può essere ridefinito per presentare una descrizione, mentre `name()` conserva il nome della costante dichiarata nel sorgente. Se un'interfaccia deve mostrare `Gennaio` in italiano e `January` in inglese, la traduzione appartiene all'interfaccia o a risorse di localizzazione, non al solo nome maiuscolo del valore enum.

Per verificare

Compila il sorgente completo con `javac --release 25 -Xlint:all -d build DemoValore.java` ed esegui `java -cp build DemoValore`. Prevedi prima le sette righe. Poi cambia solo il testo di `seconda` in `"JVM"`: quali righe cambiano? L'attività del capitolo 10 guida anche una piccola diagnosi del contratto fra `equals` e `hashCode`.

11. Stringhe e testo Unicode

Il testo è un valore, non un contenitore da modificare

Quando scriviamo `String titolo = "Java"`, otteniamo un riferimento a un oggetto `String`. Una stringa è **immutabile**: nessun metodo ne cambia il contenuto dopo la creazione. Un'operazione come `strip()` restituisce una stringa con gli spazi iniziali e finali rimossi; la variabile originale continua a riferirsi al testo precedente. È una proprietà utile quando più parti del programma condividono una stringa, ma richiede di assegnare il risultato se vogliamo conservarlo.

Nel programma completo partiamo da `" Ciao "`. La stampa del risultato di `strip()` produce `[Ciao]`, mentre una stampa successiva dell'originale produce `[ Ciao ]`. Le parentesi quadre servono solo a rendere visibili gli spazi. Non fanno parte del testo pulito.

Il confronto più frequente è quello di contenuto. `prima.equals(seconda)` è vero se le due stringhe hanno gli stessi caratteri nella stessa sequenza; `prima == seconda` chiede invece se sono proprio il medesimo oggetto. Il programma crea deliberatamente una seconda istanza con `new String("Java")`, perciò i risultati sono `false` e `true`. In codice ordinario non serve creare così una copia della stringa: qui lo facciamo per mostrare la distinzione.

> **Nota bene – Il pool delle stringhe.** Java può condividere l'oggetto associato ai letterali uguali e dispone del metodo `intern()`. Questo spiega alcuni risultati di `==` fra letterali, ma non trasforma `==` in un confronto affidabile del contenuto. Per il valore del testo si usa `equals`; quando una variabile può essere `null`, si decide esplicitamente come trattarla, ad esempio con `Objects.equals(a, b)`.

La figura 11.1 confronta i letterali con gli oggetti creati con `new`. Le frecce descrivono l'identità logica dei riferimenti; non indicano in quale area della memoria la JVM debba conservare il pool.

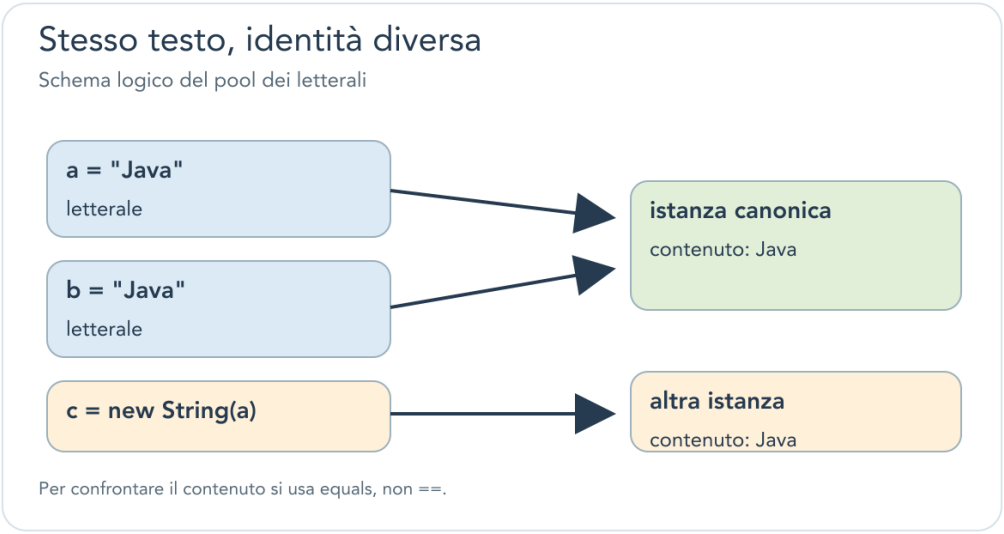


Figura 11.1 – Due letterali uguali possono indicare l'istanza canonica; `new String(a)` *crea un'istanza distinta con lo stesso contenuto.* `equals` *confronta il testo, mentre == confronta l'identità dei riferimenti.*

Dai letterali agli oggetti: quattro confronti

Un *letterale di stringa* è il testo racchiuso fra virgolette, per esempio `"Java"`; anche `""` è un letterale valido e rappresenta una stringa vuota. Possiamo chiamare un metodo direttamente su un letterale: `"Java".length()` vale 4. La forma `String titolo = "Java";` è quella ordinaria per ottenere quel valore. Scrivere `new String("Java")` crea deliberatamente un'altra istanza e, di solito, non porta alcun vantaggio per il programma applicativo. La usiamo qui soltanto per rendere visibile la distinzione fra contenuto e identità.

Confrontiamo quattro casi. `String a = "Hello world"; String b = "Hello world";` usa due letterali uguali, che si riferiscono alla stessa istanza canonica. `String c = "Hello" + " world";` è una concatenazione di espressioni costanti: il risultato è una costante di compilazione e può condividere l'istanza del letterale completo. `String d = new String(a);` è invece una nuova istanza, pur avendo lo stesso contenuto. Perciò `a == b` e `a == c` sono veri, `a == d` è falso, mentre `a.equals(d)` è vero. La regola da portare nel codice quotidiano resta `equals` per il contenuto: un testo calcolato durante l'esecuzione non va trattato come un letterale internato.

La variabile e l'oggetto hanno storie diverse. Dopo `String messaggio = "prima"; messaggio = "seconda";`, la variabile indica un altro valore; la stringa `"prima"` non è stata modificata. Se dichiariamo `final String messaggio = "prima";`, impediamo la seconda assegnazione, ma non rendiamo la classe `String` «più immutabile» di quanto già sia. È lo stesso principio incontrato con gli array, visto dall'altro lato: `final` regola la riassegnazione della variabile; la mutabilità dipende dal tipo dell'oggetto.

Seguire una trasformazione fino al risultato

Proviamo a leggere tre righe senza eseguirle: `String origine = " Java "; String pulita = origine.strip(); String maiuscola = pulita.toUpperCase();`. Dopo la prima riga, `origine` indica un testo con uno spazio prima e uno dopo. Dopo `strip()`, `pulita` indica il risultato senza quegli spazi, ma `origine` conserva il valore iniziale. La terza riga produce `"JAVA"` e non cambia né `origine` né `pulita`. Se stampiamo i tre valori fra parentesi quadre, vediamo `[ Java ]`, `[Java]` e `[JAVA]`. Questo piccolo percorso è il modo migliore per evitare l'errore di chiamare un metodo che restituisce una nuova stringa e ignorarne il risultato.

Il confronto con gli array di caratteri del C aiuta a mettere a fuoco una differenza. La distinzione utile a chi legge Java è che una `String` non è un array `char[]` a cui possiamo assegnare una nuova lettera in una posizione. Possiamo ottenere un carattere o una parte del testo attraverso i metodi della classe, ma le operazioni producono risultati nuovi quando cambiano il contenuto. La rappresentazione interna scelta dal JDK non è il contratto pubblico della `String`; non dobbiamo dedurre dalla parola «immutabile» un particolare formato fisico in memoria.

Lunghezza in Java e caratteri che vediamo

Una `String` usa indici di **unità di codice UTF-16**. `length()` conta queste unità, non sempre i caratteri che una persona vede sullo schermo. L'emoji sorridente U+1F600 è rappresentata da due unità UTF-16 e da un solo **code point** Unicode: per questo il programma stampa 2 e poi 1. La figura 11.2 visualizza le due unità come una coppia che rappresenta un unico code point.

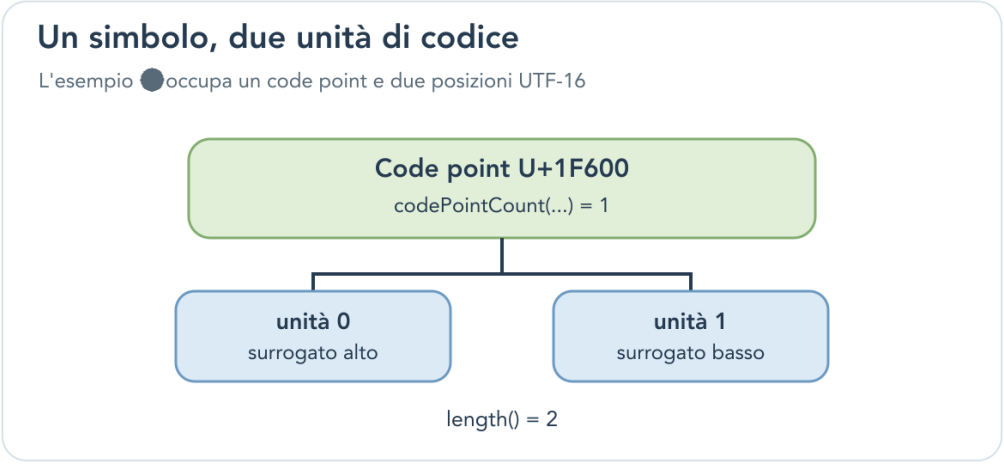


Figura 11.2 – Il simbolo dell'esempio occupa due posizioni UTF-16. `length()` *restituisce 2;* `codePointCount(0, length())` *restituisce 1.*

```
String simbolo = "\uD83D\uDE00";
System.out.println(simbolo.length());
System.out.println(simbolo.codePointCount(0, simbolo.length()));
```

Le sequenze `\uD83D` e `\uDE00` scrivono nel sorgente le due unità UTF-16 in forma esadecimale: `D83D` è il surrogato alto, `DE00` quello basso. Il carattere `u` introduce

l'escape Unicode; ogni cifra successiva rappresenta quattro bit. Il valore combinato corrisponde al code point U+1F600. Scrivere gli escape nel listato evita anche che un font privo del glifo emoji nasconda l'esempio nella pagina stampata.

Un **code point** è un valore dell'insieme Unicode. Non coincide sempre con ciò che percepiamo come un carattere: alcuni simboli visibili sono sequenze di più code point, per esempio una lettera con un segno combinante o certe emoji composte. Per un'interfaccia che deve contare i caratteri percepiti dall'utente servono regole ulteriori; `codePointCount` risolve soltanto il passaggio dalle unità UTF-16 ai code point. La specifica di `String` definisce entrambi i metodi.

`charAt(i)` restituisce la singola unità `char` all'indice `i`; su un simbolo supplementare, le due chiamate restituiscono separatamente le parti della coppia. Se vogliamo attraversare i code point, usiamo API come `codePoints()` o `codePointAt`, prestando attenzione agli indici. Un indice fuori dall'intervallo provoca un'eccezione, non un carattere vuoto.

> **Approfondimento – Tre modi di contare il testo.** Per `"A"`, `length()`, `codePointCount(0, length())` e il numero di simboli percepiti dal lettore coincidono: valgono uno. Per il simbolo U+1F600 dell'esempio, le unità UTF-16 sono due e i code point uno. Per una lettera seguita da un accento combinante, i code point possono essere due anche se il lettore vede una sola lettera accentata. La domanda «quanto è lunga la stringa?» deve quindi indicare **che cosa** vogliamo contare. Gli indici di `substring` e `charAt` restano espressi in unità UTF-16; non basta sostituire ogni `length()` con `codePointCount()` senza rivedere gli indici.

Cercare e trasformare senza cambiare l'originale

Prima di trasformare una stringa, spesso vogliamo cercare un pezzo di testo. `titolo.contains("Java")` risponde con un booleano; `titolo.indexOf("Java")` restituisce l'indice UTF-16 della prima occorrenza oppure `-1` se non la trova. Il numero `-1` non è un indice valido, ma un segnale di assenza. `substring(inizio, fine)` produce una porzione il cui inizio è incluso e la cui fine è esclusa: `"Java".substring(0, 2)` vale `"Ja"`. Il secondo limite non è la lunghezza della porzione, bensì la posizione alla quale fermarsi. Un indice che taglia fra i due `char` di una coppia surrogata può produrre una stringa difficile da interpretare come testo Unicode completo; per questo non equipariamo gli indici UTF-16 a «lettere».

Per togliere spazi ai margini possiamo usare `strip()`, che considera i caratteri di spazio secondo le regole Unicode dell'API. Il più vecchio `trim()` ha un criterio diverso, basato sui valori dei caratteri fino a U+0020. Se il testo viene da un modulo o da un file, scegliamo la regola che corrisponde ai dati attesi, e proviamo un input concreto. `toUpperCase()` e `toLowerCase()` restituiscono nuove stringhe; il risultato può dipendere dalla locale se usiamo le forme senza parametro. Per chiavi tecniche che devono comportarsi allo stesso modo indipendentemente dalla lingua dell'utente, una scelta esplicita della locale è più affidabile. Inoltre la conversione non equivale sempre a sostituire una singola lettera con un'altra: alcune trasformazioni possono cambiare la lunghezza del testo.

Per eliminare semplici spazi potremmo usare `replaceAll`. `replaceAll` interpreta il primo argomento come **espressione regolare**, un modello per cercare sequenze di testo. Se vogliamo sostituire il carattere spazio letterale, `replace(" ", "")`

comunica meglio l'intenzione e non introduce una grammatica nuova. `replaceFirst` usa invece un'espressione regolare e cambia soltanto la prima corrispondenza. Nella classe `String` di Java 25 non esiste un metodo `replaceLast`: per l'ultima corrispondenza occorre un'altra strategia, per esempio trovare la posizione con `lastIndexOf` e costruire il risultato. Questo è un caso in cui una tabella di nomi plausibili sarebbe dannosa: il lettore proverebbe codice che non compila.

> **Definizione – Espressione regolare.** È un linguaggio per descrivere insiemi di sequenze testuali. Il modello `[A-Za-z0-9]+` indica una o più lettere latine senza accenti o cifre; non indica «qualsiasi parola di qualsiasi lingua». Quando usi `replaceAll`, i caratteri speciali del modello hanno un significato che `replace` non attribuisce loro. Prima di adottare una regex, verifica esempi che devono corrispondere ed esempi che non devono corrispondere.

Una ricerca guidata, dal risultato alla scelta del metodo

Prendiamo il testo `"Java Mattone dopo Mattone"` e cerchiamo `"Mattone"`. `indexOf` restituisce la posizione della **prima** occorrenza, misurata in unità UTF-16 a partire da zero; `lastIndexOf` trova l'ultima. Qui la prima comincia a 5 e l'ultima a 18. `contains` risponde soltanto alla domanda «esiste almeno un'occorrenza?», senza darci la posizione. Se la posizione serve per estrarre una parte, controlliamo prima che non sia `-1`: usare direttamente `substring(-1)` non significa «nessun risultato», ma provoca un'eccezione. La scelta del metodo nasce quindi dalla domanda che dobbiamo risolvere, non dalla somiglianza dei nomi.

Per estrarre la prima parola, `int separatore = testo.indexOf(' ');` individua il primo spazio e `testo.substring(0, separatore)` restituisce `"Java"`. Questo funziona per il dato scelto; se la stringa non contiene spazi, `separatore` vale `-1` e occorre decidere se restituire tutto il testo, nulla o un errore. Se il testo contiene un simbolo fuori dal piano BMP prima dello spazio, l'indice ottenuto rimane corretto per `substring` perché **entrambi** usano unità UTF-16. Mescolare quell'indice con un numero di code point senza conversione produrrebbe un limite sbagliato.

Distinguiamo `replace`, `replaceFirst` e `replaceAll` sul testo `"uno uno uno"`. `replace("uno", "due")` cambia tutte le occorrenze **letterali** e produce `"due due due"`. `replaceFirst("uno", "due")` cambia la prima e produce `"due uno uno"`; il primo argomento, però, è una regex. `replaceAll("uno", "due")` produce di nuovo `"due due due"`, ma interpreta anch'esso una regex. Con la parola `uno` la differenza fra letterale e modello non si vede; con un punto si vede subito: `replace(".", "!")` cambia soltanto i punti reali, mentre `replaceAll(".", "!")` usa `.` come modello che corrisponde a molti caratteri. Se vogliamo davvero una regex per un punto letterale dobbiamo eseguire l'escaping del modello. Finché non serve il potere delle regex, `replace` è la scelta che un lettore può comprendere senza un'altra grammatica.

> **Per verificare – Output prima del codice.** Scrivi su carta il risultato atteso di `" Java ".strip()`, `"Java".substring(1, 3)` e `"uno uno".replaceFirst("uno", "due")`. Poi esegui le tre espressioni e controlla rispettivamente spazi, limite finale escluso e numero di sostituzioni. Se una previsione è sbagliata, correggi la regola che avevi usato, non soltanto la risposta.

Costruire un testo per passi

L'operatore `+` unisce stringhe ed è comodo per espressioni brevi. Se però un algoritmo aggiunge molti frammenti in un ciclo, `StringBuilder` rende esplicito un contenitore modificabile che poi produce una `String` finale. Nel nostro esempio i due `append` formano `Java`.

```
StringBuilder costruttore = new StringBuilder();
costruttore.append("Ja").append("va");
System.out.println(costruttore.toString());
```

Non serve trasformare ogni piccola concatenazione in un builder: scegliamo lo strumento in base a ciò che il codice sta facendo, e misuriamo se le prestazioni contano. `StringBuilder` non è progettato per essere modificato contemporaneamente da più thread senza coordinamento, tema che incontreremo nel capitolo sulla concorrenza.

L'operatore `+` può concatenare testo e valori di altri tipi. In `"Voti: " + 3`, il numero entra nella rappresentazione testuale e il risultato è `"Voti: 3"`. Ma l'ordine delle espressioni conta: `"Totale: " + 2 + 3` produce `"Totale: 23"`, mentre `"Totale: " + (2 + 3)` produce `"Totale: 5"`. Le parentesi dichiarano che prima vogliamo fare la somma numerica. `concat(String)` è un altro metodo che produce una nuova stringa; non modifica l'originale, come nell'esempio. Per costruzioni ripetute in un ciclo, `StringBuilder` evita di usare la variabile come se fosse una stringa modificabile, senza obbligarci ad affermare che ogni concatenazione con `+` sia lenta.

Java offre anche i **text block**, delimitati da tre virgolette, per stringhe che occupano più righe nel sorgente. Restano oggetti `String`: cambiano il modo di scrivere il letterale, non il modello dei valori. La formattazione dipende dalle regole dei rientri e delle interruzioni di riga, perciò va verificata con l'output reale quando il testo deve essere esatto. Le *string template* apparse come funzionalità preview in versioni precedenti sono state ritirate: non le presentiamo come sintassi disponibile in Java 25. Il riepilogo delle modifiche del linguaggio documenta lo stato delle funzionalità.

Il programma del text block contiene due righe e stampa `Ciao,` e `Java!` su righe separate. La riga che contiene il delimitatore di chiusura determina anche se resta un'interruzione di riga finale. Gli spazi comuni davanti alle due righe del sorgente sono trattati come rientro incidentale, non stampati prima di `Ciao` e `Java!`.

Un esempio HTML mostra bene il guadagno di leggibilità: senza text block servivano più concatenazioni, `\n` e virgolette escapate; con tre virgolette il testo assomiglia di più al documento che vogliamo produrre. Il delimitatore di apertura deve essere seguito dal terminatore di riga prima del contenuto. Il delimitatore di

chiusura decide dove il testo termina e influisce sugli spazi comuni rimossi. Un text block è comunque una `String`: se l'HTML richiede dati variabili, bisogna costruirli con le normali API, tenendo conto anche delle regole di escaping del formato di destinazione. I text block sono stabili da Java 15.

```
String messaggio = """
        Ciao,
        Java!
        """;
System.out.print(messaggio);
```

Per verificare

Compila il sorgente completo con `javac --release 25 -Xlint:all -d build DemoStringhe.java` ed esegui `java -cp build DemoStringhe`. Prova a sostituire `simbolo` con una lettera semplice e poi con una sequenza di più code point: annota ogni volta `length()` e `codePointCount`. L'attività del capitolo 11 aiuta a interpretare i numeri senza chiamarli genericamente “numero di caratteri”.

12. Eccezioni e gestione degli errori

Quando il percorso normale si interrompe

Finora abbiamo letto gli esempi seguendo le istruzioni dall'alto verso il basso. Un programma reale incontra anche situazioni in cui un'operazione non può concludersi come previsto: un file non esiste, un dato non rispetta un vincolo, un riferimento è `null`. Java rappresenta molti di questi eventi con un oggetto **eccezione**. L'istruzione `throw` avvia un'uscita dal percorso corrente; il controllo risale le chiamate fino a trovare un `catch` adatto, oppure arriva fuori dal programma e la JVM mostra uno stack trace.

Uno **stack trace** elenca le chiamate attive al momento dell'errore. Per iniziare la diagnosi, leggi prima il tipo e il messaggio dell'eccezione, poi trova la prima riga che appartiene al tuo codice. Il numero di riga è un indizio riferito alla versione compilata del sorgente, non una spiegazione completa della causa. Nel capitolo 7 avevamo provocato una `NullPointerException` chiamando un metodo su `null`: il nome dell'eccezione descrive l'evento, ma occorre comunque capire da dove sia arrivato quel riferimento.

La pila vuota e il problema del valore speciale

In una prima versione della `Pila`, `pop()` restituiva `0` quando non c'erano elementi. È un espediente che sembra risolvere il problema di un metodo obbligato a restituire un `int`, ma introduce un'ambiguità: anche `0` è un numero valido che un chiamante può avere inserito. Se `pop()` restituisce `0`, non sappiamo se abbiamo estratto proprio quel valore oppure se la pila era vuota. Una `push()` che ignori il tentativo di inserimento avrebbe il difetto complementare: quando la pila era piena, non faceva nulla e il dato andava perso senza avviso.

La versione del capitolo 8 rifiuta entrambe le operazioni impossibili con un'eccezione. Il percorso normale può così restituire **qualunque** intero, incluso zero; il percorso anomalo trasporta un tipo e un messaggio separati dal valore. Non è l'unico disegno possibile: potremmo offrire anche un metodo che controlla se la pila è vuota prima dell'estrazione. In presenza di chiamanti concorrenti, però, controllare e poi estrarre come due operazioni separate richiederebbe ulteriori garanzie. Qui la lezione è più generale: un valore speciale funziona soltanto se il dominio lo riserva davvero e se il contratto lo dichiara.

Le eccezioni non rendono innocuo un errore. Interrompono il flusso ordinario e lasciano al livello che possiede abbastanza informazioni la decisione su come reagire. Quando un costruttore scopre che i dati necessari non sono validi, può lanciare un'eccezione invece di consegnare un oggetto già incoerente. Quando fallisce una lettura da file, il programma può comunicare quale file e quale operazione erano coinvolti. In entrambi i casi il valore della soluzione sta nella qualità del contratto e della gestione, non nell'avere scritto `throw`.

Una famiglia di tipi, due obblighi diversi

`Throwable` è la radice dei tipi lanciabili. Due suoi rami principali sono `Error` ed `Exception`. Sotto `Exception`, `RuntimeException` raccoglie molte eccezioni che segnalano problemi come argomenti non validi o riferimenti nulli. La figura 12.1 mostra questa struttura essenziale.

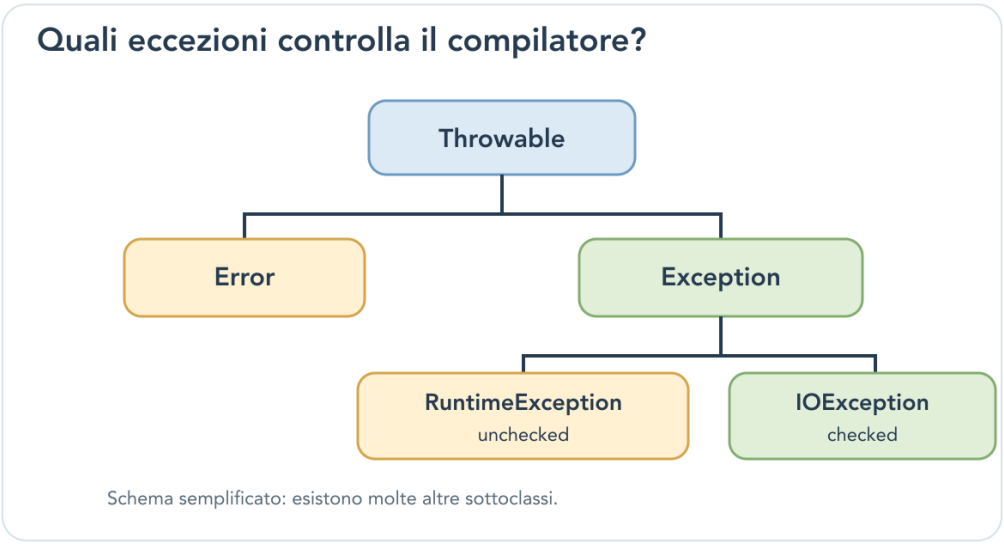


Figura 12.1 – La posizione nella gerarchia determina il controllo del compilatore. Un'eccezione che estende `Exception` *senza essere una* `RuntimeException` *è in genere* ***checked****;* `RuntimeException` *ed* `Error` *sono* ***unchecked****.*

Una **eccezione checked**, come `IOException`, deve essere gestita con `catch` oppure dichiarata con `throws` da un metodo che può lasciarla propagare. Un'eccezione **unchecked** non ha questo obbligo di compilazione. La distinzione non dice che una sia sempre più grave dell'altra: descrive che cosa il linguaggio impone al chiamante. Gli `Error` rappresentano situazioni che un normale programma di solito non cerca di recuperare catturandole genericamente. La specifica Java 25 definisce la classificazione e le regole di controllo.

Per leggere la gerarchia senza imparare una lista di nomi, seguiamo quattro casi concreti. `ClassNotFoundException` estende `Exception` e non `RuntimeException`: è checked in un'API che cerca una classe per nome. `IndexOutOfBoundsException` estende `RuntimeException`: un indice fuori dai limiti viene segnalato durante l'esecuzione, senza obbligo di `throws`. `NullPointerException` appartiene allo stesso ramo unchecked. `OutOfMemoryError` è invece un `Error`; non diventa un'eccezione checked soltanto perché interrompe un programma. La grafia `IndexOutOfBoundException` manca della `s` finale e non è il nome del tipo standard.

Tipo dell'esempio	Ramo della gerarchia	Obbligo del compilatore
`ClassNotFoundException`	`Exception` checked	gestire o dichiarare se l'API la può lanciare

Tipo dell'esempio	Ramo della gerarchia	Obbligo del compilatore
`IndexOutOfBoundsException`	`RuntimeException`	nessun obbligo di `catch` o `throws`
`NullPointerException`	`RuntimeException`	nessun obbligo di `catch` o `throws`
`OutOfMemoryError`	`Error`	nessun obbligo di `catch` o `throws`

La tabella non suggerisce di catturare sistematicamente le ultime tre. Un indice errato spesso richiede di correggere i limiti del ciclo o controllare un dato in ingresso. Un riferimento nullo richiede di capire perché il contratto permetteva o non permetteva quell'assenza. Un processo che ha esaurito memoria potrebbe non essere in grado di completare neppure un'operazione di recupero; prendere `Error` genericamente per proseguire sarebbe una promessa che il programma di solito non può mantenere. Per contro, un'eccezione checked non garantisce da sola che la gestione sia buona: un `catch` vuoto soddisfa la forma richiesta dal compilatore ma perde il problema.

Anche una classe definita da noi può rappresentare un errore specifico. Nel programma dei voti, `VotoNonValidoException` estende `Exception`: il metodo `leggiVoto` la dichiara con `throws`, mentre il chiamante la gestisce con `catch`. Per `"27"` otteniamo un voto; per `"31"` un valore fuori intervallo; per `"xx"` l'errore di conversione viene conservato come causa. Il nome del tipo comunica al chiamante quale operazione non è riuscita meglio di un generico `Exception`.

```
static int leggiVoto(String testo) throws VotoNonValidoException {
    try {
        int voto = Integer.parseInt(testo);
        if (voto < 0 || voto > 30) {
            throw new VotoNonValidoException("fuori intervallo: " + voto);
        }
        return voto;
    } catch (NumberFormatException causa) {
        throw new VotoNonValidoException("non è un numero: " + testo, causa);
    }
}
```

La chiamata a `Integer.parseInt` può lanciare `NumberFormatException`, una `RuntimeException` e quindi unchecked. Il nostro `catch` la trasforma in un errore del dominio dei voti mantenendo la causa originale. Il `throw` nel ramo fuori intervallo lancia invece direttamente la nostra eccezione checked. Il compilatore richiede che il metodo dichiari questa possibilità, e che chi lo chiama la tratti o la dichiari a sua volta.

`throw` e `throws` si assomigliano nella scrittura ma occupano posti diversi. `throw new VotoNonValidoException(...)` è un'**istruzione** dentro il metodo: in quel punto nasce l'uscita anomala. `throws VotoNonValidoException` fa parte della **dichiarazione** del metodo e avvisa il chiamante della possibilità di propagazione checked. La dichiarazione non lancia nulla da sola. Un metodo può dichiarare più tipi checked, ma un elenco sempre più lungo può segnalare che il confine della sua responsabilità non è chiaro.

Vediamo che cosa accade a `throws` quando un metodo viene ridefinito. Se la classe base promette un metodo che può lanciare una certa eccezione checked, la sottoclasse può realizzarlo senza lanciare quell'eccezione oppure dichiararne una più specifica compatibile. Non può sorprendere chi usa il tipo base aggiungendo una nuova eccezione checked più ampia che il contratto della base non prevedeva: il chiamante, compilato contro la base, non sarebbe stato obbligato a gestirla. Le eccezioni unchecked seguono regole diverse e non richiedono quella dichiarazione. È un altro esempio di sostituibilità: la firma della sottoclasse deve rispettare ciò che il chiamante era autorizzato ad aspettarsi.

Anche dentro un `catch` possiamo decidere di propagare. Se abbiamo soltanto aggiunto una riga di log e scriviamo `throw errore;`, lasciamo risalire l'oggetto catturato. In alcuni casi il compilatore riesce a determinare tipi checked più precisi di quelli apparentemente indicati dalla variabile del `catch`: è il cosiddetto *precise rethrow*. Non serve usarlo per il primo programma sulle eccezioni, ma evita una conclusione falsa: catturare con una variabile di tipo generale non obbliga sempre a dichiarare `throws Exception` quando rilanciamo l'eccezione senza riassegnare quella variabile. La specifica del controllo delle eccezioni definisce i limiti esatti; nel codice quotidiano preferiamo comunque firme che espongano i tipi utili al chiamante.

Nel gestore `catch (VotoNonValidoException errore)`, il tipo fra parentesi determina quali oggetti può intercettare. Il nome `errore` è una variabile locale con cui leggere messaggio, causa e altri dati dell'eccezione catturata. Il `catch` non riprende l'esecuzione dalla riga che ha fallito: si entra nel blocco di gestione e, se questo termina normalmente, si prosegue dopo il costrutto `try-catch`. Le istruzioni rimaste nel blocco `try` dopo il punto di errore sono saltate. Per questo mettere dentro un grande `try` molte operazioni indipendenti può rendere difficile capire quali siano state completate.

> **Definizione – Causa e soppressione.** La *causa* è l'errore precedente che una nuova eccezione conserva per spiegare da dove deriva. Un'eccezione *soppressa* è invece un altro errore avvenuto durante la chiusura di una risorsa mentre era già presente un errore principale. `getCause()` e `getSuppressed()` raccontano quindi due relazioni diverse; non sono due nomi per la stessa lista.

> **Concetto chiave – Scegliere la risposta.** Un `catch` dovrebbe compiere un'azione sensata nel punto in cui il problema è comprensibile: mostrare un messaggio utile, riprovare se è appropriato, usare un'alternativa o aggiungere contesto e propagare. Catturare `Exception` per stampare una riga e continuare può lasciare il programma in uno stato errato.

Che cosa contiene un oggetto eccezione

Leggiamo i costruttori di `Throwable` a partire dalle informazioni che vogliamo conservare. `new Exception()` crea un'eccezione senza un messaggio scelto da noi; `new Exception("lettura fallita")` aggiunge un messaggio. La forma con un'altra eccezione come **causa**, `new Exception("catalogo non disponibile", errore)`, conserva sia il contesto nuovo sia l'evento da cui siamo partiti. Esiste anche la forma che riceve soltanto la causa. Le sottoclassi non espongono necessariamente tutti i costruttori con le stesse firme: quando definiamo una nostra eccezione, scegliamo esplicitamente quali offrire e chiamiamo il costruttore appropriato della superclasse.

Il messaggio è una frase per comprendere il problema, non un campo strutturato da analizzare per prendere decisioni. Un programma che vuole distinguere «voto fuori intervallo» da «testo non numerico» può usare tipi specifici o dati propri dell'eccezione, anziché cercare parole in `getMessage()`. Il messaggio può essere `null`; inoltre può cambiare per migliorare la diagnosi senza che cambi il significato dell'errore. La causa, quando esiste, conserva la traccia originale e si ottiene con `getCause()`. Se la scartiamo nel momento in cui traduciamo un'eccezione di basso livello in una del dominio, chi diagnostica il problema perde una parte importante della storia.

> **Nota bene – Tipo, messaggio, causa e traccia.** Il *tipo* permette al `catch` di selezionare ciò che sa gestire. Il *messaggio* spiega l'evento a una persona. La *causa* collega un errore a quello che lo ha provocato. Lo *stack trace* mostra il percorso delle chiamate attive nel punto in cui l'eccezione è stata prodotta. Nessuno di questi quattro elementi sostituisce gli altri; stamparne uno soltanto può rendere difficile la diagnosi.

Una `IOException` è checked: se un metodo la lascia uscire, il suo contratto deve dichiararlo con `throws` e il chiamante deve a sua volta gestirla o dichiararla. `NumberFormatException`, pur potendo essere prevista davanti a dati esterni, è unchecked: il compilatore non obbliga a citarla nella firma. La classificazione dipende dalla posizione del tipo nella gerarchia, non dal fatto che l'errore ci sembri «frequente». Un metodo che riceve input umano può comunque scegliere di intercettare la conversione e restituire un risultato più adatto all'interfaccia. Al contrario, dichiarare `throws Exception` per evitare di scegliere i tipi perde informazione utile per chi chiama.

Dove gestire un problema: tre livelli della stessa operazione

Supponiamo che `leggiCatalogo` apra un file, `mostraCatalogo` chiami quel metodo e `main` avvii il programma. Se il file non esiste, `leggiCatalogo` conosce il percorso tentato e può aggiungerlo al messaggio. `mostraCatalogo` conosce lo scopo dell'operazione e può decidere se esiste un catalogo alternativo. `main` conosce il contesto dell'utente e può mostrare un avviso, terminare con un esito chiaro o chiedere un nuovo percorso. Non c'è una regola che obblighi il metodo più vicino al file a catturare l'eccezione: la gestione va nel livello che possiede davvero una risposta sensata.

Se `leggiCatalogo` cattura `IOException` e restituisce una lista vuota, il chiamante potrebbe mostrare «nessun libro» anche se in realtà non è riuscito a leggere nulla.

Abbiamo trasformato un guasto in un risultato valido e perso l'informazione necessaria per reagire. Può essere corretto restituire una lista vuota quando il file è stato letto e non contiene libri; è un evento diverso. Questa distinzione aiuta a decidere se usare un valore di ritorno, un'eccezione checked o un'eccezione unchecked. La scelta non nasce dal desiderio di scrivere meno `catch`, ma dal significato che il chiamante deve poter attribuire all'esito.

Un metodo che dichiara `throws IOException` non si assume che l'errore avverrà sempre: dichiara una possibilità del suo contratto. Un chiamante può catturarla e scegliere un'alternativa; un altro metodo può dichiararla a sua volta e lasciare la decisione al proprio chiamante. Ogni propagazione conserva la stessa eccezione finché non la si intercetta o si lancia un nuovo oggetto con una causa. Quando si aggiunge contesto, un messaggio come `"Impossibile leggere catalogo.txt"` aiuta più di `"Errore"`, ma non deve contenere credenziali o dati che non dovrebbero finire nei log.

> **Per capire – `throw` non è `return`.** `return` completa normalmente un metodo consegnando il valore dichiarato; `throw` interrompe il percorso normale e cerca un gestore. Dopo `throw`, le istruzioni successive dello stesso blocco non vengono eseguite lungo quel percorso. Un `catch` compatibile può gestire l'evento più in alto nella catena; se non c'è, l'errore arriva fuori dal punto di ingresso e il runtime lo segnala. È questo movimento fra livelli, e non il solo nome dell'oggetto, che chiamiamo propagazione.

Seguire la pila delle chiamate

Torniamo alla classe `Pila` del capitolo 8, senza crearne una nuova versione. `estrai()` lancia `IllegalStateException` con il messaggio `pila vuota` quando non c'è un elemento da restituire. Nel programma sulla propagazione, `main` chiama `mostra`, che chiama `leggi`, che infine chiama `estrai`. Nessuno dei due metodi intermedi ha un `catch`: l'eccezione risale fino al gestore di `main`.

```
static int leggi(Pila pila) {
    return pila.estrai();
}

static int mostra(Pila pila) {
    return leggi(pila);
}
```

La figura 12.2 segue l'eccezione dal punto in cui nasce al gestore. Non mostra una pila di numeri come quella del capitolo 8: mostra la **pila delle chiamate**, cioè i metodi ancora attivi nel momento dell'errore. Sono due usi della parola *pila*, collegati dall'ordine in cui le chiamate entrano ed escono, ma non sono lo stesso oggetto del programma.

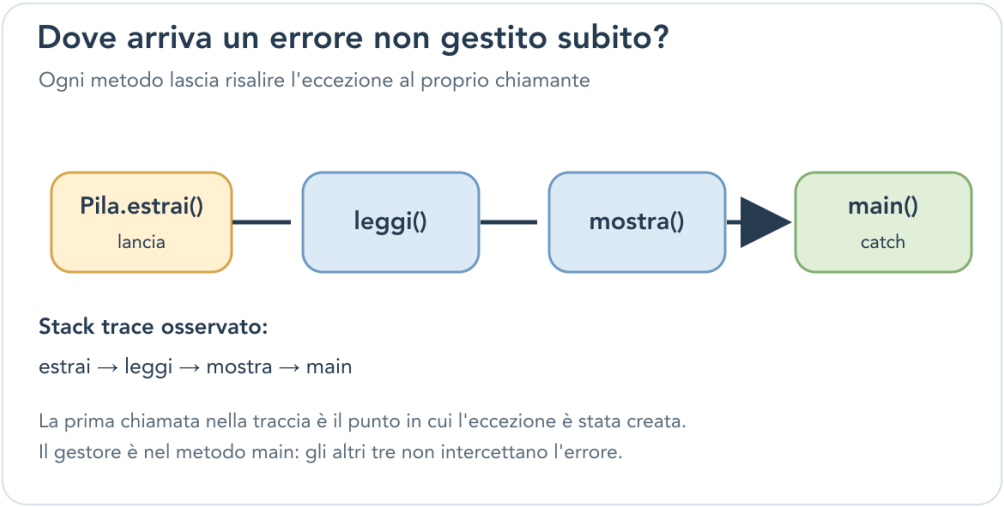


Figura 12.2 – L'eccezione parte da `Pila.estrai()` *e viene intercettata nel* `main`*.* `leggi` *e* `mostra` *lasciano risalire l'errore perché non hanno un gestore adatto.*

Il programma stampa `pila vuota`, poi i nomi `estrai`, `leggi`, `mostra` e `main`. Per ottenere questi nomi legge i `StackTraceElement` dell'eccezione, oggetti che descrivono le singole chiamate. Nella stampa ordinaria di uno stack trace comparirebbero anche file e numeri di riga; il nostro esempio mostra solo i nomi, così l'output resta stabile se spostiamo il codice nel file. Per riprodurlo, compila insieme `Pila.java` e `DemoPropagazione.java` con `javac --release 25 -Xlint:all -d build Pila.java DemoPropagazione.java` da una cartella in cui hai copiato i due sorgenti, poi avvia `java -cp build DemoPropagazione`.

> **Nota bene – Propagazione non significa soluzione.** Togliere un `catch` da `leggi` non corregge la pila vuota: permette soltanto a un livello chiamante di decidere che cosa fare. Nel nostro `main` la catturiamo per osservare la traccia; in un'applicazione il contratto dovrebbe spiegare quando l'estrazione è lecita e quale risposta è appropriata se la pila è vuota.

Leggere una traccia partendo dal punto giusto

Se un'eccezione nasce in `Pila.estrai()`, la prima riga applicativa dello stack trace indica proprio quel metodo; le righe successive mostrano `leggi`, `mostra` e `main` nell'ordine inverso alle chiamate fatte per arrivarci. Non leggere la traccia come un elenco di istruzioni eseguite dall'alto verso il basso: è una fotografia delle chiamate ancora attive al momento del problema. La riga più in alto individua dove l'eccezione è stata costruita o lanciata; la causa logica può essere più indietro, per esempio nel codice che ha chiamato `estrai` senza aver inserito nulla.

Alcuni errori ricorrenti meritano nomi precisi. `NullPointerException` può nascere quando si dereferenzia `null`; `ArrayIndexOutOfBoundsException` quando si accede a un indice fuori dall'array; `ArithmeticException` per una divisione intera per zero. La divisione in virgola mobile per zero segue regole diverse e può produrre infinito o NaN. `ClassNotFoundException` è checked in API che cercano una classe per

nome; `OutOfMemoryError` appartiene al ramo `Error`, e intercettarlo come un normale caso di recupero non rende affidabile un processo che ha esaurito memoria. Il tipo restringe la diagnosi, ma nessuno di questi nomi spiega da solo *perché* il programma ha raggiunto quella situazione.

Per una verifica rapida, puoi chiedere a un'eccezione `getMessage()`, `getCause()` e `getStackTrace()`. Il messaggio può essere assente; la causa può essere `null`; gli elementi della traccia possono includere codice della libreria e del runtime. In un'applicazione reale spesso è meglio registrare l'eccezione completa con uno strumento di logging, conservando le cause, invece di stampare soltanto `errore.getMessage()` e perdere il percorso che ha condotto al guasto.

Ritrovare il caso dell'array nullo

Seguiamo `main`, `metodo1` e `metodo2`: il primo passa un array nullo al secondo, che lo passa al terzo, dove un accesso all'elemento zero causa `NullPointerException`. La stessa catena compare nel sorgente `DemoTracciaNull.java`, ma stampiamo soltanto tipo e nomi dei metodi, così l'output non dipende dai numeri di riga del file.

```
public class DemoTracciaNull {
    static void metodo1(int[] a) {
        metodo2(a);
    }

    static void metodo2(int[] b) {
        System.out.println(b[0]);
    }

    public static void main(String[] args) {
        try {
            metodo1(null);
        } catch (NullPointerException errore) {
            System.out.println(errore.getClass().getSimpleName());
            for (StackTraceElement elemento : errore.getStackTrace()) {
                System.out.println(elemento.getMethodName());

            }
        }
    }
}
```

Le righe sono `NullPointerException`, `metodo2`, `metodo1`, `main`. La chiamata è avvenuta nell'ordine `main` → `metodo1` → `metodo2`, mentre la traccia elenca per primo il punto del problema e poi i chiamanti. Questa inversione non è un dettaglio grafico: spiega perché il metodo in cima alla traccia è spesso il posto migliore in cui osservare l'operazione fallita, ma il valore nullo può avere origine più in basso nella storia delle chiamate. In questo esempio lo abbiamo passato deliberatamente da `main`; in un programma reale potrebbe provenire da un file, da un parametro o da un risultato non controllato.

I JDK moderni possono offrire messaggi di `NullPointerException` più precisi, indicando l'operazione tentata su un riferimento nullo. Il testo esatto del messaggio non è un'API da analizzare per prendere decisioni, e può dipendere dal contesto. Il nostro programma lo evita nell'output atteso, ma durante la diagnosi va letto: spesso aiuta a riconoscere quale riferimento era nullo. La soluzione non è catturare

ogni `NullPointerException` e continuare; è capire se il contratto ammetteva `null` e correggere la sorgente del valore o il controllo mancante.

Tre esiti dello stesso calcolo

Prima di occuparci della chiusura delle risorse, osserviamo un programma in cui il percorso normale e due errori diversi partono dalla stessa operazione. `prova` riceve due stringhe, le converte in interi e divide il primo numero per il secondo. L'esempio è completo nel sorgente `DemoDivisione.java`.

```
public class DemoDivisione {
    static void prova(String primo, String secondo) {
        try {
            int a = Integer.parseInt(primo);
            int b = Integer.parseInt(secondo);
            System.out.println("risultato: " + (a / b));
        } catch (NumberFormatException errore) {
            System.out.println("serve un numero intero");
        } catch (ArithmeticException errore) {
            System.out.println("non si divide per zero");
        } finally {
            System.out.println("prova terminata");
        }
    }

    public static void main(String[] args) {

        prova("8", "2");
        prova("otto", "2");
        prova("8", "0");
    }
}
```

La prima chiamata stampa `risultato: 4` e poi `prova terminata`. Nella seconda, `Integer.parseInt("otto")` lancia `NumberFormatException`: non si arriva alla divisione, si stampa `serve un numero intero` e quindi `prova terminata`. Nella terza, la conversione riesce ma la divisione intera per zero lancia `ArithmeticException`: si stampa `non si divide per zero` e infine `prova terminata`. In tutto vediamo sei righe, due per ogni chiamata. Il blocco `finally` viene eseguito anche dopo un `catch`; non sostituisce il gestore e non rende valido un dato errato.

I due `catch` sono separati perché il programma vuole dare messaggi diversi. Se la risposta fosse davvero identica, potremmo scrivere `catch (NumberFormatException | ArithmeticException errore)` e un solo corpo. Non potremmo invece mettere nello stesso multi-catch due tipi dei quali uno è sottotipo dell'altro: il ramo più specifico sarebbe già coperto dall'altro. Né avrebbe senso catturare genericamente `Throwable` per nascondere ogni problema. Il gestore si sceglie in base al **tipo** dell'eccezione che arriva dal `try`; se più gestori separati sono applicabili, l'ordine deve lasciare prima quello più specifico.

Vediamo la forma completa nel programma `DemoMultiCatch`. Stavolta le due anomalie hanno una risposta comune: l'operazione richiesta non può produrre un quoziente, quindi il metodo segnala che il tentativo è stato rifiutato. Il nome del

tipo resta nella stampa per capire **quale** controllo ha fermato il calcolo; non usiamo quel nome come messaggio destinato all'utente finale.

```
public class DemoMultiCatch {
    static void calcola(String dividendo, String divisore) {
        try {
            int primo = Integer.parseInt(dividendo);
            int secondo = Integer.parseInt(divisore);
            System.out.println("quoziente: " + primo / secondo);
        } catch (NumberFormatException | ArithmeticException errore) {
            System.out.println("rifiutato: " + errore.getClass().getSimpleName());
        }
    }

    public static void main(String[] args) {
        calcola("12", "3");
        calcola("dodici", "3");
        calcola("12", "0");
    }

}
```

Prima di eseguirlo, percorri le tre chiamate. La prima passa entrambe le conversioni e stampa `quoziente: 4`. La seconda si ferma già su `Integer.parseInt("dodici")` e stampa `rifiutato: NumberFormatException`; la divisione non viene tentata. La terza converte i due valori, ma il divisore zero provoca `ArithmeticException`, perciò stampa `rifiutato: ArithmeticException`. La risposta comune non cancella la distinzione fra i due eventi. Se volessimo chiedere all'utente di correggere un numero oppure vietare esplicitamente lo zero, torneremmo ai due gestori separati del programma precedente, perché le risposte sarebbero diverse.

Il multi-catch non significa «cattura tutte le eccezioni»: nomina soltanto i tipi elencati. Per esempio, il calcolo potrebbe ricevere il testo `"8"` e `"0"`: `ArithmeticException` è gestita; potrebbe anche contenere un errore di programmazione in un'altra riga che produce `NullPointerException`, e il multi-catch mostrato non lo intercetterebbe. Questo è desiderabile quando la risposta «input non valido» sarebbe fuorviante per un difetto diverso. La precisione del gestore protegge il significato dell'errore. Se più tipi condividono davvero una risposta, il blocco comune evita duplicazione; se richiedono messaggi o recuperi diversi, due `catch` separati raccontano meglio il programma.

Consideriamo poi l'ordine di due gestori ipotetici: `catch (RuntimeException e)` seguito da `catch (NumberFormatException e)`. Il secondo non potrebbe mai ricevere una `NumberFormatException`, perché il primo la catturerebbe già: è una sottoclasse di `RuntimeException`. Il compilatore segnala il gestore irraggiungibile. Invertendo l'ordine, il caso specifico può essere trattato prima e il caso generale resta per altri errori runtime. Anche quando il codice compila, però, un gestore ampio può nascondere difetti inattesi. La domanda da porre non è «quanti tipi posso mettere nel catch?», ma «per quali di questi eventi so davvero scegliere una risposta?».

Il `finally` del programma serve a mostrare il flusso. Immagina di aggiungere una riga `System.out.println("dopo")` dopo l'intero `try-catch-finally`: in questi tre

casi verrebbe eseguita dopo `prova terminata`, perché ogni errore è stato gestito. Se togliessimo il `catch` adatto, il `finally` sarebbe comunque attraversato, ma l'eccezione continuerebbe a propagarsi e la riga `dopo` non verrebbe raggiunta da quel percorso. La differenza fra «esegui un'azione finale» e «risolvi l'errore» è il cuore del costrutto. Un `return` nel `finally` renderebbe il percorso ancora più difficile da seguire e potrebbe nascondere un'eccezione già in viaggio.

> **Per capire l'ordine.** Immagina di dichiarare `int risultato;` prima del `try`, assegnargli `a / b` dentro il `try` e stamparlo dopo il `try-catch`. Il compilatore segnalerà che `risultato` potrebbe non essere stato inizializzato: uno dei due percorsi di errore salta infatti l'assegnazione. Disegnare i tre percorsi prima di spostare le istruzioni aiuta a distinguere il flusso normale da quello eccezionale.

Chiudere una risorsa anche quando qualcosa fallisce

Una risorsa come un file o una connessione va rilasciata quando il suo uso termina. Il costrutto **try-with-resources** chiama automaticamente `close()` sugli oggetti dichiarati nella parentesi del `try` che implementano `AutoCloseable`. Non è un sostituto della comprensione degli errori: rende affidabile il passaggio di chiusura anche quando il corpo lancia un'eccezione.

Nel programma completo usiamo una risorsa didattica che stampa `chiusura` e poi lancia un'`IOException`. Anche il corpo del `try` lancia un'`IOException`. È una situazione costruita apposta per vedere quale errore resta principale.

```
try (Risorsa risorsa = new Risorsa()) {
    risorsa.usa();
    throw new IOException("errore durante il lavoro");
} catch (IOException errore) {
    System.out.println("principale: " + errore.getMessage());
    for (Throwable secondaria : errore.getSuppressed()) {
        System.out.println("secondaria: " + secondaria.getMessage());
    }
}
```

L'output, nell'ordine, è `lavoro`, `chiusura`, `principale: errore durante il lavoro` e `secondaria: errore durante la chiusura`. L'errore nel corpo resta quello principale; l'errore di chiusura viene conservato fra le **eccezioni soppresse**, accessibili con `getSuppressed()`. Non viene perso. Se il corpo terminasse normalmente e fallisse soltanto `close()`, sarebbe invece l'errore di chiusura a essere propagato. La documentazione di `AutoCloseable` spiega il contratto della chiusura.

La variabile `risorsa` rende evidente quale oggetto sarà chiuso. Se ne dichiariamo più di uno nella parentesi, Java li chiude in ordine inverso rispetto alla loro creazione. In un'applicazione reale il metodo `close()` deve fare il possibile per rilasciare ciò che possiede anche quando segnala un problema; il nostro esempio lancia sempre soltanto per rendere osservabile la regola delle eccezioni soppresse.

Con due risorse, il percorso si può seguire come una piccola pila. Se scriviamo `try (Risorsa prima = ...; Risorsa seconda = ...)`, `seconda` viene chiusa prima di `prima`. Se il corpo e entrambe le chiusure falliscono, l'errore del corpo resta

principale e gli errori di chiusura vengono conservati come soppressi, nell'ordine in cui emergono. Se il corpo termina normalmente, un errore di chiusura può diventare l'eccezione principale. Questa regola impedisce che un problema secondario durante la pulizia cancelli il problema che ha interrotto il lavoro. La struttura delle eccezioni non è però una scusa per ignorare il rilascio delle risorse: nel codice applicativo bisogna ancora decidere quali dati e quali file sono validi dopo un fallimento.

Una risorsa dichiarata nella parentesi deve essere compatibile con `AutoCloseable`, cioè offrire l'operazione `close()` prevista dall'interfaccia. Il costrutto governa **la chiusura**; non sa se il file che abbiamo scritto contiene dati corretti, né annulla le modifiche già compiute dal corpo del `try`. Se il programma deve mantenere un'operazione atomica, occorre una strategia ulteriore, per esempio scrivere in un file temporaneo e sostituire quello finale soltanto dopo il successo. Il confine fra gestione della risorsa e consistenza dei dati merita di essere nominato: altrimenti «chiuso automaticamente» sembra promettere più di quanto faccia.

Il vecchio stile con un `finally` che chiama manualmente `close()` obbliga a considerare varie situazioni: la risorsa potrebbe non essere stata aperta, il lavoro potrebbe fallire, e la chiusura stessa potrebbe lanciare un'altra eccezione. Un `finally` scritto ingenuamente può sostituire l'errore del lavoro con quello della chiusura, facendo perdere la causa che ci interessava per prima. Il try-with-resources rende esplicita la proprietà della risorsa e conserva l'errore di chiusura fra i soppressi quando esiste già un errore principale. Non impedisce di usare un `finally` per altre attività di completamento; distingue semplicemente ciò che il linguaggio può gestire in modo strutturato dalla logica specifica dell'applicazione.

> **Per verificare – Due fallimenti, due relazioni.** Nel nostro `DemoRisorse`, l'eccezione di chiusura compare in `getSuppressed()` dell'errore principale. Se invece un metodo catturasse un errore di lettura e lanciasse una nuova eccezione «catalogo non disponibile», l'errore iniziale dovrebbe comparire in `getCause()` della nuova eccezione. Disegna due frecce diverse: *errore principale → errore di chiusura soppresso* e *nuova spiegazione → causa originale*. Chiamare entrambi «errore precedente» impedisce di leggere correttamente una diagnosi reale.

La distinzione conta anche quando stampiamo la traccia completa: le cause e le eccezioni soppresse possono comparire con intestazioni diverse e con parti comuni della pila abbreviate. Non è una perdita casuale di righe; la presentazione cerca di evitare di ripetere le stesse chiamate. Prima di concludere che una risorsa non sia stata chiusa o che una causa manchi, consulta l'oggetto dell'eccezione e le sue relazioni esplicite. Il semplice messaggio principale, isolato dal resto, non racconta necessariamente tutta la sequenza del fallimento.

Un blocco `finally` viene eseguito quando si lascia il corrispondente `try`, sia che il corpo termini normalmente sia che un'eccezione venga lanciata, salvo eventi che interrompono l'esecuzione della JVM. È utile per operazioni di completamento che non sono risorse `AutoCloseable`; per queste ultime, `try-with-resources` rende più chiaro anche l'ordine di chiusura e la conservazione delle eccezioni soppresse. Evita di introdurre un nuovo `return` o un nuovo `throw` in `finally` senza comprenderne l'effetto: può sostituire il risultato o l'eccezione precedente.

Quando due tipi di errore indipendenti richiedono la stessa risposta, Java consente un **multi-catch**, per esempio `catch (IOException | NumberFormatException errore)`. Il gestore deve avere senso per entrambi. L'ordine dei `catch` separati conta: un tipo più specifico deve essere intercettato prima di una sua superclasse, altrimenti il ramo specifico non sarebbe raggiungibile. Queste scelte si applicano a errori reali del metodo; non occorre avvolgere ogni riga in un `try`.

Propagare senza perdere la causa

Un metodo può aggiungere contesto a un errore prima di lasciarlo risalire. Per esempio, quando la lettura di un file fallisce, il livello che conosce il nome del documento può lanciare una nuova eccezione con un messaggio che lo includa, conservando l'eccezione originale come **causa**. Così il chiamante legge il significato dell'operazione fallita e può ancora risalire all'errore iniziale con `getCause()`.

```
throw new IllegalStateException("Impossibile caricare il catalogo", errore);
```

Questa riga è un frammento da collocare in un contesto in cui `errore` sia stato catturato. Non trasformiamo però ogni errore in una `RuntimeException` solo per evitare `throws`: se il chiamante può reagire all'evento, la scelta fra gestione e propagazione fa parte del contratto del metodo. Per un argomento che viola un vincolo immediato, come il voto `31` del capitolo 8, `IllegalArgumentException` comunica invece che il chiamante ha passato un valore non ammesso.

> **Nota bene – Non usare eccezioni per il percorso ordinario.** Una ricerca che può non trovare un elemento dovrebbe esprimere quel risultato nel proprio contratto, anziché lanciare un'eccezione a ogni mancata corrispondenza. Le eccezioni servono a segnalare percorsi che richiedono attenzione e gestione. La scelta concreta dipende dall'API e dal dominio, non dal solo fatto che `try` e `catch` siano disponibili.

Prima di scrivere un `catch`, formulare la risposta

Se un utente inserisce un voto testuale non valido, possiamo chiedergli di correggerlo. Se il programma legge un file che non esiste, possiamo permettere di scegliere un altro percorso. Se una pila è piena, possiamo rifiutare il nuovo elemento e mostrare la capacità raggiunta. Queste tre risposte appartengono a contesti diversi e richiedono informazioni diverse. Scrivere `catch (Exception errore) { System.out.println("Errore"); }` per tutti i casi cancella proprio le informazioni che il meccanismo delle eccezioni era stato progettato per trasportare.

Un buon esercizio consiste nel leggere ogni gestore e completare la frase: «Dopo questo blocco, il programma può proseguire perché...». Nel `DemoDivisione`, il metodo `prova` termina dopo aver spiegato il risultato o l'input errato; non consegna al resto dell'applicazione un numero inventato. In `DemoRisorse`, il gestore mostra sia l'errore principale sia quello di chiusura, poi l'esempio termina. Nel caso della pila checked, il gestore sa quale valore è stato rifiutato e può decidere se comunicarlo o proporre una pila più capiente. Se non riusciamo a completare la

frase senza fingere che un'operazione sia riuscita, quel livello probabilmente deve aggiungere contesto e propagare, oppure lasciare risalire l'eccezione.

Le eccezioni possono nascere anche in un costruttore. Il `Cliente` del capitolo 8 rifiuta nome e cognome vuoti; se il costruttore lancia `IllegalArgumentException`, il chiamante non ottiene un cliente completato. In una lambda passata a un'API funzionale o in un'attività concorrente, l'errore segue inoltre il contratto dell'interfaccia o del meccanismo che esegue il lavoro: non è corretto supporre che arrivi sempre nel `catch` posto attorno alla riga che ha avviato il task. Quando incontreremo lambda e concorrenza torneremo su quei confini con esempi completi. Qui fissiamo il principio generale: ogni passaggio di controllo va letto nel contesto di **chi esegue davvero** l'operazione che può fallire.

Progettare una nostra eccezione senza perdere il dominio

`VotoNonValidoException` racconta meglio di una generica `Exception` quale operazione è fallita. Il suo costruttore che riceve un messaggio serve quando il voto numerico è fuori intervallo; quello che riceve anche una causa serve quando la conversione del testo è fallita. Il chiamante può trattare entrambi come mancata acquisizione di un voto, mentre chi diagnostica il programma conserva la differenza fra i due percorsi. Se il chiamante ha bisogno di distinguere automaticamente i casi, possiamo aggiungere alla classe un dato strutturato, per esempio un codice o una categoria, invece di obbligarlo a interpretare la frase del messaggio.

La scelta fra estendere `Exception` e `RuntimeException` non è una graduatoria morale della gravità. Con `Exception` chiediamo ai chiamanti di considerare esplicitamente il percorso di errore nella compilazione. Con `RuntimeException` non imponiamo quel vincolo sintattico, utile per violazioni di precondizioni che indicano un difetto nel codice del chiamante. Nel nostro esempio l'input testuale può essere sbagliato senza che il programma sia scritto male; la versione checked rende visibile questa possibilità nella firma. In un'altra API si potrebbe scegliere un risultato che rappresenta successo o fallimento. Qualunque scelta facciamo, documentiamo il comportamento e lo proviamo con input validi, valori fuori intervallo e testo illeggibile.

Una pila che dichiara l'errore nella firma

La `Pila` del capitolo 8 segnala la condizione di pieno con `IllegalStateException`, che è unchecked. Possiamo costruirne una versione con una **eccezione personalizzata checked** capace di trasportare la capacità massima. Le due scelte sono possibili, ma i loro contratti per il chiamante sono diversi. Il programma `DemoPilaChecked` mette a confronto le due API dello stesso concetto e rende esplicita la differenza rispetto alla `Pila` già usata. Questa seconda pila ha capacità due, prova a inserire tre numeri e conserva la dimensione corretta dopo il rifiuto dell'ultimo.

```
public class DemoPilaChecked {
    static final class PilaPienaException extends Exception {
        private static final long serialVersionUID = 1L;
        private final int capacita;
        private final int valoreRifiutato;

        PilaPienaException(int capacita, int valoreRifiutato) {
            super("pila piena");
            this.capacita = capacita;
            this.valoreRifiutato = valoreRifiutato;
        }

        int capacita() {
            return capacita;
        }


        int valoreRifiutato() {
            return valoreRifiutato;
        }
    }

    static final class PilaControllata {
        private final int[] elementi;
        private int dimensione;

        PilaControllata(int capacita) {
            if (capacita <= 0) {
                throw new IllegalArgumentException("capacita non positiva");
            }
            elementi = new int[capacita];
        }


        void inserisci(int valore) throws PilaPienaException {
            if (dimensione == elementi.length) {
                throw new PilaPienaException(elementi.length, valore);
            }
            elementi[dimensione++] = valore;
        }

        int dimensione() {
            return dimensione;
        }
    }

    public static void main(String[] args) {
        PilaControllata pila = new PilaControllata(2);
        try {
            pila.inserisci(7);
```

```
            pila.inserisci(12);
            pila.inserisci(99);
        } catch (PilaPienaException errore) {
            System.out.println(errore.getMessage() + ": " + errore.capacita());
            System.out.println("rifiutato: " + errore.valoreRifiutato());
        }
        System.out.println("dimensione: " + pila.dimensione());
    }
}
```

L'output è `pila piena: 2`, `rifiutato: 99`, `dimensione: 2`. I campi dell'eccezione sono dati **strutturati**: il chiamante può leggere la capacità e il valore rifiutato senza analizzare la frase `pila piena`. Il metodo `inserisci` dichiara `throws PilaPienaException` e chi lo chiama deve gestire o propagare questa possibilità. Il controllo `dimensione == elementi.length` avviene prima della scrittura: così il terzo valore non viene perso sotto una falsa stampa di successo e non porta la dimensione a tre. L'esempio storico voleva insegnare proprio questo passaggio dal silenzio a un contratto osservabile; qui lo mostriamo senza far credere che ogni pila debba obbligatoriamente scegliere un'eccezione checked.

Il file contiene l'eccezione e la pila come classi annidate per rendere autosufficiente la prova. La parola `static` sulle due classi annidate dice che le loro istanze non hanno bisogno di un'istanza della classe `DemoPilaChecked` che le circonda. `serialVersionUID` serve a evitare un warning sulla serializzazione ereditata da `Exception`; non è il dato che rende checked l'eccezione. La proprietà checked deriva dal fatto che estende `Exception` senza estendere `RuntimeException`. Se togli il `catch` dal `main` senza aggiungere un `throws` adatto, il compilatore rifiuta il programma. È una prova negativa utile, distinta dalla normale esecuzione dell'esempio.

Per verificare

Compila il sorgente completo con `javac --release 25 -Xlint:all -d build DemoRisorse.java` ed esegui `java -cp build DemoRisorse`. Prima della prova, scrivi l'ordine delle quattro righe. Poi rimuovi soltanto il `throw` dentro il corpo del `try`: prevedi quale messaggio raggiungerà il `catch` e quante eccezioni soppresse troverai. L'attività del capitolo 12 include questa diagnosi e una lettura guidata dello stack trace.

13. Interfacce, classi astratte e polimorfismo

Quando la stessa domanda riguarda oggetti diversi

Nel capitolo sull'ereditarietà abbiamo potuto trattare un'automobile come un veicolo. La classe base forniva una parentela reale e una parte dell'implementazione. Adesso immaginiamo di voler chiedere un valore numerico tanto a un termometro quanto a un contatore di prestiti. Nessuno dei due è una variante dell'altro. Non serve inventare una superclasse che accomuni «cose che hanno un numero»: ci interessa soltanto poter rivolgere a entrambi la domanda `valore()`.

Un'**interfaccia** dichiara un tipo attraverso operazioni che le classi promettono di offrire. La classe scrive `implements` e fornisce il comportamento richiesto. Una variabile del tipo dell'interfaccia può riferirsi a oggetti di classi differenti che la implementano. Quando chiamiamo il metodo su quella variabile, viene eseguita l'implementazione dell'oggetto effettivo. Questo è il **polimorfismo per sottotipo** che useremo nel resto del libro: chi chiama dipende da un contratto comune e non deve conoscere ogni classe concreta.

> **Definizione – Contratto.** Per un'interfaccia, il contratto comprende firme dei metodi, significato delle operazioni e condizioni che i chiamanti possono aspettarsi. Il compilatore controlla le firme, ma non può dimostrare da solo che `valore()` rappresenti davvero la stessa idea in tutte le implementazioni. Nomi identici con significati incompatibili non costruiscono una buona astrazione.

Una pila di `int` può sembrare troppo specifica: se sostituiamo il tipo degli elementi con `Object`, accoglie qualunque oggetto. Questa scelta perde la verifica dei tipi: il chiamante deve eseguire cast e può scoprire un errore soltanto durante l'esecuzione. Qui separiamo i problemi. L'interfaccia serve a descrivere **che cosa si può fare**; i generics del capitolo 15 serviranno a descrivere **il tipo degli elementi** senza rinunciare al controllo del compilatore. La `Pila` del capitolo 8 rimane un esempio concreto di stato e invarianti, e non diventa magicamente generica perché implementa un'interfaccia.

Seguiamo le operazioni di una pila prima di scegliere come dichiararne il contratto. Con una capacità di due posti, la sequenza `push(7)`, `push(9)`, `pop()` lascia `7` nella pila e restituisce `9`: l'ultimo elemento entrato è il primo a uscire. Per un'altra pila che conservi titoli vorremmo la stessa regola, ma il metodo `pop()` dovrebbe restituire una `String`, non un `int`. Se dichiariamo `Object pop()` nell'interfaccia, entrambi gli oggetti possono promettere la stessa firma; però il chiamante che riceve `Object` non sa se il risultato sia un numero o un titolo. Dopo `Object dato = pila.pop()`, un cast a `String` compila anche quando la pila contiene un `Integer`; l'errore appare soltanto quando il cast viene eseguito. L'interfaccia ha reso uniforme il nome dell'operazione, ma ha perso una relazione essenziale fra l'argomento di `push` e il risultato di `pop`.

Non basta dire che tutti gli oggetti estendono `Object`. Quella parentela permette di conservare riferimenti eterogenei, non promette che qualunque oggetto estratto

sia del tipo desiderato dal chiamante. Né `pop()` dovrebbe restituire lo zero quando la pila è vuota: zero potrebbe essere un elemento legittimo e non distinguerebbe un risultato da un errore. Nel capitolo 8 abbiamo scelto un'eccezione per il caso vuoto; nel capitolo 15 useremo `Pila<T>` per mantenere insieme il tipo di ingresso e quello di uscita. Il passaggio storico dall'interfaccia con `Object` alla soluzione generica resta importante: mostra *quale problema* risolve il parametro di tipo, anziché presentare le parentesi angolari come una semplice novità grafica.

> **Concetto chiave – Due generalizzazioni diverse.** La variabile di tipo `Misurabile` consente a una funzione di lavorare con più classi che condividono un comportamento. Il parametro `T` di `Pila<T>` le consentirà di mantenere una relazione fra più posizioni del contratto. Possiamo usare interfacce e generics insieme, ma l'uno non sostituisce l'altro.

Il primo contratto

Il programma completo dichiara `Misurabile` e `Descrivibile` dentro una classe dimostrativa, così tutti i pezzi necessari restano in un unico file. In un'applicazione più grande, le interfacce pubbliche avrebbero file propri. La dichiarazione centrale è piccola:

```
interface Misurabile {
    int valore();

    default boolean supera(int soglia) {
        return valore() > soglia;
    }

    static int differenza(Misurabile primo, Misurabile secondo) {
        return Math.abs(primo.valore() - secondo.valore());
    }
}
```

Il metodo `valore()` non ha corpo: è implicitamente `public` e `abstract`. Una classe concreta che implementa `Misurabile` deve fornire un metodo `public int valore()`. Non basta avere per coincidenza un metodo con lo stesso nome: la classe deve dichiarare la relazione con `implements`. Un'interfaccia non possiede campi di istanza né costruttori, dunque non crea direttamente oggetti. Può dichiarare costanti, che sono implicitamente `public static final`, ma una costante pubblica va scelta perché appartiene davvero al contratto e non per nascondere stato condiviso.

`supera` è un metodo **default**: offre un comportamento comune basato sul metodo astratto. Se il valore effettivo cambia, `supera` usa sempre il valore restituito dall'oggetto corrente. `differenza` è invece **static**: si invoca come `Misurabile.differenza(a, b)` e non attraverso un'istanza. Il suo compito è un calcolo utile accanto al contratto; non viene ridefinito dalle classi che implementano l'interfaccia. La specifica Java 25 sulle interfacce descrive anche i metodi `private`, utili per condividere un dettaglio fra metodi default o statici senza esporlo ai chiamanti.

La figura 13.1 distingue la relazione fra tipo e implementazioni. Le frecce indicano che le due classi soddisfano lo stesso contratto, non che una erediti lo stato dell'altra.

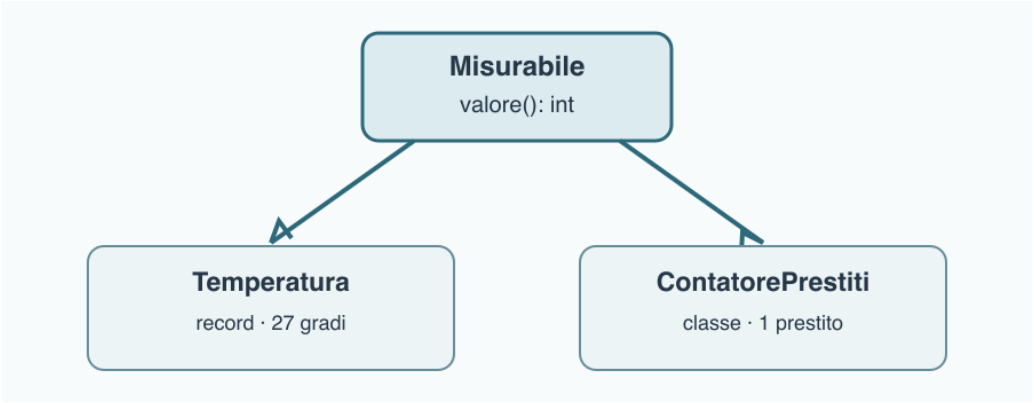


Figura 13.1 – Un'interfaccia permette di chiedere `valore()` *a oggetti non imparentati per classe.* `Temperatura` *e* `ContatorePrestiti` *possono anche implementare* `Descrivibile` *senza rinunciare alla loro diversa struttura interna.*

Nel programma, `Temperatura` è un record; il suo componente `valore` genera un accessore `valore()` che soddisfa l'interfaccia. Non abbiamo bisogno di scrivere un secondo metodo identico. `ContatorePrestiti` eredita invece `valore()` da una classe astratta. Le due strade sono diverse, ma il chiamante che riceve un `Misurabile` usa la medesima operazione. Eseguendo il programma leggiamo `27 gradi`, `true`, `1 prestiti` e `26`: la differenza usa i valori 27 e 1. Il programma mostra anche un limite dell'astrazione: sottrarre gradi e numero di prestiti è matematicamente possibile ma semanticamente discutibile. In un progetto reale il nome del contratto e i tipi dovrebbero impedire di confrontare grandezze che non hanno lo stesso significato. L'esempio serve proprio a far notare che la sola compatibilità sintattica non basta.

> **Nota bene – Il compilatore non misura il senso.** `Misurabile.differenza` accetta entrambi gli oggetti perché entrambi implementano l'interfaccia. Se il dominio consente soltanto differenze fra temperature, occorre un contratto più specifico, oppure una funzione che accetti soltanto `Temperatura`. Generalizzare troppo presto trasferisce errori di significato dal compilatore ai lettori e agli utenti.

Classe astratta o interfaccia?

La classe `Contatore` del programma conserva un campo privato e implementa due operazioni: `aumenta()` e `valore()`. È dichiarata `abstract` perché vogliamo usarla come base di specializzazioni e non crearla direttamente. `ContatorePrestiti` eredita sia lo stato sia il comportamento, aggiungendo la descrizione. Una classe Java può estendere una sola classe, astratta o concreta, ma può implementare più interfacce. Per questo il record `Temperatura` e `ContatorePrestiti` dichiarano entrambi `implements Misurabile, Descrivibile`.

Una classe astratta è adatta quando le sottoclassi condividono stato, invarianti o una sequenza di operazioni che la base può governare senza forzature. Un'interfaccia è adatta quando vogliamo esprimere una capacità utilizzabile da classi altrimenti indipendenti. Non è una regola che si applica contando soltanto i metodi: anche un'interfaccia può avere metodi default, e anche una classe astratta può non avere campi. La domanda più utile è quale relazione promettiamo a chi userà il tipo e quale evoluzione futura vogliamo rendere possibile.

Un solido, due responsabilità

Un solido rende concreta questa distinzione: per conoscerne il peso occorrevano un peso specifico e un volume; per conoscere il volume, invece, bisognava sapere quale forma concreta avessimo davanti. Seguiamo il ragionamento in un programma completo. `SolidoMisurabile` dichiara soltanto le domande `volume()` e `superficie()`. La classe astratta `Solido` conserva una densità, controlla che sia positiva e finita e offre `massa()`. `Cubo` conosce il lato e realizza le due misure geometriche. Se un'altra classe, senza essere sottoclasse di `Solido`, potesse calcolare volume e superficie, potrebbe comunque implementare `SolidoMisurabile`: è questa la libertà che il contratto concede.

Il programma stampa `8.0`, `24.0` e `24.0`. Per un lato lungo 2 unità, il volume del cubo è `2 × 2 × 2 = 8` unità cubiche; la superficie è la somma delle sei facce quadrate, ciascuna di area `2 × 2`, quindi `6 × 4 = 24` unità quadrate. Nell'esempio la densità è 3 unità di massa per unità di volume; la massa risulta `3 × 8 = 24` unità di massa. La geometria non è il tema del capitolo, ma le unità rendono evidente una questione di progetto: `volume()` e `superficie()` restituiscono entrambi `double`, eppure rappresentano grandezze diverse. Il tipo numerico da solo non impedisce di sommarle per errore. I nomi, la documentazione e, nei sistemi che lo richiedono, tipi di dominio distinti completano il contratto.

Nel `main`, il riferimento `SolidoMisurabile misurabile = cubo` espone soltanto le due misure geometriche. `misurabile.massa()` non compilerebbe: `massa()` appartiene alla classe astratta e quindi al tipo concreto `Cubo`, non all'interfaccia. L'oggetto a cui punta il riferimento è sempre il medesimo cubo; cambiare il tipo della variabile non cambia l'oggetto, ma cambia le operazioni che il compilatore permette di chiamare. `cubo.massa()` funziona e richiama `volume()` implementato da `Cubo`. Questa è una prova concreta del legame fra classe astratta e metodo ridefinito.

> **Nota bene – La formula ha ipotesi.** `densita × volume` calcola la massa solo se la densità è omogenea e le unità sono compatibili. Se la densità varia dentro il solido, quel metodo non rappresenta più il fenomeno; non lo salva né un'interfaccia né un `double` più preciso. Il contratto di un metodo comprende anche le ipotesi entro cui il risultato ha senso.

Per le interfacce possiamo parlare di «ereditarietà multipla di tipo». È vero che una classe può implementarne più di una e che un'interfaccia può estenderne più di una. Non significa che Java erediti liberamente lo stato di più classi. Questa distinzione evita di immaginare conflitti di campi o costruttori che appartengono ad altri modelli di linguaggio.

Quando due metodi default hanno lo stesso nome

Se due interfacce forniscono lo stesso metodo default e una classe le implementa entrambe, Java chiede alla classe di risolvere il conflitto. Nel secondo programma, `Primo` e `Secondo` dichiarano entrambi `nome()`. `Scelta` scrive un proprio `nome()` e richiama esplicitamente `Primo.super.nome()` e `Secondo.super.nome()`. L'output è `primo+secondo`.

Il compilatore non può scegliere sulla base dell'ordine in `implements`: quell'ordine non esprime una priorità semantica. La classe può scegliere una sola implementazione, combinarle o fornire un comportamento completamente nuovo, purché rispetti il contratto che promette. Se una classe base fornisce già un metodo d'istanza compatibile, quello prevale sul default dell'interfaccia. La scelta è parte del progetto della classe, non un dettaglio da affidare al caso.

Consideriamo un'interfaccia `Autore` che deve crescere senza rompere le implementazioni esistenti: dopo che `AutoreDelLibro` aveva già implementato due metodi, l'API pubblicata voleva aggiungerne un terzo senza obbligare tutti gli autori delle implementazioni a scriverlo. La nuova operazione deve avere senso per tutti gli autori. Se l'interfaccia promette `nome()` e `dataNascita()`, può aggiungere `default String etichetta() { return nome() + " (" + dataNascita() + ")"; }`. La vecchia classe continua ad avere un comportamento per `etichetta()` perché il metodo default si appoggia a due operazioni già richieste dal contratto. Una classe può comunque ridefinirlo, per esempio se la data non deve apparire in una ricevuta pubblica.

L'esempio storico aggiungeva invece `getPeso()` con il valore fisso `100.0`. Una classe compilerebbe anche in quel caso, ma il contratto direbbe una cosa falsa per molti oggetti. È il punto più interessante dell'esercizio: la compatibilità della firma non garantisce la correttezza del significato. Se non esiste un'implementazione generale sensata, conviene una nuova interfaccia oppure un servizio separato, invece di insegnare a tutte le classi un metodo che restituisce un valore inventato. Anche la compatibilità binaria di un'aggiunta va valutata nel contesto dell'intera gerarchia: metodi già presenti in altre interfacce o classi possono introdurre conflitti. Il `default` aiuta a evolvere un'API, ma non dispensa dal provare le implementazioni esistenti.

Un metodo statico dell'interfaccia risolve un'esigenza diversa. Se vogliamo verificare che due misure siano entrambe non negative, un metodo `static` può lavorare sui valori ricevuti e restare vicino al contratto. La chiamata si scrive `Misurabile.nomeDelMetodo(...);` le classi che implementano l'interfaccia non lo ridefiniscono con il normale overriding. Un metodo `private` può servire come dettaglio condiviso fra più default o statici: il chiamante esterno non lo vede. È quindi possibile scegliere fra una regola comune che ogni oggetto eredita, un'operazione di utilità legata al tipo e un dettaglio interno, senza confondere i tre livelli di visibilità e dispatch.

Interfacce estese, classi anonime e confini chiusi

Un'interfaccia può estenderne altre: `interface ArchivioLeggibile extends Descrivibile` aggiungerebbe nuove operazioni pur mantenendo il contratto di descrizione. Quando il tipo di un parametro è l'interfaccia più piccola che basta al metodo, chi chiama rimane libero di passare implementazioni diverse. Dichiarare ogni parametro con l'interfaccia più generale possibile, però, non è sempre una virtù: deve ancora rappresentare chiaramente ciò che il metodo richiede.

Un'interfaccia può anche essere dichiarata dentro una classe o un'altra interfaccia. Il nome qualificato, per esempio `Archivio.Lettore`, comunica che il contratto appartiene al contesto di `Archivio`; non crea automaticamente un oggetto esterno per ogni implementazione. La visibilità dipende da dove si trova la dichiarazione e dai suoi modificatori. È una scelta utile se il contratto è davvero legato a quell'API, meno utile quando altre parti del programma devono usarlo come concetto autonomo.

Prima delle lambda, per fornire una piccola implementazione locale si usava spesso una **classe anonima**: una classe senza nome dichiarata nell'espressione `new Interfaccia() { ... }`. È ancora utile quando occorre implementare più metodi o conservare un piccolo stato locale; per le interfacce con un solo metodo astratto, nel capitolo 16 vedremo spesso una lambda più leggibile. Anche un'enumerazione può implementare un'interfaccia se i suoi valori condividono un'operazione. Queste forme non cambiano il significato di `implements`.

Una **interfaccia sealed** limita quali classi o interfacce possono implementarla direttamente, con una clausola `permits`. È utile quando il dominio è intenzionalmente chiuso e vogliamo poter ragionare su tutti i casi, per esempio in uno `switch` esaustivo. Una delle implementazioni può essere un record, purché sia ammessa dalla dichiarazione e rispetti le regole della gerarchia sealed. Se nuovi casi devono poter essere introdotti liberamente da altri moduli, chiudere il tipo sarebbe invece un ostacolo. Il capitolo 9 ha già mostrato questa scelta per le classi; il principio è lo stesso.

Nel programma `DemoEsito.java`, un prestito produce un `Esito`: o è `Confermato` e porta un codice, oppure è `Rifiutato` e porta un motivo. I due record implementano un'interfaccia sealed. Uno `switch` su `Esito` tratta entrambi i casi e restituisce una descrizione. Il compilatore sa che l'elenco dei casi ammessi è chiuso, perciò non serve un ramo `default` che mascheri dimenticanze.

```
static String descrivi(Esito esito) {
    return switch (esito) {
        case Confermato confermato -> "prestito " + confermato.codice();
        case Rifiutato rifiutato -> "rifiuto: " + rifiutato.motivo();
    };
}
```

L'output è `prestito B-17` e `rifiuto: non disponibile`. Se aggiungiamo un terzo record consentito in `permits`, il compilatore ci chiede di aggiornare lo `switch`. Questo è un vantaggio quando il programma deve trattare ogni esito, non una ragione per chiudere tutte le interfacce. L'interfaccia `Misurabile` rimane volutamente aperta: una futura classe può offrire una misura senza chiedere di modificare la sua dichiarazione. La scelta fra aperto e chiuso nasce dalla stabilità del dominio, non dalla novità della sintassi.

Una classe senza nome e un enum con comportamento

Classi anonime ed enumerazioni possono entrambe realizzare un'interfaccia. Prima di scegliere la forma più breve, osserviamo che cosa viene dichiarato. Il programma `DemoFormeLocali.java` definisce `Operazione`, il cui metodo `applica` riceve due numeri e ne restituisce uno. `Calcolo` è un enum: ogni costante fornisce una propria implementazione del metodo. `SOMMA` restituisce la somma, `PRODOTTO` il prodotto. Nel `main` compare invece una classe anonima creata con `new Operazione() { ... }`; la sua implementazione sottrae uno sconto dal totale. I tre risultati stampati sono, nell'ordine, `6.0`, `8.0` e `15.0`.

La sequenza del primo calcolo merita di essere letta senza saltare passaggi: `applica(5, 3)` somma 5 e 3, poi sottrae lo sconto 2, ottenendo 6. Il metodo è nel corpo della classe anonima; l'espressione `new Operazione() { ... }` crea un'istanza di quella classe, che implementa l'interfaccia. Non stiamo creando direttamente un'istanza di un'interfaccia astratta. Non stiamo nemmeno usando un semplice «oggetto senza nome» nel senso generico di `new String(...)` passato senza assegnarlo a una variabile: qui è **la classe** a non avere un nome dichiarato nel sorgente.

La classe anonima può usare `sconto` perché quella variabile locale, dopo l'assegnazione iniziale, non viene più riassegnata: è *effectively final*. Se aggiungessimo `sconto = 3;` più avanti nel metodo, il compilatore rifiuterebbe la cattura, anche se la riassegnazione comparisse dopo la creazione dell'oggetto. La regola riguarda la variabile locale, non la possibilità astratta di modificare qualsiasi oggetto: un riferimento locale stabile può ancora puntare a un oggetto modificabile. Dire che `final` garantisce da sola purezza o assenza di effetti sarebbe sbagliato. Non c'è neppure una regola generale per cui aggiungere la parola `final` a ogni locale produce automaticamente ottimizzazioni impossibili con una variabile effectively final.

Per questa singola operazione, una lambda come `(primo, secondo) -> primo + secondo - sconto` sarebbe più breve. La useremo nel capitolo 16, dove chiariremo il tipo bersaglio. La classe anonima rimane istruttiva perché mostra esplicitamente il metodo implementato e perché può dichiarare uno stato e più metodi quando il contratto lo consente. Una lambda richiede un'interfaccia funzionale, mentre una classe anonima può implementare anche un'interfaccia con più metodi astratti. Se l'implementazione merita un nome, va riusata in più luoghi o richiede spiegazioni proprie, una classe dichiarata è spesso più leggibile di un corpo anonimo sempre più grande.

L'enum mostra un'altra decisione. `Calcolo.SOMMA` e `Calcolo.PRODOTTO` sono due valori di un insieme chiuso, non due sottoclassi pubbliche che il chiamante debba costruire. La chiamata `Calcolo.SOMMA.applica(5, 3)` esegue il corpo associato a quella costante. Potremmo aggiungere `SOTTRAZIONE`, ma dovremmo anche darle una semantica coerente con `Operazione`. Se ogni costante finisse per contenere molti dettagli di stato o una politica che cambia durante l'esecuzione, un enum diventerebbe una gabbia troppo rigida; classi separate che implementano il contratto descriverebbero meglio quel dominio.

Per osservare. Sostituisci `Operazione sottraiSconto = new Operazione() { ... };` con la lambda equivalente e controlla che le tre righe stampate rimangano uguali. Poi prova a riassegnare `sconto` prima di chiamare `applica`: la compilazione deve fallire, e la diagnostica riguarda la cattura della variabile locale. Infine aggiungi una costante `SOTTRAZIONE` all'enum: non basta scriverne il nome, perché il contratto richiede ancora il comportamento di `applica`.

Polimorfismi che non vanno confusi

Nel libro abbiamo incontrato più usi della parola *polimorfismo*. Il **sovraccarico** sceglie fra metodi con lo stesso nome e parametri diversi, in base alle firme disponibili al momento della compilazione. La **ridefinizione** permette a una sottoclasse o a una classe che implementa un'interfaccia di fornire il comportamento chiamato attraverso un tipo più generale. I **generics** permettono di scrivere un algoritmo o un contenitore parametrizzato su un tipo. L'idea di un comportamento comune resta utile, purché distinguiamo il controllo statico del tipo dalla scelta del metodo durante l'esecuzione.

Supponiamo di avere `Misurabile misura = new Temperatura(27);`. Il tipo dichiarato del riferimento è `Misurabile`, quindi possiamo chiamare soltanto i metodi che quel contratto rende disponibili. L'oggetto effettivo è `Temperatura`, e per `valore()` viene eseguito il comportamento fornito dal record. Non possiamo chiamare direttamente `descrizione()` attraverso `misura`, perché `Descrivibile` non è promesso dal tipo dichiarato, anche se quell'oggetto specifico lo implementa. Un cast o un pattern `instanceof` potrebbe recuperare l'altra capacità, ma se il metodo richiede sempre entrambe le capacità è più onesto progettare un tipo che le esprima entrambe.

La classe astratta presenta un altro confine utile. `Contatore` possiede un campo e il metodo `aumenta()`, ma non possiamo scrivere `new Contatore()`: la classe è incompleta come entità da esporre direttamente. Questo non significa che il suo costruttore, se definito, non venga eseguito quando nasce una sottoclasse. La parte di classe base viene comunque inizializzata prima che il costruttore della sottoclasse completi il lavoro, secondo le regole già viste nel capitolo 9. Chiamare dal costruttore astratto un metodo ridefinibile è rischioso: il metodo potrebbe osservare campi della sottoclasse non ancora inizializzati.

Approfondimento – Un metodo `private` nell'interfaccia. Un'interfaccia con più metodi default può ripetere una piccola regola interna. Un metodo `private` dell'interfaccia consente di darle un nome e riusarla senza aggiungerla al contratto pubblico. Non è un metodo astratto da implementare nelle classi concrete. Questo è un esempio di come l'interfaccia moderna possa contenere comportamento condiviso, pur non diventando una classe con campi di istanza.

Tre prove sullo stesso riferimento

Per vedere davvero il polimorfismo non basta dichiarare due classi con `implements`. Torniamo a `Misurabile misura = new Temperatura(27)` e facciamo tre prove mentali. Prima chiamiamo `misura.valore()`: la compilazione riesce perché `valore()` è nel contratto di `Misurabile`, e la chiamata restituisce 27 dall'oggetto `Temperatura`. Poi assegniamo `misura = new ContatorePrestiti()` e chiamiamo ancora `misura.valore()`: il sorgente della chiamata è identico, ma ora l'implementazione eseguita appartiene al contatore. Infine tentiamo `misura.descrizione()`: la chiamata non compila, anche se entrambe le classi concrete hanno quella capacità, perché il tipo dichiarato della variabile non la promette.

Potremmo risolvere l'ultimo caso con un cast, ma prima dobbiamo chiedere che cosa sa il chiamante. Se *ogni* valore ammesso in quel punto deve essere anche descrivibile, un'interfaccia più specifica che estende entrambe le capacità rende il requisito esplicito. Se soltanto alcuni lo sono, un controllo `instanceof Descrivibile descrivibile` consente di trattare il caso senza assumere che tutti gli oggetti lo siano. La distinzione non serve a evitare un messaggio del compilatore: serve a non trasformare un'ipotesi non dichiarata in un errore a esecuzione avviata.

L’idea che oggetti diversi possano presentare operazioni comuni richiede un contratto riconoscibile. In Java quella forma è un tipo dichiarato, quindi il compilatore ne controlla la relazione. Non basta che una classe abbia casualmente un metodo chiamato `valore`: senza `implements Misurabile` non può essere usata come `Misurabile`. Questa caratteristica rende il contratto verificabile e permette alla documentazione di descrivere un significato comune. Il prezzo è una dichiarazione esplicita, che in una libreria ben progettata diventa anche una guida per chi legge.

Progettare un contratto che sopravvive alle implementazioni

Un'interfaccia utile non nasce copiando tutti i metodi di una classe esistente. Nasce da una domanda che più chiamanti devono poter porre senza conoscere il dettaglio della risposta. Immaginiamo un metodo che stampa una ricevuta di prestito. Se gli basta ottenere una descrizione, ricevere `Descrivibile` è più chiaro che ricevere un oggetto `ContatorePrestiti` soltanto perché la prima implementazione disponibile è quella. Ma se il metodo deve anche cambiare il numero dei prestiti, `Descrivibile` non promette abbastanza: occorre un contratto che esponga esplicitamente l'operazione ammessa.

La **segregazione** dei contratti significa evitare che un chiamante dipenda da operazioni che non usa. Non vuol dire creare un'interfaccia per ogni singolo metodo a prescindere dal contesto. `Misurabile` e `Descrivibile` sono separati perché chiedono cose diverse: una misura e una rappresentazione testuale. Una classe può implementarle entrambe, ma il metodo che richiede soltanto la misura non riceve per questo anche il diritto di descrivere o modificare l'oggetto. Questo restringe le aspettative e rende più facile sostituire un'implementazione.

Una sostituzione corretta richiede anche comportamento coerente. Se `Misurabile.valore()` promette una misura stabile durante una singola operazione,

un'implementazione che cambia casualmente il valore a ogni lettura rompe quella promessa pur compilando. Se `Descrivibile.descrizione()` è destinata a una ricevuta per l'utente, un'implementazione che restituisce un codice interno incomprensibile rispetta la firma ma non l'uso. La documentazione del contratto deve quindi spiegare significato, eventuali valori ammessi, errori e stabilità del risultato, non solo mostrare `int valore();`.

Anche l'evoluzione merita attenzione. Aggiungere un nuovo metodo astratto a un'interfaccia pubblica può obbligare tutte le classi che la implementano a cambiare. Un metodo default può fornire un comportamento iniziale compatibile quando esiste una regola generale sensata, come `supera` che si basa su `valore()`. Non va usato per mascherare una domanda a cui alcune implementazioni non possono rispondere correttamente. Se il comportamento nuovo è opzionale o appartiene a una capacità diversa, un'altra interfaccia può essere un modello più fedele.

Infine, un'interfaccia non rende un oggetto immutabile né thread-safe. `ContatorePrestiti` può cambiare il proprio numero; due thread che lo aggiornano richiederebbero regole di concorrenza ulteriori. Il contratto può dichiarare queste garanzie se servono, ma la parola `interface` non le crea. Questa cautela prepara i capitoli sulle collezioni: `List` dice quali operazioni esistono, mentre la scelta dell'implementazione e il contratto dell'API stabiliscono proprietà pratiche come mutabilità, ordine e uso concorrente.

Per verificare

Compila i tre programmi con `javac --release 25 -Xlint:all -d build DemoInterfacce.java DemoConflittoDefault.java DemoEsito.java`. Prova poi a togliere `nome()` dalla classe `Scelta`: il compilatore deve rifiutare il conflitto dei default. Nel primo programma, cambia il parametro di `supera` da 25 a 30 e spiega perché cambia soltanto la riga booleana. Nel terzo, aggiungi un nuovo esito consentito e osserva la diagnostica dello `switch` prima di aggiornarlo. L'attività applicativa dei capitoli 13–18 riprenderà questi contratti per introdurre un tipo generico senza cast.

14. Annotazioni: metadati e controlli

Un'informazione che il programma può leggere

Nel capitolo precedente abbiamo usato `@Override` davanti a un metodo. Quelle poche lettere non costituiscono il corpo del metodo: dichiarano al compilatore l'intenzione di ridefinire un metodo ereditato. Se abbiamo sbagliato la firma, il compilatore ci ferma. Senza l'annotazione, un metodo con lo stesso nome ma parametri diversi potrebbe essere soltanto un sovraccarico, e il nostro errore passerebbe inosservato. Per esempio, se il metodo di un veicolo accetta `int` e la sottoclasse scrive per sbaglio `double`, `@Override` fa emergere subito l'errore.

Un'**annotazione** associa metadati a una dichiarazione o, in certi casi, a un uso di un tipo. I metadati sono dati *sul programma*: non sono la velocità del veicolo o il titolo del libro, ma informazioni che un compilatore, una libreria o uno strumento possono interpretare. La presenza di un'annotazione non esegue automaticamente un'azione. `@Override` ha un effetto perché il linguaggio glielo assegna; un'annotazione inventata da noi acquista un effetto solo se un componente la legge. Questo distingue il meccanismo Java dai comportamenti aggiunti da framework esterni.

> **Definizione – Annotazione e processore.** L'annotazione è il dato scritto nel sorgente. Un *annotation processor* è un programma richiamato durante la compilazione per esaminare gli elementi del programma e, se previsto, produrre diagnostiche o nuovi file. La reflection, invece, può leggere a esecuzione avviata le annotazioni mantenute fino al runtime. Sono tre momenti distinti.

Dichiarare un'intenzione non significa farla accadere

La programmazione dichiarativa ci aiuta a capire perché tanti progetti usano annotazioni. Quando scriviamo una query SQL come `SELECT titolo FROM libri WHERE disponibile = true`, dichiariamo il risultato cercato; il motore decide come attraversare indici e pagine di dati. Java rimane un linguaggio in cui normalmente descriviamo esplicitamente i passi, ma alcune sue caratteristiche permettono di aggiungere intenzioni che altri componenti interpreteranno. Scrivere `@Etichetta("catalogo")` sopra una classe è più vicino a «questo elemento appartiene al catalogo» che a «esegui ora queste istruzioni».

Il paragone con SQL ha un limite importante. Una query ha un motore con una semantica definita; una nostra annotazione non possiede da sola un motore. Possiamo annotare una classe `@DaSalvare` e scoprire che non viene salvato nulla, se non abbiamo scritto o installato il componente che legge quel metadato. Prima di adottare uno stile dichiarativo chiediamoci quindi tre cose: chi interpreta la dichiarazione, quando lo fa e quale errore otteniamo se la dichiarazione è incoerente? Per `@Override` la risposta è il compilatore. Per `@Etichetta` nel programma che segue è il nostro codice di reflection. Per alcune annotazioni di

framework è una libreria esterna, magari avviata quando l'applicazione costruisce i suoi componenti.

Questa distinzione evita di attribuire poteri misteriosi alla chiocciola. Le annotazioni sono state introdotte in Java 5 per associare informazioni strutturate agli elementi del programma. Possono aiutare strumenti, controlli di compilazione e librerie, e in certi casi riducono file di configurazione separati. Ma spostare una regola da un file XML a un'annotazione non la rende automaticamente semplice: la regola deve ancora avere un significato, un lettore e una prova. In tutto il capitolo useremo la stessa domanda come bussola: **se tolgo questa annotazione, quale componente cambia comportamento e come posso vederlo?**

Definire un'annotazione piccola

Nel sorgente `Etichetta.java` dichiariamo un'annotazione che attribuisce parole a una classe. Il suo unico elemento si chiama `value`, perciò `@Etichetta("catalogo")` è la forma breve di `@Etichetta(value = "catalogo")`.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Repeatable(Etichette.class)
public @interface Etichetta {
    String value();
}
```

La sintassi `@interface` dichiara un tipo di annotazione. `String value();` non è un normale metodo da implementare in una classe applicativa: definisce l'elemento che chi usa l'annotazione deve valorizzare. Possiamo assegnargli un valore predefinito con `default`, per esempio `String value() default "generale";`; in tal caso l'uso senza argomenti diventa valido. I tipi ammessi per gli elementi comprendono primitivi, `String`, `Class`, enum, altri tipi di annotazione e array di questi tipi. Un oggetto arbitrario, come `List<String>`, non è ammesso. La specifica Java 25 fissa questi vincoli.

Le tre annotazioni davanti alla dichiarazione sono **meta-annotazioni**: annotazioni che descrivono un tipo di annotazione. `@Target(ElementType.TYPE)` limita l'uso a classi, interfacce, record ed enum. `@Retention(RetentionPolicy.RUNTIME)` permette alla reflection di ritrovare i valori durante l'esecuzione. `@Repeatable(Etichette.class)` ammette più `@Etichetta` sullo stesso elemento; `Etichette` è l'annotazione contenitore richiesta per rappresentarle. Il file completo la dichiara con un elemento `Etichetta[] value()` e con target e retention coerenti.

Le politiche di conservazione rispondono a domande diverse. Con `SOURCE`, l'annotazione serve al sorgente e non viene conservata nel file `.class`; con `CLASS`, viene conservata nel file compilato ma non resa disponibile dalla normale reflection a runtime; con `RUNTIME`, può essere letta durante l'esecuzione. Se omettiamo `@Retention`, il valore predefinito è `CLASS`. La figura 14.1 collega le tre politiche ai momenti in cui le informazioni sono disponibili.

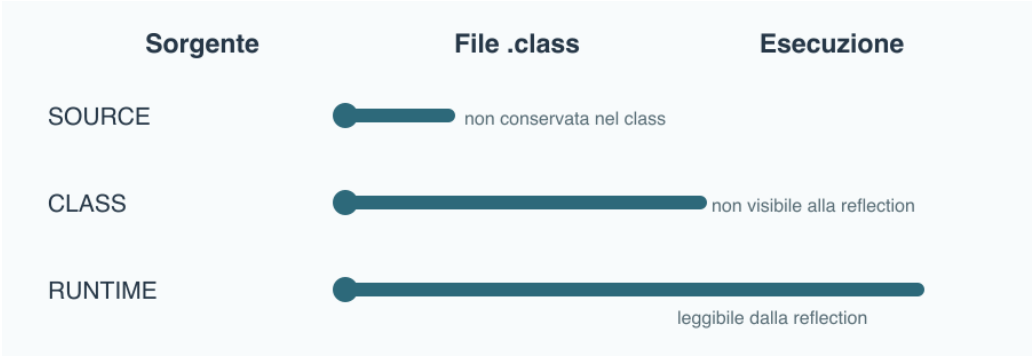


Figura 14.1 – `SOURCE`, `CLASS` e `RUNTIME` indicano fino a quale fase arriva un'annotazione. La politica non decide da sola quale lavoro verrà compiuto su quei metadati.

Nel programma `DemoAnnotazioni.java` scriviamo due etichette su una classe e le leggiamo con `getAnnotationsByType(Etichetta.class)`. Il ciclo stampa `catalogo` e `didattica`. L'API restituisce entrambe anche se la ripetizione viene rappresentata tramite il contenitore. Se cambiassimo la retention in `CLASS`, questo esempio non troverebbe le etichette a runtime: il codice compilerebbe, ma il ciclo non stamperebbe i due valori. È un esperimento utile per vedere che la disponibilità dei metadati fa parte del contratto.

Annotazioni note, responsabilità diverse

`@Override` chiede un controllo sulla ridefinizione. `@FunctionalInterface` chiede che un'interfaccia abbia un solo metodo astratto utilizzabile come bersaglio di una lambda; la vedremo nel capitolo 16. `@Deprecated(since = "25", forRemoval = false)` comunica che un'API non è più raccomandata e può registrare da quando vale tale indicazione; `forRemoval` segnala un'intenzione di rimozione, non una data automatica. `@SuppressWarnings` limita una diagnostica del compilatore in un ambito specifico: va usata dopo aver compreso e motivato il warning, non per nascondere problemi.

`@SafeVarargs` riguarda metodi e costruttori con argomenti variabili generici o di tipo parametrizzato. È una promessa di sicurezza fatta da chi implementa il metodo, non una cura applicata dal compilatore al corpo. Nel capitolo 15 vedremo perché array e generics possono produrre *heap pollution* e quando tale promessa è giustificata. Mettere tutte queste annotazioni sotto l'etichetta «documentazione» nasconderebbe differenze importanti: alcune attivano controlli del linguaggio, altre comunicano informazioni alle API o agli strumenti.

Un'annotazione di **type use** compare su un uso del tipo, per esempio `List<@NonNull String>` se esiste un'annotazione adatta e uno strumento che ne interpreta il significato. La sola piattaforma Java non fa scattare una verifica universale di non nullità per un'annotazione personalizzata `@NonNull`. Occorre distinguere il punto sintattico in cui un metadato può essere scritto dal controllo concreto che qualcuno esegue. Lo stesso vale per annotazioni di librerie che generano codice o configurano un framework: senza quello strumento non producono l'effetto promesso.

Un processore eseguito durante la compilazione

Il processore `EtichettaProcessor.java` estende `AbstractProcessor`. Dichiara i nomi delle annotazioni che osserva e la versione del linguaggio supportata, poi legge gli elementi annotati nel metodo `process`. Quando incontra `DemoAnnotazioni`, scrive una nota del compilatore: `etichetta presente su DemoAnnotazioni`. Il messaggio compare durante `javac`, prima che `main` venga avviato. Il contratto di `Processor` in Java 25 spiega che l'elaborazione può procedere per più *round*, o tornate: un processore può generare sorgenti che entrano in una tornata successiva.

Per riprodurre l'esempio, dalla cartella dei tre sorgenti esegui prima `javac --release 25 -Xlint:all -d build Etichetta.java DemoAnnotazioni.java EtichettaProcessor.java`. Poi chiedi esplicitamente una compilazione con il processore: `javac --release 25 -cp build -processorpath build -processor EtichettaProcessor -proc:only DemoAnnotazioni.java`. L'opzione `-proc:only` fa eseguire l'elaborazione senza generare una nuova classe dal sorgente esaminato. Non assumiamo che `javac` scopra ed esegua automaticamente il processore dalla cartella: il comando dichiara il processore e il suo percorso. Questa distinzione è importante anche per la sicurezza della build e per la sua riproducibilità.

Il processore minimo si limita a produrre una nota. Un processore reale potrebbe generare sorgenti o segnalare un errore usando l'API `Messager`, ma non modifica arbitrariamente il sorgente che gli viene consegnato. Per costruirne uno robusto occorrono anche regole sui nomi dei tipi, sui round e sui file generati. Qui basta aver visto dove agisce e come si verifica che abbia agito.

Che cosa legge davvero il processore

Il parametro `annotations` del metodo `process` contiene i tipi di annotazione rilevanti per quella tornata; `RoundEnvironment` permette di chiedere quali elementi del programma ne sono marcati. Un **elemento** del modello di compilazione può essere una classe, un metodo, un campo o un altro costrutto dichiarato. Non è un'istanza Java creata da `new` durante l'esecuzione dell'applicazione. Nel nostro processore leggiamo `element.getSimpleName()` e otteniamo `DemoAnnotazioni`, il nome della classe dichiarata. Questa distinzione spiega perché il processore può operare anche quando il programma non ha ancora un `main` eseguibile.

`round.processingOver()` segnala che l'elaborazione è arrivata alla tornata conclusiva. Nel processore evitiamo di emettere la nota in quella fase. Un processore che genera un nuovo sorgente può causare un'altra tornata in cui quel sorgente viene a sua volta esaminato. Se il codice generato porta altre annotazioni riconosciute, il lavoro può proseguire. La progettazione deve evitare di generare ripetutamente lo stesso file o di basarsi su un ordine casuale di visita degli elementi.

Il nostro esempio usa `@SupportedAnnotationTypes` e `@SupportedSourceVersion` per dichiarare al compilatore che cosa tratta. Il valore `SourceVersion.RELEASE_25` rende esplicita la versione del linguaggio usata per questa prova. In una libreria destinata a più versioni, la scelta andrebbe verificata contro i JDK supportati, non copiata meccanicamente. Il `return false` del metodo `process` indica che il processore non rivendica esclusivamente quelle annotazioni: altri processori

possono ancora riceverle secondo il protocollo. Non cambia la stampa delle due etichette a runtime.

> **Approfondimento – Generare senza modificare.** Un processor può chiedere a `Filer` di creare un nuovo sorgente o una risorsa. Il file originale annotato rimane un input; non viene riscritto come se il processore fosse un editor di testo. Un progetto può poi compilare insieme sorgenti scritti dall'autore e sorgenti generati. Questo richiede una build che controlli percorsi di output, provenienza del processor e riproducibilità, perché il codice eseguito durante la compilazione è parte della catena di produzione del programma.

Quando il processore diventa una dipendenza della build

Nel piccolo esperimento abbiamo indicato `-processor EtichettaProcessor` direttamente. In una libreria, il processore può essere distribuito in un JAR e dichiarato come servizio tramite `META-INF/services/javax.annotation.processing.Processor`, con il nome completo della classe che lo implementa. È così che uno strumento di build può scoprire un processor sul percorso configurato. Ma «scoprire» non significa affidarsi al caso: per una build ripetibile dichiariamo quale JAR fornisce il processor, quale versione usiamo e in quale fase deve essere eseguito. La configurazione del compiler plugin di Maven, per esempio, appartiene al progetto tanto quanto la versione della libreria che useremo a runtime.

Il processore è codice che il compilatore **esegue** mentre costruisce l'applicazione. Se scarichiamo una dipendenza non verificata e la lasciamo eseguire durante la build, le conseguenze non sono limitate a un'annotazione sbagliata nel `.class`: quel codice può leggere file e generare output. Il capitolo 29 affronterà la provenienza delle dipendenze e la ripetibilità della build; qui il collegamento è concreto. Una diagnostica attesa come `etichetta presente su DemoAnnotazioni` è una prova che il processor è stato invocato, mentre la sola presenza di `@Etichetta` nel sorgente non lo dimostra.

Un processor che produce una classe, per esempio `CatalogoComandiGenerato`, deve poter convivere con più tornate di compilazione. Nella prima vede `@Comando` e genera il sorgente; nella successiva il compilatore analizza anche quella nuova classe. Se a ogni tornata il processor tenta di ricreare lo stesso file, la build fallisce o diventa dipendente da dettagli dell'ordine di visita. Per questo distinguiamo la fase in cui raccogliamo informazioni dalla fase in cui emettiamo un risultato, controlliamo `processingOver()` e proviamo almeno una compilazione pulita. Quando una build incrementale produce un esito diverso da `clean verify`, è un segnale da investigare, non una ragione per considerare superflua la build pulita.

Questo chiarisce una distinzione terminologica importante. Un annotation processor lavora sul modello del programma esposto dal compilatore e può creare nuovi file; non è, in generale, un semplice preprocessore testuale che sostituisce parole prima di `javac`. Alcuni strumenti esterni usano meccanismi ulteriori per trasformare ciò che il compilatore vede. Quando leggiamo la documentazione di uno strumento, dobbiamo distinguere il contratto standard dell'API `javax.annotation.processing` dalle capacità specifiche di quella libreria. È la

stessa disciplina che abbiamo applicato a `@Override` e alla reflection: individuare l'attore prima di attribuire un effetto all'annotazione.

Una scelta concreta per `@Target` e `@Retention`

Supponiamo di voler annotare i metodi che espongono un comando all'utente. Scrivere `@Target(ElementType.METHOD)` impedisce di mettere l'annotazione per sbaglio su una classe. Se il programma deve scoprire i comandi all'avvio tramite reflection, serve `RetentionPolicy.RUNTIME`. Se invece un processor genera una tabella dei comandi durante la compilazione e l'applicazione usa soltanto quella tabella, la retention runtime potrebbe essere superflua. La scelta deriva dal lettore previsto del metadato, non dalla sensazione che `RUNTIME` sia sempre «più completa».

La reflection usa l'interfaccia `AnnotatedElement`, implementata da rappresentazioni come `Class` e `Method`. I metodi per leggere una singola annotazione e quelli per leggere annotazioni ripetibili hanno contratti distinti; `getAnnotationsByType` è la forma adatta al nostro esempio. Anche qui conta l'ereditarietà: una meta-annotazione come `@Inherited` ha regole precise per certe annotazioni su classi e non rende ereditabile ogni annotazione su metodi o interfacce. Invece di confidare nel nome, si verifica il contratto della meta-annotazione scelta.

Un'annotazione può documentare una decisione, chiedere un controllo o fornire configurazione. È utile quando il metadato appartiene stabilmente all'elemento dichiarato e può essere letto da uno strumento definito. Se per capire il flusso di una funzione occorre inseguire dieci annotazioni di framework e regole esterne, la sintassi compatta non ha reso il programma automaticamente più semplice. Il lettore deve sapere quale parte è Java, quale è la libreria e quale è la configurazione della build.

> **Nota bene – Reflection non è compilazione.** `DemoAnnotazioni.class.getAnnotationsByType(...)` legge metadati durante l'esecuzione. `EtichettaProcessor` osserva il modello del programma durante la compilazione. Una retention `RUNTIME` è necessaria per il primo uso, ma non è una condizione generale per elaborare un'annotazione nel sorgente con un processore.

Il bersaglio fa parte del contratto

`ElementType` descrive i luoghi in cui possiamo applicare un’annotazione. Non serve impararla a memoria: serve riconoscere che il metadato di una classe e quello di un *uso di tipo* rispondono a domande diverse. Se `@Responsabile` descrive chi mantiene un metodo, `METHOD` è un bersaglio sensato. Se `@Unità` descrive un numero usato come metri o secondi, potremmo voler annotare l'uso del tipo, non la dichiarazione di una classe: quello è il caso di `TYPE_USE`. Il secondo esempio non produce una verifica delle unità per magia; richiede un analizzatore che ne conosca le regole.

Bersaglio	Quale elemento marca	Domanda da porsi
`TYPE`, `ANNOTATION_TYPE`	Classi, interfacce, record, enum o dichiarazione di un'annotazione	Il metadato descrive il tipo intero o un altro tipo di annotazione?

Bersaglio	Quale elemento marca	Domanda da porsi
`FIELD`, `METHOD`, `CONSTRUCTOR`	Campo, metodo o costruttore dichiarato	Il controllo riguarda quell'operazione o l'intera classe?
`PARAMETER`, `LOCAL_VARIABLE`	Parametro formale o variabile locale	Chi può ancora osservare l'informazione dopo la compilazione?
`PACKAGE`, `MODULE`	Dichiarazioni di package o modulo	La configurazione appartiene a tutto il confine?
`TYPE_PARAMETER`, `TYPE_USE`	Parametro generico o uso di un tipo	Stiamo parlando di `T` dichiarato o di un punto in cui il tipo è usato?

Per più bersagli scriviamo un array, per esempio `@Target({ElementType.TYPE, ElementType.METHOD})`. Senza `@Target`, un'annotazione è applicabile ai contesti di dichiarazione ammessi dalle regole predefinite, ma non acquista automaticamente tutti i contesti di *type use*. Definire il bersaglio restringe l'errore possibile: se `@Comando` vale solo sui metodi, il compilatore rifiuta un uso distratto sulla classe. La tabella non sostituisce però la progettazione. Un'annotazione applicabile dappertutto è raramente utile se il suo lettore sa interpretare solo metodi.

Il caso delle variabili locali è particolarmente istruttivo. Un'annotazione sulla **dichiarazione** di una variabile locale non diventa interrogabile con la normale reflection a esecuzione avviata, anche se il tipo di annotazione dichiara retention `RUNTIME`. Quella variabile non è un membro che possiamo cercare come un campo. Diverso è annotare un **uso del tipo** della variabile: le regole di conservazione nel file `.class` sono specificate separatamente. Confondere queste due posizioni porta a esperimenti in cui il sorgente sembra annotato correttamente ma il programma non trova nulla. La specifica Java 25 distingue i due casi; chi progetta l'annotazione deve sapere quale informazione vuole recuperare e con quale API.

Osservare retention ed ereditarietà, senza indovinare

Nel programma `DemoPolitiche.java` mettiamo `@SoloSorgente`, `@NelClass` e `@ARunTime` sulla stessa classe. La compilazione accetta tutte e tre. La reflection ne trova una sola: quella con retention `RUNTIME`. Non stiamo dicendo che le altre due fossero inutili; un analizzatore del sorgente può leggere la prima, e uno strumento che ispeziona il bytecode può leggere la seconda. Stiamo dicendo che *questa prova* usa la normale API reflection e quindi osserva soltanto ciò che il suo contratto consente.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public final class DemoPolitiche {
    @Retention(RetentionPolicy.SOURCE)
    @Target(ElementType.TYPE)
    @interface SoloSorgente { }

    @Retention(RetentionPolicy.CLASS)
    @Target(ElementType.TYPE)
    @interface NelClass { }

    @Retention(RetentionPolicy.RUNTIME)


    @Target(ElementType.TYPE)
    @interface ARunTime { }

    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    @interface NonEreditata { }

    @Inherited
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    @interface Ereditata { }

    @SoloSorgente
    @NelClass
    @ARunTime
    static class Documento { }


    @NonEreditata
    @Ereditata
    static class Base { }

    static class Derivata extends Base { }

    private DemoPolitiche() { }

    public static void main(String[] args) {
        System.out.println("Documento a runtime: "
                + Documento.class.getAnnotations().length);
        System.out.println("Base dichiarate: "
                + Base.class.getDeclaredAnnotations().length);
        System.out.println("Derivata dichiarate: "
                + Derivata.class.getDeclaredAnnotations().length);


        System.out.println("Derivata visibili: "
                + Derivata.class.getAnnotations().length);
    }
}
```

L'output è `Documento a runtime: 1`, `Base dichiarate: 2`, `Derivata dichiarate: 0`, `Derivata visibili: 1`. La differenza fra gli ultimi due numeri spiega `@Inherited`

meglio del suo nome. `getDeclaredAnnotations()` guarda soltanto ciò che è dichiarato direttamente su `Derivata`, quindi trova zero. `getAnnotations()` considera anche l'annotazione ereditabile della classe base e trova `@Ereditata`. `@NonEreditata`, pur avendo retention runtime, non passa alla sottoclasse. `@Inherited` opera sulle annotazioni delle classi secondo regole precise; non fa ereditare annotazioni di metodi né trasforma un'annotazione su un'interfaccia in un attributo universale delle implementazioni.

Possiamo compilare e lanciare il file con `javac --release 25 -Xlint:all -d build DemoPolitiche.java` e `java -cp build DemoPolitiche`. Se cambiamo `@Ereditata` da `RUNTIME` a `CLASS`, l'ultimo numero non sarà più uno: il metadato non è più esposto alla reflection. Se togliamo `@Inherited` mantenendo `RUNTIME`, `Base` continua a mostrare due annotazioni dichiarate ma `Derivata` non ne vede alcuna. Sono due modifiche diverse e la prova ci permette di attribuire il risultato alla proprietà giusta.

Riprendere l'errore di `@Override`

L'esempio storico del veicolo merita di essere svolto fino alla diagnostica. Immaginiamo che `Veicolo` dichiari `void accelera(int incremento)` e che `Automobile` scriva `void accelera(double incremento)`. Le due firme differiscono nel parametro: la seconda dichiarazione crea un **sovraccarico**, non ridefinisce il metodo ricevuto dalla classe base. Una chiamata attraverso una variabile `Veicolo` continuerà quindi a usare il metodo che accetta `int`. Se il programmatore voleva cambiare il comportamento dell'auto per quella chiamata, ha commesso un errore che il codice senza annotazione potrebbe nascondere.

Scrivendo `@Override` davanti al metodo con `double`, il compilatore segnala che non sta ridefinendo né implementando il metodo atteso. La correzione non consiste nel togliere `@Override` per far sparire il messaggio: consiste nel riallineare la firma a `int` se quella era l'intenzione. Se invece serve davvero un metodo aggiuntivo per valori `double`, lo si tiene come sovraccarico e lo si documenta senza promettere una ridefinizione. Il caso mostra bene perché un'annotazione utile non è una decorazione: rende controllabile un'intenzione che la sola sintassi del metodo non esprime.

Dalla forma dell'annotazione al suo lettore

Le annotazioni possono non avere elementi, contenere un solo valore o dichiararne più di uno. La distinzione diventa utile quando la colleghiamo al codice che le interpreta. Un'annotazione senza elementi, spesso detta *marker*, comunica la propria presenza: `@Controllato` non porta un argomento, e lo strumento interessato chiede se il segno sia presente. Un tipo con il solo elemento `String value();` permette la forma breve `@Etichetta("catalogo")`. Con due elementi, per esempio `String nome(); int priorita() default 0;`, chi la usa deve scrivere il nome dell'elemento obbligatorio: `@Comando(nome = "salva")`; la priorità assente vale zero. Non esiste un oggetto arbitrario costruito con `new` da conservare come valore dell'annotazione.

Il valore di un elemento deve appartenere ai tipi ammessi dalla specifica ed essere esprimibile secondo le regole delle annotazioni. `null` non è un valore consentito. Per un `String` non possiamo usare una concatenazione che dipende da un input

letto durante l'esecuzione: l'informazione deve poter essere registrata nel programma compilato. L'array è ammesso come tipo di elemento, ma non diventa per questo una collezione `List<String>`. Questa limitazione non è un difetto casuale; permette agli strumenti di leggere metadati con una rappresentazione definita, anche prima che le classi applicative vengano istanziate.

La domanda decisiva rimane: **chi legge il metadato?** Se un processore produce un file durante la compilazione, prova il risultato compilando con quel processore e controllando il file o la diagnostica prodotti. Se una libreria legge la reflection a runtime, prova avviando il programma e scegliendo `RUNTIME`. Se l'annotazione serve soltanto a un controllo di sorgente, conservarla fino al runtime può non aggiungere alcun valore. Due annotazioni graficamente identiche possono dunque vivere in fasi diverse e avere effetti diversi perché diversi sono i loro lettori.

Una dichiarazione può comparire più volte

Nell'esempio storico del libro un servizio doveva inviare un messaggio a più orari. Un'unica annotazione `@Scheduler(ora = 12, ...)` raccontava un solo appuntamento; ripeterla sulla stessa classe era inizialmente un errore di compilazione. Il bisogno è rimasto attuale: un elemento può avere più etichette, più regole o più momenti di attivazione. `@Repeatable` consente la ripetizione, ma richiede un'annotazione contenitore con un elemento `value()` che restituisca l'array delle annotazioni ripetute. In pratica il contenitore offre una rappresentazione in cui i valori possono essere raccolti.

Il programma `DemoPromemoria.java` usa due orari. Abbiamo scelto `TYPE` per l'annotazione e per il suo contenitore perché il metadato descrive la classe `Notifiche`. Se volessimo pianificare singoli metodi, cambieremmo entrambi i target a `METHOD` e il codice che cerca le annotazioni dovrebbe esaminare quei metodi. La retention è `RUNTIME` perché il programma legge gli orari durante l'esecuzione. Senza un componente che pianifichi e invii davvero il messaggio, però, questo esempio **stampa** gli orari: non è uno scheduler funzionante.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public final class DemoPromemoria {
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    @Repeatable(PromemoriaMultipli.class)
    @interface Promemoria {
        int ora();
        int minuto();
    }

    @Retention(RetentionPolicy.RUNTIME)
```

```
    @Target(ElementType.TYPE)
    @interface PromemoriaMultipli {
        Promemoria[] value();
    }

    @Promemoria(ora = 9, minuto = 30)
    @Promemoria(ora = 17, minuto = 0)
    static class Notifiche { }

    private DemoPromemoria() { }

    public static void main(String[] args) {
        for (Promemoria p : Notifiche.class.getAnnotationsByType(Promemoria.class)) {
            System.out.printf("%02d:%02d%n", p.ora(), p.minuto());
        }
    }

}
```

L'output è `09:30` e `17:00`. `getAnnotationsByType` chiede le annotazioni *del tipo ripetuto* e restituisce entrambe. Se cerchiamo soltanto una presenza generica o usiamo un metodo diverso della reflection senza capire il ruolo del contenitore, possiamo osservare una rappresentazione diversa. Togliere `@Repeatable` mantenendo i due usi è una prova negativa: il compilatore rifiuta la duplicazione. Lasciare `@Repeatable` ma assegnare al contenitore una retention più breve di quella dell'annotazione ripetuta è un altro errore: le regole di compatibilità sono controllate alla compilazione. Un’incoerenza fra il bersaglio di `@Scheduler` e quello del contenitore renderebbe invalida la dichiarazione. Dichiarare insieme target e retention permette di controllarne la compatibilità.

`@Repeatable` non è un'alternativa a una struttura dati quando gli orari devono cambiare durante l'esecuzione. Le annotazioni appartengono al programma compilato. Se un operatore deve spostare la notifica delle 17 alle 18 senza ricompilare, quegli orari appartengono a una configurazione o a dati persistiti. La scelta di mettere il dato in un'annotazione è appropriata solo quando il ciclo di vita del dato coincide con quello del codice che lo dichiara.

Lombok e il confine della piattaforma

Lombok è uno strumento esterno che può generare codice ripetitivo a partire da annotazioni. Per leggere un progetto che lo usa, occorre collocarlo correttamente: Lombok non è una parola chiave di Java né fa parte della libreria standard. Un progetto che lo usa dipende dalla versione della libreria, dalla configurazione della build e dal supporto degli strumenti di sviluppo. Un sorgente che usa `@Getter` può sembrare privo di un metodo che il codice chiamante invece usa; per capire davvero il programma bisogna vedere quale metodo è stato generato e quale compilazione lo ha prodotto.

Le annotazioni Lombok come `@Getter`, `@Setter`, `@ToString` e quelle dedicate ai costruttori risolvono ripetizioni diverse. Generare un setter per ogni campo non è automaticamente una buona scelta di incapsulamento: un oggetto che deve preservare invarianti può avere bisogno di operazioni con significato, non di accesso indiscriminato allo stato. Per semplici aggregati di dati immutabili, prima

di introdurre una dipendenza esterna valutiamo i record già incontrati nel capitolo 7; non coprono però tutti gli usi di una classe ordinaria. Se un progetto adotta Lombok, la revisione deve mostrare sorgente, configurazione e API risultante, senza attribuire il comportamento alla sola presenza di `@`.

> **Nota bene – Non tutte le annotazioni generano codice.** `@Override` verifica una relazione del linguaggio; `@Etichetta` nel nostro esempio è leggibile a runtime; `EtichettaProcessor` produce una nota durante la compilazione; Lombok aggiunge un'elaborazione esterna. Chiamarle tutte «preprocessori» nasconderebbe il momento e il responsabile dell'azione.

Dal `BoilerPlatePersona` storico a una decisione di progetto

Una classe `BoilerPlatePersona` con sei campi privati richiede costruttori, getter e setter, oltre ai metodi `equals`, `hashCode` e `toString` quando il suo contratto li richiede. Questa quantità di codice rende concreto il costo della ripetizione e ci invita a scegliere la forma adatta al ruolo dell'oggetto. Se `Persona` è un contenitore di nome e cognome che viaggia fra due componenti e non deve cambiare, in Java 25 un `record Persona(String nome, String cognome) { }` fornisce un costruttore canonico, accessori e implementazioni basate sui componenti dei metodi di `Object`. È una possibilità stabile del linguaggio, senza un processore esterno. Non sostituisce però una classe che possiede un'identità persistente, campi modificabili, invarianti particolari o un protocollo di framework che richiede una forma differente. Prima descriviamo il ruolo dell'oggetto, poi confrontiamo la forma che ci costa meno mantenere.

Se il progetto usa Lombok, possiamo esprimere la classe storica con `@Getter`, `@Setter`, `@NoArgsConstructor`, `@AllArgsConstructor`, `@EqualsAndHashCode` e `@ToString`. Ognuna risponde a un pezzo distinto della ripetizione. `@Getter` produce un accessore; `@Setter` produce una mutazione; le annotazioni sui costruttori scelgono quali parametri sono disponibili; le ultime due decidono rappresentazione testuale e uguaglianza. L'annotazione `@Data` combina alcune di queste scelte, ma la sua brevità rende ancora più importante sapere quali metodi vengono prodotti. Non dobbiamo aggiungere tutti i setter soltanto perché è comodo: una `Persona` che non deve poter avere un nome vuoto ha bisogno di un controllo nel costruttore e nelle operazioni di modifica. Un setter generato che assegna direttamente un campo può aggirare la regola che intendevamo imporre.

L'uguaglianza merita un controllo a parte. Se `equals` e `hashCode` dipendono da `nome` e `cognome`, due persone con gli stessi nomi risultano uguali secondo quella definizione, anche se nel dominio sono individui diversi. Se cambiamo il cognome dopo aver inserito l'oggetto in un `HashSet`, il suo hash può cambiare e la raccolta potrebbe non ritrovarlo dove l'aveva collocato. Non è un difetto specifico di Lombok: è la conseguenza di una regola di uguaglianza basata su stato mutabile. La generazione di codice non sceglie al nostro posto quale identità appartenga al dominio. Analogamente, un `toString` che include un campo riservato può portarlo nei log; escludere il campo nella configurazione dell'annotazione è una decisione di sicurezza, non di estetica.

Consideriamo due oggetti `Persona`: confrontiamo l'uguaglianza e poi modifichiamo il secondo con setter. Il risultato `false` fra nomi diversi è prevedibile, ma non prova

che l'uguaglianza sia corretta per ogni stato o che l'hash sia stabile nel tempo. Il numero stampato da `hashCode()` non va copiato come atteso universale: dipende dalla regola concreta e dai dati. Una prova utile confronta invarianti: oggetti che decidiamo uguali devono dare lo stesso hash; modifiche vietate devono essere rifiutate; `toString` non deve esporre segreti. In questo modo il lettore vede perché la riduzione delle righe è soltanto una parte del problema.

La documentazione ufficiale di Lombok sulle funzionalità descrive le annotazioni disponibili e il comando *delombok*, che produce una forma espansa del sorgente. Questo può aiutare nella revisione o quando un altro strumento deve leggere il codice generato. Nel caso di `@EqualsAndHashCode`, la pagina ufficiale spiega quali campi vengono considerati per impostazione predefinita e come modificare la selezione. La scelta va verificata con la versione effettiva dichiarata nella build. Lombok può anche interagire con IDE e compilatori in modi che cambiano tra versioni: per questo non stampiamo qui un comando di compilazione senza una dipendenza fissata e provata. Il programma didattico del capitolo usa soltanto il JDK; Lombok è un confronto guidato con la classe `Persona`, non una dipendenza nascosta.

Altre meta-annotazioni e altri controlli

`@Documented` merita una decisione esplicita quando un'annotazione è parte del contratto pubblico di una libreria. Segnala alla generazione della documentazione che l'uso di quell'annotazione va rappresentato nella documentazione dell'elemento annotato. Non fa eseguire un controllo e non cambia la retention scelta. Un'annotazione interna che aiuta soltanto il compilatore potrebbe non dover comparire nella pagina pubblica; una che descrive una restrizione per chi usa l'API potrebbe invece essere utile mostrarla. Come sempre, la presenza dell'etichetta non rende vera o completa la documentazione: dobbiamo leggere il risultato generato.

Con `@SuppressWarnings("unchecked")` possiamo circoscrivere un avviso dopo aver esaminato un cast che il compilatore non sa dimostrare sicuro. Nel capitolo 15 incontreremo esattamente quella situazione con i tipi generici e l'erasure. Mettere la soppressione su un'intera classe per zittire una riga allarga la zona cieca: altri avvisi della stessa categoria potrebbero comparire più tardi senza essere notati. La buona abitudine è tenere la soppressione nel punto più piccolo possibile, spiegare quale invariante la rende accettabile e accompagnarla con un test. `@SafeVarargs` è ancora più impegnativa: è una dichiarazione dell'autore del metodo sulla sicurezza dell'uso degli argomenti variabili, non un'impostazione per far sparire un messaggio fastidioso.

Una `@Deprecated` senza un percorso di migrazione lascia il chiamante a metà. Se manteniamo una vecchia API per compatibilità, la Javadoc dovrebbe dire quale metodo usare e quale differenza di comportamento attendersi. `since` registra la versione a partire dalla quale l'API è deprecata; `forRemoval` comunica se è prevista la rimozione futura, ma non fissa una data. Un team che trova l'annotazione può decidere se migrare subito, ma deve conoscere la release e il rischio del cambiamento. Questo è un altro esempio di metadato che aiuta soltanto se qualcuno lo legge e se il messaggio è abbastanza preciso per agire.

Per chiudere il cerchio, torniamo alla domanda iniziale: quale problema risolve il metadato? `@Override` rende verificabile una firma; `@Retention` e `@Target` regolano

dove un'informazione vive; `@Inherited` definisce una forma limitata di visibilità nelle sottoclassi; `@Repeatable` rappresenta più valori; un processor e Lombok possono produrre diagnosi o codice. La sintassi comune non cancella queste differenze. Se davanti a una nuova annotazione sappiamo cercare il suo lettore, il momento in cui opera e la prova del risultato, abbiamo imparato a riusare il modello anche fuori da questo capitolo.

Per verificare

Esegui `DemoAnnotazioni`, `DemoPolitiche` e `DemoPromemoria`. Prima di ogni esecuzione, scrivi le righe attese e spiega quale API legge il metadato. Poi rimuovi `@Repeatable` lasciando entrambe le annotazioni sulla classe: prevedi la diagnostica del compilatore. Come seconda prova, cambia la retention in `CLASS`, ricompila ed esegui: spiega perché il processore può ancora vedere l'annotazione del sorgente mentre la reflection non la mostra. Per un caso applicativo, definisci un'annotazione `@Responsabile` su un metodo e decidi prima chi dovrà leggerla e in quale fase: compilatore, processore o applicazione in esecuzione.

15. Generics e controllo dei tipi

La pila che accettava tutto

Torniamo alla pila che accompagna il libro. La versione del capitolo 8 contiene soltanto `int`: è semplice e ci ha permesso di studiare stato, invarianti ed eccezioni. Se vogliamo una pila di titoli, copiarla e sostituire ogni `int` con `String` produce due classi quasi identiche. Possiamo provare a generalizzare usando `Object`, il supertipo degli oggetti. L'idea evita la duplicazione, ma trasferisce un problema a chi usa la pila: `estrai()` restituisce `Object`, quindi per ottenere una `String` occorre un cast. Se per errore qualcuno vi inserisce un `Integer`, il programma può compilare e fallire soltanto quando estrae il valore.

I **generics** permettono di dichiarare una relazione fra tipi. `Pila<T>` significa che il tipo degli elementi è scelto quando si usa la classe: `Pila<String>` accetta stringhe e restituisce stringhe; `Pila<Integer>` fa lo stesso con interi rappresentati come oggetti. `T` è un **parametro di tipo**, un nome che compare nella dichiarazione e viene sostituito concettualmente dall'argomento di tipo scelto dal chiamante. `String` in `Pila<String>` è un **argomento di tipo**. Questa somiglianza con parametri e argomenti dei metodi aiuta a leggere la sintassi, ma qui il compilatore controlla tipi, non valori passati durante una chiamata.

> **Concetto chiave – Sicurezza statica.** Nel caso ordinario, inserire un valore di tipo incompatibile provoca un errore del compilatore prima dell’esecuzione. Alcune forme di codice riducono però i controlli disponibili. Un *raw type* usa una classe generica senza l’argomento di tipo, per esempio `List` anziché `List<String>`; un cast non controllato chiede una conversione generica che il compilatore non può verificare interamente. Sono forme che possiamo incontrare soprattutto ai confini con vecchie API e che approfondiremo nel capitolo. La garanzia ha quindi condizioni precise: usando coerentemente i tipi parametrizzati spostiamo molti errori dal momento dell’uso al momento della scrittura, senza rendere impossibile ogni errore durante l’esecuzione.

Il programma completo costruisce una piccola `Pila<T>` con nodi collegati. La scelta dei nodi ci permette di evitare array generici e cast nel percorso principale. Ogni nodo conserva un valore `T` e il riferimento al nodo precedente. `inserisci` crea una nuova cima; `estrai` restituisce il valore della cima e fa avanzare il riferimento. Quando la pila è vuota, mantiene il contratto già visto: lancia `IllegalStateException` invece di inventare un valore sentinella.

```
static final class Pila<T> {
    private static final class Nodo<T> {
        final T valore;
        final Nodo<T> precedente;

        Nodo(T valore, Nodo<T> precedente) {
            this.valore = valore;
            this.precedente = precedente;
        }
    }

    private Nodo<T> cima;

    void inserisci(T valore) {
        cima = new Nodo<>(valore, cima);
    }


    T estrai() {
        if (cima == null) {
            throw new IllegalStateException("pila vuota");
        }
        T valore = cima.valore;
        cima = cima.precedente;
        return valore;
    }
}
```

In `main`, `Pila<String> titoli = new Pila<>();` usa il **diamond** `<>`: il compilatore ricava l'argomento di tipo dal lato sinistro. Dopo `titoli.inserisci("Java")`, possiamo assegnare `titoli.estrai()` direttamente a una `String` e chiamare `toUpperCase()`. L'output è `JAVA`. Scrivere `titoli.inserisci(3)` non è una variante che fallirà più tardi: è codice non compilabile, perché `3` diventa un `Integer` tramite boxing e non è una `String`.

Più tipi, nomi e valori primitivi

Un parametro di tipo può descrivere un ruolo; più parametri permettono di tenere distinti ruoli diversi nella stessa classe. Pensiamo a una coppia chiave–valore: `Coppia<K, V>` usa `K` per la chiave e `V` per il valore. Se costruiamo `Coppia<String, Integer>`, un metodo che restituisce `K` offre una `String`, mentre un metodo che restituisce `V` offre un `Integer`; non serve un cast. Nomi come `T` (*type*), `E` (*element*), `K` (*key*) e `V` (*value*) sono convenzioni riconoscibili nelle API Java. Non sono parole riservate: potremmo dichiarare `Pila<Elementi>` scegliendo `Elementi` come nome del parametro, ma una lettera convenzionale rende più immediate le firme ripetute. Se i parametri sono tanti e non sappiamo più quale dato rappresentino, fermiamoci a chiarire le responsabilità della classe.

L'argomento di un tipo parametrizzato deve essere un tipo riferimento. Non scriviamo `Pila<int>`: per gli interi usiamo `Pila<Integer>`. L'autoboxing rende naturale inserire `3`, perché il compilatore produce la conversione verso `Integer`; questo non significa che boxing non abbia costo o che `null` si comporti come uno zero. In una struttura numerica molto grande, contenitore di wrapper e array primitivi hanno proprietà di memoria e accesso diverse. La scelta del generic risolve il problema del controllo statico dei tipi; non è automaticamente la scelta

più efficiente per ogni carico. Il capitolo sulle collezioni riprenderà questa distinzione quando confronteremo le strutture.

Prima del diamond si ripeteva l'argomento su entrambi i lati, per esempio `new Pila<String>();` con `new Pila<>()` il compilatore usa il tipo bersaglio già dichiarato. Il simbolo `<>` non indica «qualunque tipo» e non equivale al raw type `Pila`. Una variabile raw perde informazione e genera warning nelle operazioni non controllate. Se un vecchio progetto richiede quel confine, circoscriviamo l'interoperabilità, controlliamo il contenuto e non facciamo propagare il raw type in tutto il programma. Questo recupera il motivo della sintassi moderna: ridurre ripetizione senza rinunciare alla promessa della firma.

Perché `List<Integer>` non è `List<Number>`

Nel capitolo 9 abbiamo imparato che un `Integer` è un `Number`. Sarebbe allora lecito assegnare `List<Integer>` a `List<Number>`? No. Se fosse consentito, attraverso il secondo riferimento potremmo inserire un `Double` nella lista che il primo riferimento promette contenere solo `Integer`. I tipi generici Java sono normalmente **invarianti**: la relazione fra gli argomenti di tipo non si trasferisce automaticamente ai tipi parametrizzati.

La figura 15.1 separa la parentela fra elementi dalla relazione fra contenitori. È uno schema di tipi, non una disposizione degli oggetti nella memoria.

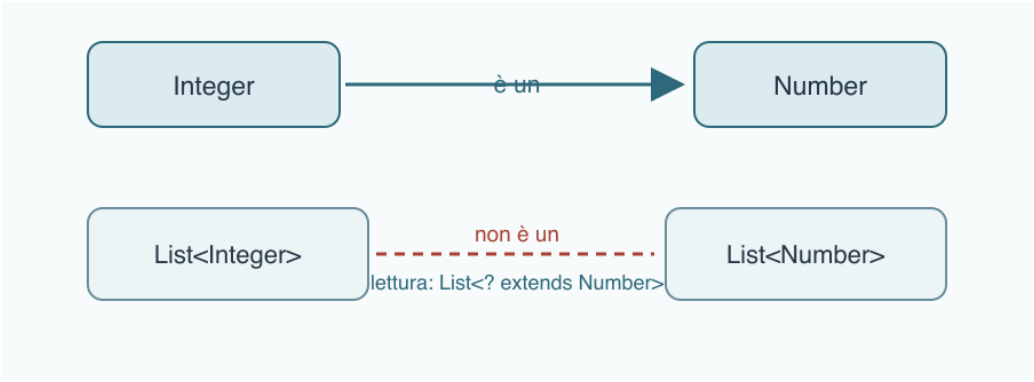


Figura 15.1 – `Integer` è un sottotipo di `Number`, ma `List<Integer>` non è un sottotipo di `List<Number>`. La wildcard descrive la variazione ammessa dal singolo metodo.

Quando un metodo deve soltanto **leggere** numeri da liste di sottotipi diversi, può ricevere `List<? extends Number>`. Il punto interrogativo è una **wildcard**, un tipo non nominato che rispetta il limite `Number`. Dal parametro possiamo leggere un `Number`, ma non inserire arbitrariamente un `Number`: la lista effettiva potrebbe essere una `List<Integer>`. Quando un metodo deve **scrivere** valori `Number` in una lista, può ricevere `List<? super Number>`: la lista effettiva può essere, per esempio, `List<Number>` o `List<Object>`. Leggere da essa non promette un tipo più preciso di `Object`.

Nel programma, `copiaNumeri` attraversa una `List<Integer>` attraverso un parametro `List<? extends Number>` e aggiunge i valori a una `List<Number>` attraverso `List<? super Number>`. L'output è `[2, 3]`. La regola spesso ricordata come **PECS** (*producer extends, consumer super*) descrive chi produce valori da leggere e chi li consuma tramite scrittura. Non sostituisce la lettura del contratto:

una stessa collezione può essere letta e scritta, e allora una wildcard potrebbe non essere adatta.

> **Definizione – Parametro di tipo e wildcard.** `T` è un nome che può legare più posizioni della stessa firma: un metodo `<T> T primo(List<T> valori)` promette di restituire lo stesso tipo degli elementi ricevuti. `?` rappresenta un tipo sconosciuto e si usa quando quella relazione non deve essere espressa con un nome. Sostituire sempre `T` con `?` farebbe perdere informazioni al chiamante.

Un metodo può dichiarare i propri parametri di tipo senza che la classe sia generica, per esempio `static <T> T primo(List<T> valori)`. Un'interfaccia può essere generica: `Comparable<T>` esprime quale tipo di valore può essere confrontato. Un parametro di tipo può avere un limite, per esempio `<T extends Number>`; la parola `extends` qui vale per tipi classe e interfaccia. Queste forme non obbligano a usare gerarchie complicate: servono quando il metodo deve conservare una relazione verificabile fra input e output.

Erasure e limiti reali

Java implementa i generics mediante **erasure**, o cancellazione dei tipi, per mantenere compatibilità con codice precedente. Il compilatore verifica le relazioni generiche e produce un file `.class` in cui una parte degli argomenti di tipo non è disponibile per le normali operazioni a runtime. Non significa che «i generics non esistano»: le firme possono conservare metadati generici leggibili con reflection, e soprattutto il controllo a compilazione ha già cambiato quali programmi sono ammessi. Significa che non possiamo scrivere `new T()` o `new T[10]` nel caso generale, né distinguere `List<String>` da `List<Integer>` usando `instanceof`.

Quando una classe generica implementa o ridefinisce un metodo con firma che, dopo erasure, richiede un adattamento, il compilatore può generare un **bridge method**. È un metodo sintetico nel bytecode che conserva il comportamento polimorfico atteso. Non occorre scriverlo a mano, ma sapere che esiste aiuta quando si esamina una classe con `javap` o reflection. La JLS 25 sui generics descrive i tipi parametrizzati, le wildcard e l'erasure.

Nel programma `DemoPonte.java` `Deposito<T>` dichiara `imposta(T)`. La sottoclasse fissa `T` a `String` e ridefinisce `imposta(String)` aggiungendo una normalizzazione. Una variabile dichiarata `Deposito<String>` punta all'oggetto `DepositoTitoli`; la chiamata deve raggiungere la ridefinizione e stampare `Java`, senza gli spazi ai bordi.

```
public final class DemoPonte {
    static class Deposito<T> {
        private T valore;

        void imposta(T nuovo) {
            valore = nuovo;
        }

        T leggi() {
            return valore;
        }
    }

    static final class DepositoTitoli extends Deposito<String> {
        @Override
        void imposta(String nuovo) {


            super.imposta(nuovo.strip());
        }
    }

    private DemoPonte() { }

    public static void main(String[] args) {
        Deposito<String> deposito = new DepositoTitoli();
        deposito.imposta("  Java  ");
        System.out.println(deposito.leggi());
    }
}
```

Compilando con `javac --release 25 -Xlint:all -d build DemoPonte.java`, poi eseguendo `java -cp build DemoPonte`, osserviamo `Java`. Dopo l'erasure, la firma del metodo nella classe base usa `Object`; quella scritta nella sottoclasse usa `String`. Se ispezioniamo la sottoclasse con `javap -p -classpath build 'DemoPonte$DepositoTitoli'`, vediamo sia `imposta(java.lang.String)` sia `imposta(java.lang.Object)`. Il secondo è il metodo ponte generato dal compilatore: riceve un `Object`, lo controlla come `String` e inoltra la chiamata al metodo della sottoclasse. In questo modo una chiamata attraverso il contratto della classe base raggiunge ancora la ridefinizione che elimina gli spazi. È il **dispatch**, cioè la scelta dell'implementazione da eseguire. Il ponte preserva questa scelta dopo l'erasure; non ricrea a runtime gli argomenti di tipo cancellati.

Un **raw type**, come `List` senza argomento, serve soprattutto per interoperare con vecchie API. Può far perdere controlli e produrre warning. Se tramite un raw type inseriamo un valore incompatibile in una lista parametrizzata, otteniamo *heap pollution*: il contenuto reale non rispetta il tipo promesso dal sorgente. Per gli array conta anche quali informazioni restano durante l'esecuzione. Un tipo è **reificabile** quando è disponibile completamente a runtime: `String` lo è, mentre `List<String>` non conserva nello stesso modo l'argomento `String`. Poiché un array controlla il tipo dei propri componenti, non possiamo creare direttamente `new List<String>[10]`. I **varargs**, parametri che accettano un numero variabile di argomenti, usano un array e possono incontrare lo stesso limite con tipi generici. `@SafeVarargs` sopprime alcuni avvisi e dichiara che l'autore ha verificato la

sicurezza del corpo; non esegue quella verifica al suo posto e non rende sicuro un array esposto o contaminato con elementi incompatibili.

> **Approfondimento – Cattura della wildcard.** Un metodo che riceve `List<?>` può leggere gli elementi come `Object`, ma non sa quale tipo specifico sia stato catturato dal punto interrogativo. Talvolta un piccolo metodo generico privato può dare un nome temporaneo a quel tipo e compiere un'operazione sicura, come scambiare due posizioni della stessa lista. La wildcard non è «qualunque oggetto»: è un tipo preciso ma sconosciuto a quel punto del sorgente.

Tre firme che sembrano simili

Consideriamo tre metodi che cercano il primo elemento di una lista. `Object primo(List<?> valori)` può leggere un elemento, ma al chiamante restituisce soltanto `Object`: l'informazione sul tipo specifico non passa attraverso la firma. `<T> T primo(List<T> valori)` lega invece il tipo degli elementi al tipo restituito. Se passiamo `List<String>`, il risultato è una `String` per il compilatore. `<T extends Number> T primoNumero(List<T> valori)` conserva la stessa relazione, ma limita i tipi ammessi ai sottotipi di `Number`. La firma più restrittiva è giustificata soltanto se il corpo o il contratto hanno davvero bisogno di operazioni numeriche.

Una wildcard `? extends Number` non è un modo più breve per dire che il metodo è generico. Va bene quando il metodo legge numeri e non deve restituire al chiamante lo stesso tipo specifico ricevuto. Se il risultato deve mantenere quella relazione, serve un parametro nominato. Questo è il motivo per cui le API standard contengono sia parametri di tipo sia wildcard: risolvono problemi diversi. Una firma troppo generale obbliga il chiamante a cast; una troppo stretta rifiuta usi perfettamente sicuri.

Quando basta rendere generico un metodo

Un metodo statico generico permette di legare tipi nella firma senza rendere generica l'intera classe. Un metodo `static <T> boolean sonoUguali(T primo, T secondo)` dichiara il suo `T` **prima** del tipo di ritorno; quel `T` appartiene alla chiamata del metodo, non a un'istanza di una classe generica. Possiamo chiamarlo con due stringhe e il compilatore inferisce un tipo compatibile. Possiamo anche scrivere un test con `Integer`; non abbiamo creato due classi diverse. Se la sola operazione del corpo è il confronto per uguaglianza, però, possiamo usare `Objects.equals(primo, secondo)` e domandarci se la firma generic aggiunga davvero un vincolo utile. Se il metodo non lega posizioni in modo significativo, la genericità può essere rumore.

Si può esplicitare l'argomento di tipo nella chiamata, per esempio `Strumenti.<String>sonoUguali("a", "b")`, ma nella maggior parte dei casi l'inferenza rende quella ripetizione inutile. L'esplicitazione torna utile quando l'inferenza sceglierebbe un tipo troppo ampio o quando vogliamo chiarire un'intenzione complessa. La domanda resta la stessa: quale relazione verificabile fra gli argomenti e il risultato stiamo promettendo?

Quando il costruttore ha un proprio tipo

Una classe può non essere generica e avere comunque un **costruttore generico**. Immaginiamo una ricevuta che deve conservare soltanto una descrizione testuale, ma può nascere da un numero, da una stringa o da un altro valore. Il chiamante fornisce sia il valore sia la funzione che sa descrivere proprio quel tipo. Il costruttore dichiara `<T>` **prima del nome della classe**: `<T> Ricevuta(T origine, Function<T, String> descrivi)`. Il medesimo `T` lega i due argomenti della singola costruzione. Non è un parametro di tipo dell'intera classe e non rimane disponibile nei suoi altri metodi.

Nel programma completo, la prima ricevuta nasce dall'intero 42 e la seconda dalla stringa `"java"`. Il compilatore deduce un tipo compatibile per ciascuna chiamata e controlla che la funzione accetti quel valore. Il risultato stampato è `n=42` e poi `JAVA`. Se passassimo una funzione che accetta soltanto `String` insieme a un valore dichiarato `Integer`, il contratto non sarebbe soddisfatto: l'errore emerge in compilazione. Questo è il beneficio concreto del parametro di tipo, che `Object` non esprimerebbe con la stessa precisione.

```
import java.util.function.Function;

public final class DemoCostruttoreGenerico {
    static final class Ricevuta {
        private final String descrizione;

        <T> Ricevuta(T origine, Function<T, String> descrivi) {
            descrizione = descrivi.apply(origine);
        }

        String descrizione() {
            return descrizione;
        }
    }

    private DemoCostruttoreGenerico() { }


    public static void main(String[] args) {
        Ricevuta numero = new Ricevuta(42, n -> "n=" + n);
        Ricevuta parola = new Ricevuta("java", String::toUpperCase);
        System.out.println(numero.descrizione());
        System.out.println(parola.descrizione());
    }
}
```

> **Approfondimento – Due vite diverse per `T`.** In `Pila<T>` il parametro è dichiarato dalla classe e collega metodi di una stessa istanza. In `<T> Ricevuta(...)` appartiene soltanto al costruttore e collega gli argomenti di quella chiamata. Un costruttore generico è utile quando questo legame evita cast o controlli tardivi; per convertire semplicemente qualunque oggetto con `toString()`, un parametro `Object` può bastare. La specifica Java 25 conferma che la classe può essere generica o meno indipendentemente dal suo costruttore.

Per una somma non basta `<T>` senza limite, perché un `T` qualunque non offre `doubleValue()`. `<T extends Number>` permette di leggere quel metodo e accetta `Integer`, `Double` e altri sottotipi di `Number`, ma esclude `String` già in compilazione. Se il corpo richiede anche che i valori siano confrontabili, possiamo esprimere più limiti, con la classe per prima e le interfacce dopo: `<T extends Number & Comparable<T>>`. Due classi come `Integer & Double` non formano invece un limite sensato e sono rifiutate. Non scegliamo i limiti per descrivere gli esempi usati nel test; li scegliamo a partire dalle operazioni che l'implementazione deve poter fare. È il legame fra problema e firma che rende leggibile una generic API.

Inferenza e tipo bersaglio

Nel codice `new Pila<>()`, il compilatore usa il tipo della variabile a sinistra per capire l'argomento omesso nel diamond. L'**inferenza di tipo** compare anche nelle chiamate a metodi generici: il compilatore combina argomenti passati e tipo atteso del risultato. Non significa che una variabile cambi tipo durante l'esecuzione. Dopo `Pila<String> titoli`, il contratto dei metodi resta quello per stringhe. Se un'inferenza produce un tipo più ampio del previsto, una dichiarazione esplicita o un nome di metodo più preciso può rendere il codice più leggibile senza combattere il compilatore.

I limiti possono essere multipli, per esempio un parametro che debba estendere una classe e implementare un'interfaccia. La sintassi usa `&` dopo `extends`; se compare un limite di classe, va per primo. È una capacità utile in librerie generiche, ma non dovrebbe comparire nel percorso principale senza un'operazione concreta che la richieda. Anche il principio del capitolo 13 vale qui: prima il contratto necessario, poi la sintassi che lo esprime.

Che cosa si può leggere con reflection

Sebbene un oggetto `new Pila<String>()` non conservi un'etichetta runtime che permetta di chiedergli direttamente «sei una pila di stringhe?», una dichiarazione come un campo `Pila<String> titoli` può conservare nella firma del file compilato informazioni generiche. Le API di reflection rappresentano queste forme con `Type` e specializzazioni come `ParameterizedType`, `TypeVariable` e `WildcardType`. Leggere una dichiarazione e interrogare un'istanza sono operazioni diverse: l'erasure limita la seconda, ma non cancella necessariamente ogni traccia della prima.

Questo chiarimento evita due estremi. Non possiamo fare un controllo `instanceof Pila<String>` su una pila qualunque, perché l'argomento di tipo non è reificato in quel modo. Non è vero nemmeno che il compilatore «dimentichi» ogni generic e che i file `.class` non possano documentare firme parametrizzate. Quando una libreria usa reflection per risolvere un tipo generico, occorre leggere esattamente *dove* quel tipo è stato dichiarato e quali metadati sono disponibili.

Una pila, tre contratti sui tipi

Facciamo evolvere la pila attraverso più forme, senza saltare la ragione di ogni passaggio. Una pila che usa `int[]` conserva solo interi primitivi; se serve anche una pila di titoli, copiare tutto il codice duplica l'invariante della cima e la gestione del caso vuoto. Una pila che conserva `Object` elimina la duplicazione ma accetta insieme `"Java"` e 42, senza poter promettere al chiamante quale tipo uscirà. Il cast `(String) pila.pop()` non ripara l'invariante: chiede soltanto alla JVM di controllare più tardi se l'ipotesi del chiamante era giusta.

Con `Pila<T>`, la relazione entra nella dichiarazione. `void inserisci(T valore)` e `T estrai()` usano lo stesso parametro `T`; quando scriviamo `Pila<String>`, il compilatore collega entrambe le posizioni a `String`. Questa è la parte essenziale, più importante del nome della classe o della scelta fra array e nodi. Il diamond `new Pila<>()` evita di ripetere `String` nel costruttore, ma non crea una pila «senza tipo». Il tipo bersaglio a sinistra fornisce il vincolo; se usiamo un raw type `Pila`, lo perdiamo e riapriamo proprio il rischio che volevamo chiudere.

Un'interfaccia generica può dichiarare il medesimo contratto, per esempio `interface Deposito<T> { void inserisci(T valore); T estrai(); }`. Una classe può implementarlo mantenendo il parametro, `class Pila<T> implements Deposito<T>`, oppure fissandolo, `class PilaDiTitoli implements Deposito<String>`. Nel secondo caso la classe concreta non è generica, ma i suoi metodi devono rispettare le firme specializzate: l'argomento e il risultato sono stringhe. Questa variante corregge un equivoco frequente: l'interfaccia generica non obbliga ogni classe che la implementa a restare genericamente parametrizzata.

> **Approfondimento – Il vincolo risponde a una domanda.** `<T extends Number>` è utile se il corpo deve chiamare metodi garantiti da `Number`. Scriverlo soltanto perché gli esempi usano numeri impedirebbe una `Pila<String>` senza offrire alcun vantaggio: `inserisci` ed `estrai` non hanno bisogno di sommare. Se un algoritmo deve ordinare gli elementi, occorre invece un contratto di confronto e un limite appropriato, oppure un `Comparator` passato dall'esterno. Il limite si sceglie dall'operazione richiesta, non dal primo caso d'uso.

Quando i controlli statici incontrano gli array

Un array Java conosce a runtime il proprio tipo di componente. Per questo un riferimento `Object[]` che punta a un vero `String[]` non può accettare un `Integer`: il tentativo viene respinto a runtime. Una `List<String>`, invece, non porta nello stesso modo un controllo runtime dell'argomento `String`; l'erasure consente interoperabilità con codice più vecchio e sposta la verifica principale alla compilazione. Combinare direttamente array e tipi parametrizzati non reificabili crea quindi problemi: `new List<String>[10]` non è consentito.

Quando un metodo varargs riceve `List<String>...`, il parametro variabile viene rappresentato tramite un array. Un'implementazione imprudente può farvi entrare una lista di un altro tipo attraverso un alias meno preciso e produrre *heap pollution*. Un warning del compilatore non va zittito solo per rendere la build pulita: indica che il contratto generico e la rappresentazione dell'array richiedono

una prova in più. `@SafeVarargs` è una dichiarazione di responsabilità dell'autore del metodo, ammessa nei casi previsti dal linguaggio; va applicata soltanto dopo aver verificato che il corpo non esponga o contamini quell'array.

Leggere e scrivere una wildcard senza indovinare

Riprendiamo il metodo `copiaNumeri` del sorgente completo. Riceve una sorgente `List<? extends Number>` e una destinazione `List<? super Number>`. Per la prima chiamiamo `get(0)` e otteniamo un valore che possiamo trattare come `Number`, perché qualunque tipo effettivo catturato dal punto interrogativo deve essere un sottotipo di `Number`. Non possiamo però chiamare `add(3.5)` sulla sorgente: se fosse una lista di `Integer`, il `Double` 3.5 violerebbe il tipo degli elementi. Possiamo aggiungere `null` dal punto di vista del tipo generico, ma farlo qui non avrebbe alcun senso per l'algoritmo.

Nella destinazione il ragionamento si inverte. Possiamo chiamare `add(numero)` con un `Number`: la lista reale è dichiarata per `Number` oppure per un suo supertipo, dunque può ospitarlo. Se poi leggiamo con `get(0)`, il tipo statico sicuro del risultato è `Object`; la lista reale potrebbe essere `List<Object>` e contenere anche valori che non sono numeri. Non è una stranezza del compilatore: sono due promesse differenti, una sulla scrittura e l'altra sulla lettura. Per spostare i numeri usiamo entrambe nel verso in cui sono valide.

La parola PECS aiuta a ricordare la direzione, ma possiamo ricostruirla anche senza sigla. Chiediamoci che cosa deve produrre il parametro per il nostro metodo e che cosa deve accettare. Quando la sorgente produce valori leggibili come `Number`, `extends` è adatto; quando la destinazione consuma valori `Number`, `super` è adatto. Se un metodo deve aggiungere e leggere valori mantenendo esattamente lo stesso tipo, un parametro di tipo nominato `T` può esprimere la relazione meglio di due wildcard indipendenti. In quel caso la firma diventa parte della dimostrazione di sicurezza del metodo.

Il tipo sconosciuto resta un tipo, non diventa `Object`

Potremmo pensare di usare `Pila<Object>` come parametro per una funzione che non conosce il tipo della pila. Sarebbe comodo, ma `Pila<String>` non è assegnabile a `Pila<Object>`, per la stessa ragione per cui `List<Integer>` non è `List<Number>`. Se il metodo deve soltanto controllare se la pila è vuota o stamparne una descrizione, può ricevere `Pila<?>`: qualunque pila parametrizzata è compatibile con quella vista. Da `estrai()` potremo usare il valore solo come `Object`, perché la firma non promette altro. Non possiamo invece chiamare `inserisci("nuovo")`: la pila reale potrebbe essere una `Pila<Integer>`. La wildcard permette quindi di esprimere ciò che il metodo sa e ciò che non può promettere.

La figura 15.1 mostra la differenza fra la parentela di `Integer` e `Number` e l'invarianza di `List<Integer>` e `List<Number>`. Applichiamo ora lo stesso ragionamento alle pile: `Pila<Integer>` e `Pila<String>` non diventano sottotipi l'una dell'altra, ma possiamo usarle entrambe attraverso `Pila<?>` quando il metodo non richiede un tipo preciso di elemento. Il punto interrogativo conserva un tipo preciso ma sconosciuto in quel punto del sorgente. Possiamo leggere un elemento come `Object` oppure passare la pila a un metodo generico che cattura e mantiene quel tipo nella propria firma. Non possiamo invece inserirvi qualunque oggetto,

come faremmo in una `Pila<Object>`. La wildcard dichiara il limite della nostra conoscenza; un cast non lo elimina senza richiedere un controllo ulteriore.

Proviamo a trasferire il modello a un catalogo di numeri. Un metodo che calcola la somma dei `doubleValue()` può ricevere `List<? extends Number>`, perché legge ogni valore come `Number` e non deve aggiungerne. Un metodo che deposita nuovi `Integer` può ricevere `List<? super Integer>`; una `List<Number>` e una `List<Object>` sono destinazioni valide. Se invece dobbiamo spostare *lo stesso tipo preciso* da una struttura a un'altra e restituire un elemento di quel tipo, `<T>` collega le posizioni meglio di due wildcard. Scrivere tre firme diverse non è una gara di sintassi: sono tre contratti diversi che il compilatore può far rispettare.

Il lettore può verificare la scelta introducendo volontariamente un'operazione vietata in ciascun metodo e leggendo l'errore. Se il metodo `somma` tenta di aggiungere un `Double` alla sorgente `? extends Number`, il compilatore lo respinge; se il metodo che deposita in `? super Integer` pretende di leggere direttamente un `Integer`, lo respinge ancora. La diagnostica è una spiegazione operativa del limite. Una volta compreso il perché, ricordare la sigla PECS diventa più facile e, soprattutto, meno pericoloso da applicare a memoria.

Per verificare

Compila `DemoGenerics.java` con `javac --release 25 -Xlint:all -d build DemoGenerics.java` ed esegui la classe. Esegui anche `DemoPonte.java` e usa `javap -p` sulla classe interna `DepositoTitoli` per individuare il metodo ponte generato dopo l'erasure. Poi prova ad assegnare `List<Integer>` a `List<Number>` in una copia del file e leggi la diagnostica. Infine modifica `copiaNumeri` affinché la sorgente sia `List<? extends Integer>`, il valore letto sia `Integer` e la destinazione sia `List<? super Integer>`; prova a passarle prima `List<Number>`, poi `List<Object>`. Entrambi i casi devono compilare, ma la scelta dei limiti cambia quali valori il corpo può leggere e aggiungere. Le attività del capitolo 15 useranno la pila per far emergere i vantaggi del controllo statico rispetto alla vecchia soluzione con `Object` e cast.

16. Funzioni e lambda in Java

Passare un comportamento come argomento

Finora abbiamo passato a un metodo numeri, stringhe e oggetti. Talvolta vogliamo passargli anche una *scelta*: come trasformare un testo, quando accettarlo, che cosa fare con il risultato. In Java questa scelta può essere rappresentata da un oggetto che implementa un'interfaccia con un solo metodo astratto. Una **lambda** è una forma concisa per fornire proprio quell'implementazione. Non trasforma Java in un linguaggio funzionale puro, ma rende più semplice trattare un comportamento come valore utilizzabile da altre operazioni.

Partiamo da un problema familiare: un metodo deve svolgere un lavoro stabile, ma il chiamante vuole cambiare una decisione. Vedremo come dichiarare il comportamento richiesto, come passarlo e come controllare il risultato. Solo quando un esempio lo richiederà daremo un nome ai principi più avanzati. Così una definizione non resta sospesa senza un programma a cui applicarla.

> **Definizione – Interfaccia funzionale.** È un'interfaccia con un solo metodo astratto da implementare, per questo chiamata anche **SAM**, da *Single Abstract Method*. Può avere metodi default o statici senza perdere questa proprietà. `@FunctionalInterface` chiede al compilatore di controllarla. La lambda riceve il **tipo bersaglio** dal contesto: non è un pezzo di codice privo di tipo che si possa assegnare a qualunque variabile.

Perché introdurre le funzioni in un linguaggio a oggetti

Immaginiamo di avere un metodo che confronta due prezzi. Il metodo sa ottenere i prezzi, ma non sa se il chiamante vuole calcolare la differenza, la somma o il maggiore. Una prima soluzione consiste nel passargli un codice numerico, per esempio 1 per la somma e 2 per la differenza. Il metodo contiene allora un `switch` che cresce ogni volta che appare una nuova operazione. Il numero 2, letto dopo sei mesi, non spiega nulla. Una seconda soluzione consiste nel duplicare il metodo per ogni scelta: anche questa strada cresce male, perché tutta la parte che ottiene e valida i prezzi viene ripetuta. La necessità concreta è **passare al metodo il comportamento che cambia**, lasciando in un solo posto il lavoro comune.

Questo problema non nasce con Java 8. Le interfacce e il polimorfismo del capitolo 13 lo risolvevano già: dichiariamo un contratto con `esegui(int, int)` e passiamo un oggetto che lo implementa. Le lambda abbreviano la scrittura dell'implementazione quando il contratto ha un solo metodo astratto. Il beneficio non è il numero di parentesi risparmiate, ma la possibilità di vedere vicino alla chiamata quale decisione stiamo affidando al metodo. Se la decisione è complessa, continua ad avere senso darle un nome in una classe o in un metodo ordinario.

Nel programma completo `DemoPercorsoLambda.java`, `calcola` riceve tre argomenti: i due numeri e un'`Operazione`. Il metodo non conosce la sottrazione né la somma; chiama `operazione.esegui(a, b)`. La prima operazione è una classe anonima, la seconda una lambda, la terza un riferimento al metodo statico `somma`. Le prime due stampano entrambe 7, la terza 13. Quel risultato uguale per due forme diverse è la

verifica che abbiamo cambiato *come forniamo* il comportamento, senza cambiare il contratto usato da `calcola`.

```
@FunctionalInterface
interface Operazione {
    int esegui(int a, int b);
    default String descrizione() {
        return "operazione su due interi";
    }
}

static int calcola(int a, int b, Operazione operazione) {
    return operazione.esegui(a, b);
}
```

La dichiarazione dell'interfaccia rende espliciti dominio e risultato: riceve due `int` e produce un `int`. Il metodo `descrizione` è `default`: possiede già un corpo e non aggiunge un nuovo obbligo a chi implementa l'interfaccia. Per questo l'interfaccia resta funzionale. L'annotazione `@FunctionalInterface` non crea la proprietà; chiede al compilatore di segnalarci se una futura modifica la distrugge. Se qualcuno aggiungesse `int annulla(int a, int b);` come secondo metodo astratto, il programma smetterebbe di compilare nel punto in cui il contratto è dichiarato, non soltanto nelle chiamate sparse nel progetto.

> **Dal problema alla soluzione.** Quando una famiglia di operazioni cambia ma chi le esegue resta uguale, un'interfaccia funzionale rappresenta la parte variabile. Una lambda è una possibile implementazione di quel contratto. Verifichiamo passando due comportamenti diversi allo stesso metodo e osservando risultati diversi sui medesimi dati; trasferiamo poi la stessa idea a un filtro di nomi, a un ordinamento o a una regola di validazione.

Prima delle lambda: la classe anonima

La classe anonima nasce in un'espressione con `new Operazione() { ... }`. Non ha un nome dichiarato nel sorgente, ma è un'implementazione concreta dell'interfaccia. Dobbiamo ripetere `@Override`, la firma completa del metodo e il `return`. La forma è più lunga perché possiamo dichiarare anche campi e altri metodi nell'oggetto anonimo. Non è soltanto una lambda scritta con più parole: la sua identità e il significato di `this` sono differenti.

```
Operazione classeAnonima = new Operazione() {
    @Override
    public int esegui(int a, int b) {
        return a - b;
    }
};
Operazione lambda = (a, b) -> a - b;
```

Nella riga della lambda, i tipi di `a` e `b` sono dedotti dal metodo astratto `esegui`. Il compilatore conosce il **tipo bersaglio** grazie alla dichiarazione `Operazione lambda`. Se tentassimo `var lambda = (a, b) -> a - b;`, mancherebbe il tipo bersaglio e il compilatore non saprebbe quale interfaccia implementare. `var` può

essere usato per i singoli parametri in una lambda quando il tipo bersaglio è già noto, per esempio `(var a, var b) -> a - b`, ma non risolve l'assenza del tipo della lambda intera. Questo dettaglio evita una confusione frequente: la deduzione dei parametri dipende dal contratto, non lo inventa.

Una funzione passata a `calcola` è un valore nel senso pratico che possiamo assegnarla a una variabile, trasmetterla a un metodo e farla restituire da un metodo. Il metodo `aggiungi(int incremento)` del programma restituisce un'`Operazione`; la lambda prodotta usa `incremento` in una chiamata successiva. Il metodo che riceve o restituisce un comportamento è chiamato **funzione di ordine superiore**. Java mantiene il proprio modello di tipi: il valore è riferito tramite un'interfaccia funzionale. Parlare di “funzioni di prima classe” è utile per descrivere la possibilità di spostare il comportamento, purché non faccia dimenticare il tipo concreto richiesto dal compilatore.

Che cosa rende funzionale un'interfaccia

Il conteggio riguarda i metodi astratti *da implementare*, non tutte le righe che sembrano metodi. Un metodo `default` ha già un corpo. Un metodo `static` appartiene all'interfaccia e non viene implementato dalla lambda. I metodi pubblici di `Object` dichiarati in un'interfaccia, come `equals(Object)`, non introducono una seconda funzione indipendente. Una gerarchia può ereditare dichiarazioni equivalenti dello stesso metodo astratto; il compilatore ne verifica la compatibilità. Non conviene imparare una formula superficiale come “una sola parola `abstract` nel file”: il criterio è che esista un solo contratto astratto funzionale effettivo.

Supponiamo di estendere due interfacce, una con `void invia(String testo)` e l'altra con `boolean verifica(String testo)`. L'interfaccia figlia richiederebbe due comportamenti e non può essere implementata da una sola lambda. Se invece entrambe ereditano la stessa firma `void invia(String testo)`, le dichiarazioni compatibili possono rappresentare un unico metodo funzionale. Conviene lasciare che `@FunctionalInterface` lo controlli: il compilatore conosce tutte le regole di override, inclusi i tipi di ritorno e le eccezioni, meglio di una conta manuale.

Un numero eccessivo di metodi `default` può nascondere un’interfaccia diventata troppo ampia. La cautela resta utile, ma un metodo `default` ben scelto non è un errore: `Function.andThen`, `Predicate.and` e `Consumer.andThen` sono precisamente metodi di composizione che rendono utilizzabile il contratto. La domanda da fare è se quel metodo appartiene al significato della funzione e se mantiene chiaro ciò che la lambda deve fornire. Se due interfacce ereditate definiscono lo stesso metodo `default` con corpi in conflitto, bisogna risolvere esplicitamente l'ambiguità; l'annotazione funzionale non decide quale corpo scegliere.

La forma di una lambda, letta senza scorciatoie

La freccia `->` separa parametri e corpo. `() -> "pronto"` non riceve argomenti e restituisce una stringa; `testo -> testo.length()` riceve un parametro; `(a, b) -> a + b` ne riceve due. Le parentesi intorno a un solo parametro inferito possono essere omesse. Un corpo formato da una sola espressione usa il valore di quell'espressione come risultato quando il contratto richiede un valore. Se il corpo è un blocco fra graffe e il contratto richiede un risultato, occorre un `return` su ogni percorso che può terminare normalmente.

```
BinaryOperator<Integer> sommaBreve = (a, b) -> a + b;
BinaryOperator<Integer> sommaEstesa = (a, b) -> {
    int risultato = a + b;
    return risultato;
};
```

La seconda forma non è sbagliata: può essere necessaria per controlli o passaggi che meritano di restare visibili. Se la lambda cresce fino a contenere una logica di dominio complessa, estrarre un metodo con un nome chiaro aiuta sia il lettore sia il debugger. Viceversa, togliere le graffe non è possibile se restano più istruzioni. Una lambda destinata a `Consumer<String>` può chiamare `System.out.println(testo)` senza restituire un valore, perché il metodo astratto `accept` ha ritorno `void`. La stessa espressione non può essere assegnata senza verifica a `Function<String, String>`, che promette una stringa come risultato.

L'espressione `x++` merita attenzione: come espressione restituisce il **valore precedente** di `x`, mentre `++x` restituisce quello successivo. In una lambda `UnaryOperator<Integer> incremento = x -> x++;`, il parametro `x` è una variabile locale della chiamata, perciò la riassegnazione non modifica l'oggetto `Integer` del chiamante; il valore restituito è addirittura quello iniziale. Per comunicare l'intenzione di incrementare, `x -> x + 1` è più corretto e più leggibile. L'analisi della sintassi deve arrivare fino al comportamento osservabile, altrimenti una riga "compatta" insegna la regola sbagliata.

Una lambda può lanciare eccezioni compatibili con la dichiarazione del metodo astratto. `Function.apply` non dichiara eccezioni checked, quindi una lambda assegnata a `Function` non può propagare direttamente, per esempio, una `IOException` senza gestirla. Un'interfaccia propria con `throws IOException` rende invece quel contratto esplicito. Non bisogna trasformare ogni eccezione checked in una `RuntimeException` solo per entrare in uno stream: prima si decide dove l'errore deve essere trattato e se la pipeline è davvero la forma adatta al problema.

Il catalogo di `java.util.function`, a partire dalla domanda

Quando cerchiamo un'interfaccia funzionale, chiediamoci prima che cosa entra e che cosa esce. `Supplier<T>` non riceve input e fornisce un `T` tramite `get`: può rappresentare una fabbrica di valori o il rinvio di un calcolo. `Consumer<T>` riceve un `T` ed esegue un'azione con `accept`, senza promettere un risultato. `Predicate<T>` riceve un `T` e risponde `true` o `false` tramite `test`. `Function<T, R>` riceve un `T` e restituisce un `R` tramite `apply`. Quattro forme rispondono già a molte necessità quotidiane. Nel programma di esempio `Supplier<String> etichetta = () -> "Java"` produce il testo, mentre `Predicate<String> lungo` decide se ha almeno quattro caratteri: la stampa `true` verifica che risultato e condizione non sono la stessa specie di comportamento.

> **Box – Dominio e codominio.** In matematica una funzione associa a ciascun elemento del dominio un risultato nel codominio. In `Function<String, Integer>`, `String` indica il tipo degli ingressi e `Integer` quello delle uscite. La firma Java non esprime da sola tutte le precondizioni: una funzione che chiama `Integer.parseInt` accetta formalmente `String`, ma alcune stringhe causano un errore. Il contratto didattico deve quindi spiegare anche il sottoinsieme di input validi e la gestione dei casi non validi.

Quando ci sono due ingressi, le famiglie `BiFunction<T, U, R>`, `BiConsumer<T, U>` e `BiPredicate<T, U>` esplicitano il secondo tipo. `BinaryOperator<T>` è un caso di `BiFunction` nel quale i due ingressi e il risultato hanno lo stesso tipo. `UnaryOperator<T>` è il caso di `Function<T, T>`. Nel nostro programma `BinaryOperator<Integer> operatore = Integer::sum` produce `13` passando `10` e `3`. Se i numeri devono restare primitivi e l'operazione è davvero numerica, `IntBinaryOperator` evita la conversione implicita fra `int` e `Integer`: il suo metodo si chiama `applyAsInt`. La scelta fra due tipi non si riduce alla lunghezza della dichiarazione; parte dalla semantica e, quando serve, dai costi misurati.

`BiFunction<Integer, Integer, String>` potrebbe prendere due interi e restituire una frase; `BinaryOperator<Integer>` non potrebbe, perché il risultato deve restare `Integer`. Queste differenze di tipo aiutano il compilatore a segnalare un'associazione problema-soluzione sbagliata. Se proviamo a usare un `Consumer` per calcolare un valore, non c'è un risultato da assegnare. Se proviamo a usare un `Supplier` per una validazione che dipende da un parametro, manca l'input. È più facile ricordare le interfacce partendo da questi vincoli che da un catalogo alfabetico di nomi.

Le varianti primitive coprono combinazioni frequenti: `IntSupplier` restituisce un `int`; `IntConsumer` lo riceve; `IntPredicate` lo valuta; `IntFunction<R>` riceve un `int` e restituisce un oggetto; `ToIntFunction<T>` riceve un oggetto e restituisce un `int`. `Long` e `Double` hanno famiglie analoghe. Il nome indica la direzione della conversione. Per esempio, `ToIntFunction<String> lunghezza = String::length` descrive da una stringa a un `int`. `Function<String, Integer>` è anch'essa valida ma usa il wrapper `Integer`; su milioni di elementi la distinzione può interessare. Sul singolo elemento, l'obiettivo è anzitutto avere un contratto comprensibile.

Comporre significa controllare l'ordine

`Function.andThen` crea una funzione che applica prima quella a sinistra e poi quella passata come argomento. `compose` fa il contrario: applica prima la funzione ricevuta e poi quella su cui è chiamato il metodo. Se `pulisci` elimina gli spazi e `maiuscolo` converte le lettere, `pulisci.andThen(maiuscolo)` rende esplicito l'ordine: pulisci, poi converti. `maiuscolo.compose(pulisci)` esprime lo stesso ordine da un altro punto di partenza. `Function.identity()` restituisce il valore che riceve e serve soprattutto quando una API richiede una funzione ma non vogliamo trasformare l'elemento.

Per vedere che l'ordine non è un dettaglio, prendiamo `aggiungiPrefisso = testo -> "#" + testo` e `rimuoviPrimo = testo -> testo.substring(1)`. Applicare il prefisso e poi rimuovere il primo carattere riporta il testo di partenza; invertire i passaggi elimina invece il primo carattere originale e aggiunge `#`. Le funzioni composte non diventano automaticamente commutative perché sono pure. Prima di

comporre, seguiamo un input concreto attraverso i passaggi e scriviamo l'output atteso. È il metodo "mattone dopo mattone" applicato alla pipeline che incontreremo nel capitolo 18.

`Predicate.and` e `Predicate.or` combinano condizioni con la normale logica booleana e arresto anticipato: nel caso di `and`, se la prima condizione è falsa, la seconda non serve. `negate` rovescia il risultato. Un esempio: un titolo è valido se `nonVuoto.and(abbastanzaBreve).test(titolo)` è vero. Se `nonVuoto` esclude `null` con `testo != null`, il secondo predicato può chiamare `length()` senza errore, purché l'ordine delle due condizioni rimanga quello voluto. Un nome come `titoloValido` racconta al lettore la regola complessiva; i predicati interni ne spiegano i pezzi.

`Consumer.andThen` esegue due azioni nell'ordine indicato. Qui gli effetti contano: stampare e poi salvare non è sempre equivalente a salvare e poi stampare. Se la prima azione lancia un'eccezione, la seconda non viene eseguita. Non esiste una "purezza automatica" delle interfacce funzionali: è il corpo concreto della lambda o del metodo referenziato a determinare se modifica stato, fa I/O o dipende dall'ora. Questo chiarisce anche perché `Supplier<Integer> casuale = ...` è un supplier legittimo pur non essendo una funzione pura: il valore può cambiare a ogni chiamata.

Il tipo bersaglio decide la forma della lambda

L'espressione `testo -> !testo.isBlank()` assume un significato preciso perché viene assegnata a `Predicate<String>`. Da quel tipo, il compilatore ricava che `testo` è una `String` e che il risultato deve essere un `boolean`. Se assegnassimo la stessa espressione a un'interfaccia con una firma incompatibile, la compilazione fallirebbe. Le parentesi intorno ai parametri possono essere omesse nel caso di un solo parametro con forma semplice; con due parametri, come nel `BiConsumer` dell'esempio, la lista va fra parentesi. Quando il corpo contiene più istruzioni, servono parentesi graffe e un `return` esplicito se la funzione produce un valore.

Un metodo sovraccarico che accetta due interfacce funzionali diverse può rendere ambigua una lambda: la stessa forma potrebbe adattarsi a entrambe. In quel caso non bisogna inventare una regola di precedenza basata sul nome dell'interfaccia. Un cast esplicito, una variabile intermedia con tipo dichiarato o API meglio distinte possono chiarire la scelta. È un motivo in più per progettare le firme pubbliche pensando anche a chi passerà lambda come argomenti.

Le interfacce funzionali standard non dichiarano in genere eccezioni checked nel loro metodo astratto. Se un'operazione passata a `Function` deve leggere un file e può lanciare `IOException`, non possiamo semplicemente far propagare quell'eccezione checked attraverso `apply`. Occorre gestirla nel corpo, scegliere un'interfaccia propria che dichiari `throws` oppure spostare la lettura fuori dalla trasformazione. Catturare l'eccezione e sostituirla con un valore vuoto senza spiegazione cancellerebbe un errore reale. Le regole del capitolo 12 valgono anche quando il codice è scritto in una lambda.

Funzioni che restituiscono funzioni

Una funzione può restituire un'altra funzione. Per esempio, un metodo potrebbe ricevere una soglia e restituire un `Predicate<Integer>` che controlla se un numero la supera. La funzione restituita cattura il valore della soglia e potrà essere usata più tardi. Questa è una forma concreta di funzione di ordine superiore: un comportamento è passato o restituito come valore tipizzato. Non occorre introdurre una nuova grammatica del linguaggio; si usano interfacce funzionali, generics e lambda già viste.

Se scrivessimo `Predicate<Integer> maggioreDiDieci = maggioreDi(10);`, il nome del metodo racconta che cosa viene costruito e il tipo racconta quale domanda potremo porre. Un'espressione annidata di lambda potrebbe essere più compatta, ma meno leggibile. Un nome che descrive l'operazione aiuta chi legge il codice a riconoscerla e a riusarla; ridurre il numero di caratteri non basta a rendere chiaro il programma.

> **Approfondimento – Currying.** In teoria, il *currying* trasforma una funzione di più argomenti in una catena di funzioni ciascuna con un argomento. In Java possiamo rappresentare l'idea con `Function<A, Function<B, R>>`, ma questa forma non è automaticamente migliore di `BiFunction<A, B, R>`. Diventa utile solo quando fissare il primo argomento crea un comportamento riusabile e leggibile. È un concetto facoltativo per il percorso principale.

Una trasformazione leggibile

Il programma completo prepara alcuni titoli: elimina gli spazi ai bordi, converte in maiuscolo, scarta il testo vuoto e stampa soltanto i risultati abbastanza lunghi. Non è un problema che richieda necessariamente le lambda; è piccolo proprio per isolare le decisioni. La figura 16.1 mostra le due trasformazioni e il filtro come passaggi distinti.

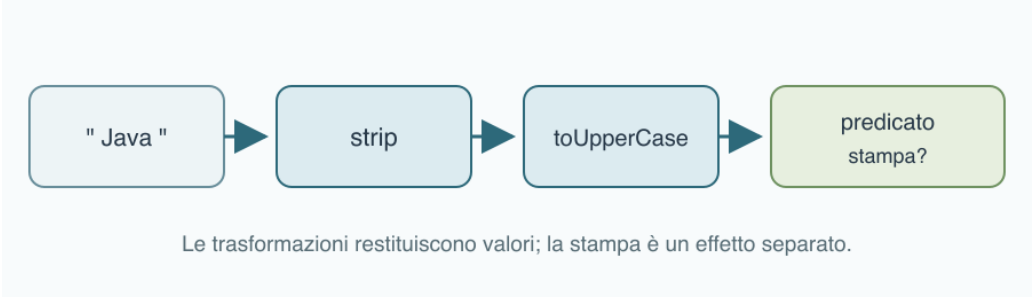


Figura 16.1 – Una funzione pulisce il testo, un'altra lo converte, un predicato decide se usarlo. La stampa resta un effetto separato dalla trasformazione.

```
Function<String, String> pulisci = String::strip;
UnaryOperator<String> maiuscolo = String::toUpperCase;
Function<String, String> prepara = pulisci.andThen(maiuscolo);
Predicate<String> nonVuoto = testo -> !testo.isBlank();
```

`Function<String, String>` descrive una funzione che riceve una stringa e ne restituisce una. `String::strip` è un **method reference**: dice di usare il metodo `strip` della stringa ricevuta. È leggibile qui perché il nome del metodo comunica già l'operazione. La forma lambda equivalente sarebbe `testo -> testo.strip()`. `UnaryOperator<String>` è una specializzazione di `Function<String, String>` per input e output dello stesso tipo. `andThen` costruisce una nuova funzione: prima esegue `pulisci`, poi `maiuscolo`. Se invertissimo l'ordine in questo caso otterremmo lo stesso testo per molti input, ma non è una proprietà generale delle funzioni: l'ordine va scelto in base al significato delle trasformazioni.

`Predicate<String>` rappresenta una domanda con risposta booleana. Nel codice, `testo -> !testo.isBlank()` è una lambda con un parametro. La parola a sinistra della freccia è una variabile locale della lambda; l'espressione a destra produce il risultato. Potremmo combinare predicati con `and`, `or` e `negate`, ma una catena lunga di condizioni anonime diventerebbe difficile da spiegare. Quando la condizione ha un nome del dominio, un metodo ordinario con quel nome può essere la scelta più chiara.

La stampa è affidata a un `BiConsumer<String, Integer>`, che riceve testo e lunghezza e non restituisce un valore. Nel programma viene invocato con `accept`. `Consumer`, `BiConsumer`, `Supplier`, `Predicate`, `Function`, `UnaryOperator` e le varianti primitive sono famiglie di interfacce in `java.util.function`. Una variante come `IntPredicate` evita il boxing quando il dato è un `int`; non occorre però sostituire meccanicamente ogni tipo con una variante primitiva. Prima si chiarisce quale relazione fra input e output serve.

Il programma stampa `JAVA: 4` e `120`. Il titolo `" Go "` diventa `GO`, ma non raggiunge la lunghezza minima di 3. La stringa di soli spazi diventa vuota e viene scartata. Il valore `120` proviene dalla seconda parte dell'esempio, dedicata alla ricorsione.

Seguire i tipi attraverso una trasformazione completa

Il programma `DemoFunzioni.java` contiene in poche righe quasi tutte le idee del capitolo. Rileggiamolo come leggeremmo il percorso di un dato in una applicazione reale. L'input `" Java "` entra in `pulisci`, che restituisce `"Java"`; entra poi in `maiuscolo`, che restituisce `"JAVA"`. Il predicato `nonVuoto` risponde `true`. La condizione sulla lunghezza confronta 4 con `lunghezzaMinima`, che vale 3, e lascia eseguire `stampa.accept`. La stampa `JAVA: 4` è un effetto collaterale dichiarato dalla scelta di `BiConsumer`. Il programma distingue così tra **trasformazione** di un valore, **decisione** booleana e **azione** che interagisce con l'esterno.

Seguiamo invece `" Go "`: dopo `strip` otteniamo `"Go"`, poi `"GO"`; il predicato non vuoto è ancora vero, ma la lunghezza 2 non raggiunge la soglia 3. La stringa formata da un solo spazio diventa vuota già nella prima trasformazione e viene fermata dal predicato. Se modificassimo il programma per controllare la lunghezza *prima* di `strip`, gli spazi conterebbero e la decisione potrebbe cambiare. La posizione di un filtro nella pipeline non è solo una possibile ottimizzazione: può essere parte del significato. Per questo formuliamo il contratto con parole precise, per esempio “titolo pulito di almeno tre caratteri”, prima di scegliere l'ordine delle chiamate.

Possiamo anche sostituire la soglia locale con una funzione restituita da un metodo: `Predicate<String> lungoAlmeno(int minimo) { return testo -> testo.length() >= minimo; }`. Se chiediamo `lungoAlmeno(3)`, la soglia è fissata

mentre il testo varierà a ogni chiamata di `test`. La firma del metodo racconta due tempi diversi: prima configuriamo la regola, poi la applichiamo. È un caso in cui la closure è utile per modellare una scelta, non soltanto per mostrare un meccanismo del linguaggio. Se il valore minimo deve cambiare durante l'esecuzione, invece, creare un nuovo predicato o usare un oggetto con stato esplicito sarà più onesto che tentare di aggirare la regola `effectively final`.

Un contratto funzionale per validare, non solo per calcolare

Immaginiamo che un sistema accetti nomi utente. Le regole richieste oggi sono “non vuoto”, “almeno quattro caratteri” e “senza spazi”. Possiamo rappresentarle con tre `Predicate<String>` e comporle con `and`. Prima, però, dobbiamo decidere che cosa fare con `null`: chiamare `isBlank()` su `null` lancia un'eccezione. Un primo predicato `testo -> testo != null` protegge quelli successivi grazie all'arresto anticipato di `and`. A quel punto l'insieme delle condizioni può essere verificato su `null`, `" "`, `"anna"` e `"an na"`. Quattro dati scelti per coprire casi differenti sono più istruttivi di una sola stringa valida.

```
Predicate<String> presente = testo -> testo != null;
Predicate<String> nonVuoto = testo -> !testo.isBlank();
Predicate<String> lungo = testo -> testo.length() >= 4;
Predicate<String> senzaSpazi = testo -> !testo.contains(" ");
Predicate<String> nomeValido = presente.and(nonVuoto).and(lungo).and(senzaSpazi);
```

La regola non è perfetta: `contains(" ")` controlla soltanto lo spazio ordinario, non ogni carattere di spaziatura. Se il requisito è “nessun carattere whitespace”, va scelta una verifica diversa e vanno definiti i caratteri che il dominio considera proibiti. Questa osservazione è un esempio di trasferimento: la composizione funziona, ma il problema reale chiede una definizione più accurata. È anche il motivo per cui una regola di business complessa può meritare un metodo o un oggetto nominato, invece di restare un concatenamento anonimo che nessuno sa aggiornare.

Il risultato di una funzione dipende da tutto ciò che legge

La definizione di funzione pura sembra semplice finché i parametri sono numeri immutabili. Prendiamo `int totale(List<Integer> valori)`: se il metodo legge la lista e ne calcola la somma senza modificarla, è puro rispetto a una **fotografia stabile** della lista. Ma se un altro codice può modificare la stessa lista fra due chiamate, i parametri hanno lo stesso riferimento e tuttavia contenuti diversi. Dire soltanto “stessi argomenti” diventa ambiguo. Per ragionare sulla purezza dobbiamo considerare lo stato osservabile raggiungibile attraverso gli argomenti. Una copia immutabile costruita con `List.copyOf` può dare un input più stabile, purché anche i suoi elementi non portino stato mutabile rilevante.

La trasparenza referenziale è una prova mentale utile su un'espressione concreta. `calcolaIVA(100)` può essere sostituito con il suo risultato solo se aliquota, arrotondamento e altri dati letti sono stabiliti dagli argomenti o da valori immutabili noti. Se il metodo consulta una configurazione modificabile o la data corrente, due chiamate apparentemente uguali possono produrre risultati diversi. Non dobbiamo eliminare tali funzioni da un programma: dobbiamo rendere visibile la dipendenza, per esempio passando aliquota e data come argomenti, e lasciare

che il livello esterno si occupi di leggere configurazione e orologio. Così il nucleo del calcolo torna verificabile con esempi ripetibili.

> **Box – Effetto non significa errore.** Scrivere un file, inviare una richiesta HTTP o aggiornare un contatore sono effetti necessari in molte applicazioni. Il problema è nasconderli dentro una funzione chiamata `normalizzaTesto` o in un predicato che sembra limitarsi a rispondere vero/falso. Un nome e un tipo adeguati aiutano, ma occorre anche un contratto: quando avviene l'effetto, quante volte può avvenire e come si segnala un fallimento. Queste domande diventano decisive se il comportamento è consegnato a un'altra API che controlla il momento della chiamata.

Purezza, immutabilità ed effetti

Una funzione **pura** produce lo stesso risultato per gli stessi argomenti e non cambia uno stato osservabile all'esterno. `testo -> testo.strip()` è un buon esempio: `String` è immutabile, e la trasformazione restituisce un valore. La stampa, invece, è un **effetto collaterale** intenzionale: scrive sull'output del programma. Non va trattata come un difetto da eliminare sempre; va collocata dove il lettore può vederla e dove non sorprende chi riusa la funzione.

La **trasparenza referenziale** è la possibilità, in un dato contesto, di sostituire una chiamata con il suo risultato senza cambiare il comportamento del programma. Se `somma(5, 5)` è pura e restituisce 10, usare 10 al suo posto non cambia il risultato. Se la funzione incrementa un contatore globale ogni volta che viene chiamata, la sostituzione perderebbe quell'effetto e non sarebbe equivalente. Nel codice Java, questa prova ci aiuta a riconoscere una dipendenza nascosta prima che renda un calcolo difficile da prevedere.

> **Nota bene – Immutabilità del riferimento e dell'oggetto.** Un riferimento `final` non può essere riassegnato, ma l'oggetto a cui punta può ancora essere modificabile. Per una funzione pura non basta scrivere `final`: occorre capire quali oggetti vengono letti o mutati e quali effetti possono essere osservati da fuori.

Le funzioni pure sono più semplici da provare e comporre perché non dipendono dall'ordine di chiamate non dichiarate. Tuttavia un'applicazione deve leggere dati, registrare eventi e comunicare con l'esterno. Il criterio pratico è tenere le trasformazioni pure dove aiutano e confinare gli effetti in punti riconoscibili, non sostenere che Java o gli stream eliminino automaticamente gli effetti.

La cattura delle variabili locali

Nel programma, la lambda non legge `lunghezzaMinima` direttamente, perché il confronto è nel ciclo; potremmo però scrivere `Predicate<String> abbastanzaLungo = testo -> testo.length() >= lunghezzaMinima;`. La lambda **cattura** allora il valore della variabile locale. Java richiede che una variabile locale catturata sia `final` o **effectively final**: può mancare la parola `final`, ma dopo l'assegnazione iniziale non deve essere riassegnata. Se scrivessimo `lunghezzaMinima++` più avanti nello stesso metodo, il compilatore rifiuterebbe la cattura.

La regola non rende automaticamente immutabile l'oggetto catturato. Una lambda può avere un riferimento finale a una lista e modificarne gli elementi, ma questa possibilità va usata con giudizio: rende il risultato dipendente da uno stato esterno e può creare problemi nei flussi paralleli. Una **closure**, nel senso utile qui, è un comportamento che conserva l'accesso ai valori del contesto in cui è stato creato. In Java occorre leggere questo concetto insieme alla regola delle variabili locali catturate e alla distinzione fra riferimento e oggetto.

Le lambda possono apparire dentro cicli, chiamate e assegnazioni. Non sono sempre la forma più chiara. Se il corpo ha molti rami, più effetti o un significato che merita un nome, un metodo ordinario spesso riduce il lavoro del lettore. Un method reference è utile quando sostituisce una lambda ovvia; non vale la pena usarlo se costringe a decifrare una firma senza contesto.

La cattura: un comportamento che conserva il contesto

Torniamo al metodo `aggiungi(int incremento)` del programma. `incremento` è un parametro locale: la sua normale visibilità termina quando il metodo ritorna. Eppure l'`Operazione` restituita può essere eseguita dopo quel ritorno. La lambda continua a usare il valore necessario al suo calcolo. Questa è la situazione concreta per cui parliamo di **closure**: un comportamento porta con sé ciò che gli serve del contesto lessicale in cui è stato definito. Non occorre immaginare che tutto lo stack del metodo rimanga vivo. Occorre capire quale valore è stato catturato e quali regole rendono sicura quella cattura.

```
static Operazione aggiungi(int incremento) {
    return (a, b) -> a + b + incremento;
}

Operazione conDue = aggiungi(2);
int risultato = conDue.esegui(10, 3); // 15
```

`incremento` è effectively final: dopo aver ricevuto il valore, nel metodo non viene riassegnato. Se aggiungessimo `incremento++;` dopo la definizione della lambda, il compilatore rifiuterebbe il programma. La regola vale anche se la riassegnazione è in un ramo che ci sembra improbabile: il compilatore controlla la struttura del metodo. Per una variabile locale catturata, la lambda osserva il valore necessario alla propria esecuzione e non una casella locale che il chiamante possa cambiare più tardi. La differenza emerge soprattutto quando il comportamento sopravvive al metodo che lo ha creato.

Le variabili di istanza seguono un'altra strada. Nel programma `DemoCattura.java`, l'oggetto possiede il campo `totale`; il metodo `prepara(3)` restituisce un `IntSupplier` che somma 3 al campo ogni volta che viene invocato. Le due chiamate stampano `3` e `6`. L'incremento locale è stabile, ma il campo dell'oggetto cambia: la lambda usa il medesimo oggetto attraverso `this`. Questa è una **lambda con effetto**. È legale e talvolta utile, ma le chiamate non sono intercambiabili: la seconda dipende dalla prima. In presenza di più thread servirà anche una regola di sincronizzazione, tema del capitolo sulla concorrenza.

```
IntSupplier prossimo = conto.prepara(3);
System.out.println(prossimo.getAsInt()); // 3
System.out.println(prossimo.getAsInt()); // 6
```

Un campo statico è accessibile allo stesso modo in cui lo è in un metodo ordinario, se la visibilità lo consente. Una lambda può leggerlo e modificarlo; ciò non la rende pura e non fa scattare la regola `effectively final` delle variabili locali. Tuttavia, il fatto che una modifica compili non significa che sia una buona scelta. Un contatore globale usato in un filtro rende il risultato dipendente dall'ordine di valutazione e difficile da riprodurre. Per questo, quando una lambda viene passata a una pipeline, è necessario dichiarare se osserva soltanto l'elemento corrente oppure usa uno stato che cambia.

> **Box – `final` non congela una lista.** Scriviamo `final List<String> nomi = new ArrayList<>();` e poi `Consumer<String> aggiungi = nomi::add;`. Il riferimento `nomi` non può essere riassegnato, ma la lista può essere modificata. La lambda è valida e produce un effetto. Se lo stesso consumer fosse chiamato da più thread senza coordinamento, la parola `final` non proteggerebbe la struttura interna della lista. La protezione richiesta dipende dall'oggetto condiviso e dalle sue regole, non dall'immutabilità del riferimento locale.

Lambda e classe anonima: che cosa indica `this`

Una classe anonima è un nuovo oggetto con il proprio `this`; una lambda usa il `this` della classe che la contiene. Possiamo verificarlo senza affidarci a una definizione astratta. Una classe `Etichetta` ha un campo `nome = "esterna"`. All'interno di un suo metodo, creiamo una classe anonima con un proprio campo `nome = "anonima"` e un metodo che stampa `this.nome`. Poi creiamo una lambda che stampa `this.nome`. La prima stampa è `anonima`, la seconda `esterna`. La lambda non crea un nuovo ambito per `this` come lo crea la classe anonima.

```
Runnable anonima = new Runnable() {
    private final String nome = "anonima";
    @Override public void run() { System.out.println(this.nome); }
};
Runnable lambda = () -> System.out.println(this.nome);
```

Questo frammento è valido dentro un metodo di istanza di `Etichetta`; non è pensato per stare in un metodo `static`, dove non esiste un `this` dell'istanza esterna. Anche `super` in una lambda si risolve nel contesto lessicale che la racchiude. Un'altra differenza pratica è che una classe anonima può implementare un'interfaccia con più metodi, mentre una lambda richiede un tipo bersaglio funzionale. Non si deve trasformare meccanicamente ogni classe anonima in lambda solo perché il corpo visibile è breve: controlliamo campi, metodi, `this`, `super` e identità dell'oggetto.

Quando confrontiamo lambda e classi anonime, dobbiamo distinguere il comportamento garantito dai dettagli di prestazione e allocazione: la specifica definisce il comportamento osservabile delle lambda, non promette una strategia fissa di allocazione valida per tutte le JVM. La scelta fra classe anonima e lambda

parte dal contratto e dalla leggibilità. Se un profilo reale dimostra un costo importante in un punto caldo, si misura quella implementazione sulla JVM e sul carico pertinenti. In assenza di tali dati, un'affermazione assoluta sulla velocità insegna una scorciatoia ingannevole.

Riferimenti a metodi: quattro problemi, quattro forme

Quando la lambda passa i propri argomenti a un metodo esistente senza aggiungere una logica propria, possiamo spesso usare `::`. Il riferimento a metodo conserva il tipo bersaglio: `System.out::println` non è un tipo autonomo, ma può essere assegnato per esempio a `Consumer<String>`. Le quattro forme più utili si distinguono da **chi possiede il metodo** e da **dove viene preso il ricevitore**.

La prima forma, `NomeClasse::metodoStatico`, usa un metodo statico. `Integer::sum` può implementare `BinaryOperator<Integer>` perché riceve due `Integer` che Java converte ai parametri `int` del metodo e restituisce un risultato adattabile a `Integer`. Nel nostro programma `DemoPercorsoLambda::somma` implementa l'interfaccia propria `Operazione`. Le firme non devono coincidere come stringhe di testo: il compilatore applica le regole di compatibilità dei parametri e del risultato, compresi adattamenti permessi come boxing e unboxing. Se i tipi non si accordano, l'errore avviene alla compilazione.

La seconda forma, `oggetto::metodoIstanza`, ha già un ricevitore. `System.out::println` conserva il riferimento a `System.out` usato per costruire quel method reference; il parametro che arriverà a `Consumer.accept` diventerà l'argomento di `println`. Un esempio di dominio potrebbe essere `catalogo::registra` assegnato a un `Consumer<Libro>` se `catalogo` è un oggetto con il metodo `registra(Libro)`. Qui il ricevitore è scelto quando prepariamo il comportamento. Se `catalogo` è `null`, la costruzione del riferimento a metodo di istanza produce un errore: non equivale a una chiamata posticipata su un riferimento forse nullo.

La terza forma, `NomeClasse::metodoIstanza`, non possiede ancora il ricevitore. Il primo argomento dell'interfaccia funzionale diventa l'oggetto su cui chiamare il metodo. `String::length` può diventare `Function<String, Integer>`: quando `apply("Java")` viene chiamato, la stringa ricevuta è il ricevitore di `length()`. `String::startsWith` può diventare `BiPredicate<String, String>`: il primo argomento è il testo da interrogare, il secondo è il prefisso. Leggere i due ruoli nell'ordine giusto evita di invertire la domanda. La forma lambda equivalente, `(testo, prefisso) -> testo.startsWith(prefisso)`, è una prova semplice della corrispondenza.

La quarta forma, `NomeClasse::new`, richiama un costruttore. `Supplier<ArrayList<String>> fabbrica = ArrayList::new` produce una nuova lista a ogni `get()`, mentre `Function<String, StringBuilder> crea = StringBuilder::new` usa il costruttore che riceve una stringa. Possiamo anche usare un riferimento a costruttore di array, `String[]::new`, con un'interfaccia che riceve la dimensione e restituisce un array. Non è necessario memorizzare tutte le firme: guardiamo il tipo bersaglio, individuiamo gli argomenti disponibili e verifichiamo quale costruttore è applicabile.

> **Verifica guidata.** Per `BiPredicate<String, String> iniziaCon = String::startsWith`, `iniziaCon.test("Java", "Ja")` vale `true`; scambiando i due argomenti vale `false`. Per `Function<String, Integer> lunghezza = String::length`, `lunghezza.apply("Java")` vale 4. Se il riferimento a metodo rende meno visibile l'ordine degli argomenti, usiamo la lambda esplicita. La leggibilità è una parte del contratto didattico.

Un riferimento a metodo sovraccarico può essere risolto dal tipo bersaglio, ma talvolta più firme restano applicabili e il compilatore segnala un'ambiguità. L'ambiguità di overload è un problema di **compilazione**: il chiamante non può scegliere in modo univoco la firma. Se abbiamo due overload `processa(Callable<String>)` e `processa(Supplier<String>)`, l'espressione `() -> "ok"` è compatibile con entrambi. Possiamo usare un cast al tipo desiderato, ma nomi distinti per operazioni dal significato diverso sono di solito più chiari e più stabili.

Ricorsione: un problema che richiama sé stesso

`fattoriale` nel programma mostra la **ricorsione**: un metodo chiama sé stesso su un problema più piccolo. Il caso `n == 0` interrompe la discesa; per `n > 0`, il risultato è `n` moltiplicato per `fattoriale(n - 1)`. Per `5`, le chiamate arrivano a zero e risalgono producendo 1, 1, 2, 6, 24 e 120.

Disegniamo un triangolo fatto di righe di punti: permette di *vedere* che cosa cambia fra un caso e il successivo. Nella figura, `T(n)` è il totale dei punti dopo aver aggiunto la riga numero `n`. Non è l'area geometrica di un triangolo continuo; è il conteggio di unità discrete disposte a triangolo.

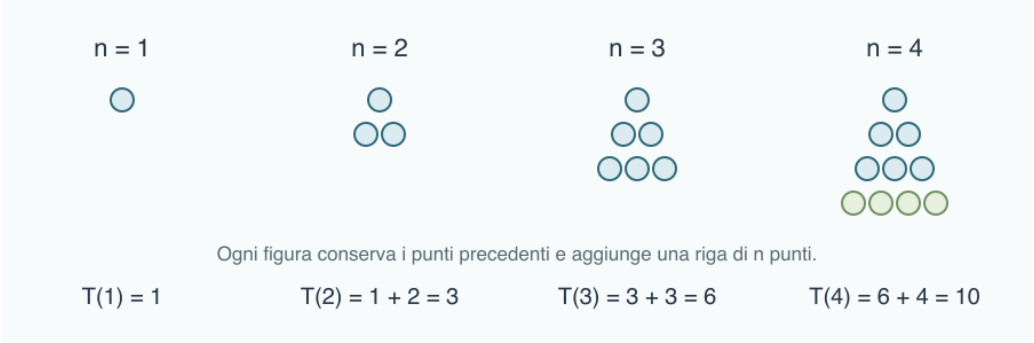


Figura 16.2 – Per ottenere `T(n)` si conservano i punti di `T(n − 1)` e si aggiunge una riga di `n` punti. Il colore diverso dell'ultima riga del quarto disegno mostra il passo ricorsivo.

Il caso iniziale è `T(0) = 0`: nessuna riga, nessun punto. Per un numero intero positivo `n`, la regola è `T(n) = T(n − 1) + n`. In questa formula `n` è il numero di punti nella nuova riga, `T(n − 1)` è il totale già costruito e il risultato è un conteggio di punti, senza unità fisiche. Seguendo i passi della figura otteniamo `T(1) = 1`, `T(2) = 1 + 2 = 3`, `T(3) = 3 + 3 = 6` e `T(4) = 6 + 4 = 10`. La formula vale qui per `n` interi non negativi; non descrive una misura continua né un ingresso negativo.

Il programma completo `DemoTriangolari.java` traduce direttamente la regola: per zero restituisce zero, altrimenti somma `n` al risultato di `triangolare(n - 1)`. Per

`triangolare(4)`, la JVM attende il risultato di 3, che attende quello di 2, poi di 1 e infine di 0. Risalendo, le somme producono 1, 3, 6 e 10; sono proprio le quattro righe stampate da `main`. `Math.addExact` segnala un overflow di `int`, ma non impedisce l'esaurimento dello stack per valori molto grandi: questi sono due limiti diversi. Una versione con ciclo può calcolare lo stesso totale con profondità di chiamata costante. Anche la formula chiusa `n(n + 1)/2` richiede attenzione all'overflow del prodotto prima della divisione; la scorciatoia algebrica non elimina automaticamente i limiti dei tipi Java.

I due esempi controllano rischi diversi: `DemoTriangolari` rifiuta un valore negativo e usa `Math.addExact` nella somma ricorsiva; il metodo `fattoriale` di `DemoFunzioni` usa `Math.multiplyExact` nel prodotto. In entrambi i casi un'eccezione per overflow evita un risultato intero silenziosamente errato. Non basta però a rendere la ricorsione adatta a numeri enormi: molte chiamate annidate consumano stack, e un caso senza arresto può causare `StackOverflowError`. Java non promette una trasformazione generale della ricorsione di coda in un ciclo. Per somme e prodotti lineari molto grandi, un ciclo può essere più semplice e più prudente.

La composizione incontrata con `andThen` costruisce una trasformazione da operazioni più piccole; la ricorsione costruisce una soluzione da problemi più piccoli. In entrambi i casi il passo va letto e verificato: nella composizione conta l'ordine delle funzioni, nella ricorsione conta il caso che ferma le chiamate. `compose` applica le funzioni nell'ordine opposto rispetto a `andThen` nella scrittura. Il confronto prepara il controllo del costo della ricorsione, senza trasformare le due tecniche in sinonimi.

Ricorsione: costi, limiti e alternative

La figura 16.2 ci ha dato il caso base `T(0) = 0` e il passo `T(n) = n + T(n - 1)` per `n > 0`. Ora guardiamo il costo di quelle chiamate. Se manca il caso base o se il passo non vi si avvicina, le chiamate continuano fino a esaurire lo stack. Se `n` è negativo, la regola matematica scelta nel capitolo non lo definisce: il programma deve rifiutare quell'ingresso invece di scendere indefinitamente.

Per `T(4)`, la pila delle chiamate contiene nell'ordine `T(4)`, `T(3)`, `T(2)`, `T(1)`, `T(0)`. Poi i risultati risalgono: 0, 1, 3, 6, 10. La JVM conserva il punto in cui ogni chiamata deve riprendere e i suoi dati locali. Questo spiega perché la memoria usata cresce con la profondità, anche se ogni riga di codice è breve. Nel programma `DemoTriangolari.java`, `Math.addExact` intercetta il superamento del limite di `int`; non intercetta uno stack troppo profondo. Sono due problemi separati: uno riguarda il **valore** calcolato, l'altro la **struttura** dell'esecuzione.

Una versione iterativa mantiene un accumulatore: parte da zero e somma `1, 2, ... n`. Per `n = 4` ottiene gli stessi valori intermedi della risalita ricorsiva senza mantenere quattro chiamate in attesa. La formula chiusa `T(n) = n(n + 1)/2` si ricava accostando due triangoli uguali: insieme formano un rettangolo di lati `n` e `n + 1`, quindi un triangolo ne contiene metà dei punti. Non conviene tradurre la formula ingenuamente in `int` con `n * (n + 1) / 2`, perché il prodotto intermedio può andare in overflow anche quando un ragionamento algebrico permetterebbe di dividere prima. Per input grandi scegliamo un tipo adeguato, controlliamo le operazioni e consideriamo i limiti di memoria e tempo. L'obiettivo qui è imparare a scegliere una rappresentazione, non dichiarare la ricorsione sempre migliore o sempre peggiore.

La funzione `fattoriale` del primo programma ha un altro caso base, `0! = 1`, e per `n > 0` usa `n! = n × (n-1)!`. Nel codice `Math.multiplyExact` avvisa quando un risultato non entra più in `int`. Scriviamo una piccola tabella a mano: `0! = 1`, `1! = 1`, `2! = 2`, `3! = 6`, `4! = 24`. Poi confrontiamola con l'esecuzione. Questa verifica è più istruttiva di una stampa finale isolata, perché ci dice *dove* il risultato potrebbe divergere dalla regola. Il trasferimento è un metodo che attraversa un albero: anche lì serve un caso base, per esempio un nodo assente, e un passo che visita sottoalberi più piccoli.

Applicazione parziale e currying, quando servono davvero

Supponiamo che un negozio debba applicare spesso uno sconto del 10 per cento. Una funzione generale `calcola(prezzo, percentuale)` riceve due valori; una funzione `scontoDieci` conserva la percentuale e riceve soltanto il prezzo. In Java possiamo costruirla con un metodo che restituisce `UnaryOperator<BigDecimal>` o con un'interfaccia propria, purché le regole monetarie siano definite: arrotondamento, scala e significato della percentuale. Il vantaggio è che la regola “10 per cento” diventa un comportamento nominato che può essere riutilizzato nei punti del programma in cui cambia solo il prezzo.

Il **currying** in senso stretto trasforma una funzione di due argomenti in una funzione che ne riceve uno e restituisce un'altra funzione: `Function<A, Function<B, R>>`. La **applicazione parziale** fissa invece alcuni argomenti di una funzione e ne produce una più specifica. Nell'uso quotidiano Java, i due concetti sono spesso avvicinati, ma non sono identici. Il metodo `aggiungi(2)` è un esempio pratico di fissare un valore nel comportamento restituito; non abbiamo bisogno di fingere che ogni metodo a più argomenti debba essere riscritto come una catena di funzioni. `BiFunction` è spesso la firma più chiara per un'operazione binaria ordinaria.

```
static Function<Integer, Integer> sommaCon(int incremento) {
    return valore -> valore + incremento;
}
Function<Integer, Integer> sommaTre = sommaCon(3);
int sette = sommaTre.apply(4);
```

Seguiamo il dato: `sommaCon(3)` non calcola ancora il risultato finale; prepara una funzione che ricorda 3. `apply(4)` fornisce il secondo valore e produce 7. Possiamo verificare `sommaTre.apply(10) == 13` senza ricreare la funzione. Il trasferimento a una regola di validazione è analogo: `lunghezzaAlmeno(8)` può restituire un `Predicate<String>` che controlla molte password candidate secondo una soglia scelta una volta. Se la regola non è riusata o la chiamata annidata diventa meno leggibile di un semplice metodo con due argomenti, il passaggio intermedio non aggiunge valore.

Pipeline e monadi: una distinzione necessaria

Chiamare “monade” qualsiasi classe di funzioni concatenabili è una semplificazione troppo larga: molte funzioni si possono concatenare senza formare una monade. Per il nostro percorso è sufficiente comprendere la **pipeline**: l'uscita di un passaggio diventa l'ingresso del successivo, con contratti che rendono possibile la composizione. `pulisci.andThen(maiuscolo)` è una pipeline di due trasformazioni; una pipeline Stream aggiunge una sorgente e una terminale. La concatenazione da sola non promette parallelismo: gli stream sequenziali eseguono il lavoro nel thread chiamante, e i passaggi possono essere fusi o ottimizzati dall'implementazione.

> **Approfondimento – Per chi incontrerà il termine “monade”.** In programmazione funzionale il termine riguarda un modo disciplinato di comporre calcoli che producono valori in un contesto, con operazioni e leggi precise. `Optional` rappresenta per esempio un possibile valore assente e permette di concatenare trasformazioni che rispettano quel contesto tramite `map` e `flatMap`. Ma conoscere il nome non è una precondizione per usare `Optional` correttamente; e dire che ogni pipeline è una monade sarebbe falso. Qui impariamo prima a riconoscere il problema concreto dell'assenza di valore e il contratto delle operazioni. I concetti teorici torneranno utili quando avremo casi reali con cui confrontarli.

`Optional`: un risultato che può mancare

Un metodo `trovaLibro(String codice)` può non trovare nulla. Restituire `null` obbliga ogni chiamante a ricordare una convenzione non espressa nel tipo; dimenticarla porta a `NullPointerException`. Restituire `Optional<Libro>` mette l'assenza nella firma. `Optional.empty()` rappresenta il caso assente; `Optional.of(libro)` richiede un valore non nullo; `Optional.ofNullable(libro)` accetta il caso nullo quando dobbiamo adattare codice esistente. Non usiamo `Optional.ofNullable` per nascondere indiscriminatamente errori di progettazione: lo usiamo quando l'assenza è un esito normale e dichiarato.

```
Optional<String> titolo = Optional.of(" Java ");
String visualizzato = titolo.map(String::strip)
        .filter(t -> !t.isEmpty())
        .orElse("senza titolo");
```

La `map` trasforma il valore se è presente e lascia vuoto l'`Optional` se non lo è. Il `filter` conserva il valore soltanto se passa la condizione. `orElse` fornisce un valore alternativo. Se l'alternativa richiede lavoro, `orElseGet(() -> calcolaAlternativa())` la calcola solo quando serve; l'argomento di `orElse`, essendo una normale espressione Java, viene invece valutato prima della chiamata. `orElseThrow` è opportuno quando l'assenza viola un contratto già controllato o quando un'eccezione esplicita comunica il problema. Chiamare `get()` senza verificare la presenza riporta il rischio al chiamante e perde gran parte della chiarezza ottenuta con il tipo.

`Optional` non è una collezione da usare per ogni campo o per ogni parametro. È particolarmente utile come valore di ritorno di una ricerca. Nel capitolo 18 incontreremo `findFirst` e `findAny`, che possono non trovare elementi e quindi restituiscono `Optional`. La domanda corretta non è “come estraggo subito il valore?”, ma “che cosa deve fare il programma quando il valore manca?”. Un catalogo può mostrare “non trovato”; un processo di pagamento può fermarsi con un errore; una ricerca esplorativa può proseguire. Queste sono decisioni del dominio, non dettagli dell'API.

Dai valori opzionali alle funzioni di trasformazione

Un `Optional<String>` può contenere il titolo trovato da una ricerca. `map(String::strip)` lo pulisce se è presente, mentre `flatMap` serve quando la funzione chiamata restituisce già un `Optional`. Supponiamo che `cercaAutore(String titolo)` restituisca `Optional<Autore>`. Se scriviamo `titolo.map(this::cercaAutore)`, otteniamo un `Optional<Optional<Autore>>`: un contenitore possibile intorno a un altro contenitore possibile. `flatMap(this::cercaAutore)` conserva un solo livello di assenza. Questo passaggio anticipa la stessa differenza fra `map` e `flatMap` negli stream: guardiamo sempre il tipo prodotto dalla funzione interna, non soltanto il nome del metodo esterno.

La scelta dell'alternativa in `Optional` è parte della soluzione. `orElse("sconosciuto")` è adatto se l'etichetta sostitutiva è già disponibile; `orElseGet(this::caricaEtichetta)` rimanda un lavoro potenzialmente costoso finché l'Optional non è vuoto. Se l'assenza è un errore, `orElseThrow(() -> new IllegalArgumentException("titolo assente"))` esplicita il fallimento. Se dobbiamo soltanto agire quando c'è un valore, `ifPresent` accetta un `Consumer`. La scelta non è “qual è il metodo più moderno?”, ma “quale risposta richiede il problema in presenza e in assenza?”.

Ci sono anche limiti deliberati: `Optional` non dovrebbe diventare automaticamente un campo di ogni entità, né un modo per accettare `null` ovunque. Alcune librerie di serializzazione e persistenza hanno convenzioni proprie; una firma con `Optional` non le sostituisce. Il valore maggiore è nella chiamata a un metodo che *può legittimamente non trovare* un risultato. Da qui il collegamento con `findFirst` nel capitolo successivo: l'assenza è un esito previsto e il chiamante deve gestirla.

Scelte di stile che proteggono il significato

Quando una lambda è lunga tre schermate, il lettore deve ricostruire insieme il significato dell'operazione e tutti i suoi rami. Un metodo nominato come `calcolaPrezzoFinale` può rendere la pipeline più chiara, purché il nome sia accurato e il corpo rimanga verificabile. Un riferimento a metodo va preferito quando spiega immediatamente l'azione; `String::strip` è chiaro, mentre una catena di riferimenti a metodi sovraccarichi con generics complessi può richiedere più sforzo della lambda esplicita. Il codice deve permettere a chi lo legge di seguire dato in ingresso, trasformazione e risultato senza indovinare.

Per una regola importante del dominio, una piccola interfaccia propria può essere migliore di `Predicate<Ordine>`: `RegolaSconto` o `ControlloOrdine` possono esprimere un'intenzione che il generico `test` non comunica. Tuttavia, creare una nuova interfaccia per ogni lambda di una riga produce molto codice senza

migliorare il modello. Prima proviamo le interfacce standard; se il nome o il contratto delle eccezioni non bastano, definiamo il tipo di dominio. Anche qui il percorso è problema, scelta, verifica: un chiamante deve capire dalla firma quale comportamento fornire e che cosa può aspettarsi in uscita.

L'overload di metodi che accettano interfacce funzionali simili richiede attenzione. `esegui(Supplier<String>)` e `esegui(Callable<String>)` ricevono entrambi una lambda senza parametri che produce una stringa; la chiamata `esegui(() -> "ok")` è ambigua. Un cast risolve il caso locale, ma due nomi diversi possono chiarire la diversa semantica delle operazioni. Inoltre, `Callable.call` può dichiarare `Exception`, mentre `Supplier.get` no: anche quando l'espressione sembra identica, il contratto dell'errore cambia. Questo è un buon esempio di come la firma funzionale non sia solo una comodità sintattica.

Per valutare se una lambda è adatta a un'API, chiediamoci se può essere richiamata zero, una o più volte, se l'ordine è garantito, se può essere eseguita in un altro thread e se il chiamante si aspetta una funzione senza effetti. `Comparator.compare`, per esempio, può essere richiamato molte volte da un ordinamento. Un comparatore che cambia risposta in base a un contatore produce un ordine incoerente. `Predicate` in uno stream può essere saltato in alcune ottimizzazioni. `Supplier` dato a una cache può essere invocato solo quando manca il valore. Capire queste condizioni vale più che imparare a memoria tutte le forme di `->`.

> **Verifica e trasferimento.** Riprendi il problema iniziale dei prezzi: scrivi una `Operazione` per la differenza, passala a `calcola` e verifica il risultato con valori scelti da te. Poi prova a sostituirla con `Integer::sum`; spiega perché il metodo chiamante non cambia. Infine trasporta il modello a una lista di titoli: il lavoro comune attraversa i titoli, il comportamento variabile decide quali accettare. Se la decisione deve consultare un archivio esterno, rendi visibile quell'effetto e spiega come verificherai l'errore di accesso.

Una prova di trasferimento: classificare documenti

Un archivio riceve descrizioni testuali e deve assegnare una categoria in base al contenuto. Possiamo separare tre responsabilità: una `Function<String, String>` pulisce il testo, un `Predicate<String>` verifica che sia utilizzabile e una `Function<String, Categoria>` lo classifica. Prima di scrivere lambda, decidiamo i casi: descrizione `null`, soli spazi, testo valido ma categoria sconosciuta. Se la classificazione può fallire senza che il dato sia scorretto, il tipo di ritorno potrebbe essere `Optional<Categoria>`. Se invece un servizio remoto decide la categoria, l'operazione porta I/O, latenza ed errori; nasconderla in una lambda chiamata `classifica` non elimina queste responsabilità.

La prima verifica può essere completamente locale: tre stringhe, tre risultati attesi, nessuna rete. La seconda può sostituire il classificatore con una implementazione finta che restituisce categorie note. Questo mostra il vantaggio di passare un comportamento come argomento: possiamo cambiare la regola senza duplicare il percorso di pulizia e validazione. Il trasferimento dal problema dei prezzi a quello dei documenti è quindi reale: la struttura stabile attraversa i dati, la parte variabile è un comportamento tipizzato e le verifiche controllano entrambe le parti.

Tre errori di lettura da riconoscere prima che diventino difetti

Il primo errore è confondere la **brevità sintattica** con la purezza. `x -> { contatore++; return x * 2; }` è una lambda breve, ma dipende da un campo esterno e lo modifica. `x -> x * 2` è una trasformazione senza quell'effetto. Quando leggiamo un comportamento passato come argomento, non fermiamoci alla freccia: individuiamo tutti i valori letti e scritti. Se è una regola di calcolo, chiediamoci se due chiamate con lo stesso input possono essere scambiate senza cambiare il risultato dell'applicazione. Se la risposta è no, documentiamo la dipendenza e verifichiamo l'ordine delle chiamate.

Il secondo errore è credere che `@FunctionalInterface` sia obbligatoria per usare una lambda. Un'interfaccia con un solo metodo astratto è funzionale anche senza annotazione. La annotiamo perché vogliamo proteggere l'intenzione del progetto: se l'interfaccia evolve, il compilatore deve avvertirci che le lambda esistenti perderebbero il loro contratto. È lo stesso motivo per cui `@Override` aiuta quando implementiamo un metodo: la dichiarazione esplicita permette al compilatore di controllare una scelta che altrimenti potrebbe restare soltanto nella nostra testa.

Il terzo errore è assegnare a un riferimento a metodo più chiarezza di quanta ne abbia. `String::replaceAll` richiede di sapere che il primo argomento della funzione bersaglio diventa la stringa su cui chiamare il metodo, mentre gli altri diventano gli argomenti della chiamata. Per un lettore che sta ancora imparando questo meccanismo, `(testo, espressione, sostituto) -> testo.replaceAll(espressione, sostituto)` può essere più chiaro. Inoltre `replaceAll` interpreta il primo parametro come espressione regolare, non come testo letterale; se il problema richiede una sostituzione letterale, `replace` è un'altra operazione. La forma più corta non corregge una scelta semantica sbagliata.

Verificare le funzioni con casi che discriminano

Per `sommaCon(3)`, proviamo zero, un numero positivo e uno negativo. Il risultato atteso è rispettivamente 3, 7 per input 4 e 1 per input -2. Questi casi verificano che il valore fissato venga davvero aggiunto e che non sia confuso con un moltiplicatore o con una soglia. Per `nomeValido`, proviamo `null`, stringa vuota, quattro lettere senza spazi e quattro caratteri con uno spazio. La prova non mira a ripetere il corpo della lambda: mira a rappresentare diverse classi di input del problema.

Quando una funzione viene composta con un'altra, aggiungiamo un caso in cui cambiare l'ordine cambia il risultato. Se due trasformazioni commutano per tutti i dati del test, un ordine errato passerebbe inosservato. Per questo l'esempio con il prefisso e la rimozione del primo carattere è didatticamente più forte del solo `strip` seguito da `toUpperCase`. La regola si trasferisce a conversioni di unità, normalizzazioni, calcoli fiscali e validazioni: scegliere un input che distingua due implementazioni plausibili rende la verifica informativa.

Nel capitolo 17 incontreremo le collezioni che conservano i dati; nel capitolo 18 useremo i comportamenti tipizzati di queste pagine nelle pipeline. La distinzione costruita qui resta valida in entrambi: una funzione descrive una trasformazione, mentre l'API che la riceve decide quando e quante volte invocarla.

Per verificare

Compila ed esegui `DemoFunzioni.java`, `DemoPercorsoLambda.java` e `DemoCattura.java` con JDK 25. Modifica `lunghezzaMinima` da 3 a 2 e prevedi quale nuova riga comparirà. Poi sostituisci `String::strip` con la lambda equivalente e verifica che l'output resti uguale. Prima di eseguire `DemoCattura`, prevedi perché la seconda chiamata dà un risultato diverso dalla prima. Per controllare la ricorsione, scrivi su carta le chiamate di `fattoriale(3)` prima di eseguirle. Infine prova `fattoriale(-1)` in una copia del programma e spiega quale contratto è stato violato.

17. Scegliere una collezione

Prima la domanda, poi la classe

La pila dei capitoli precedenti ci ha aiutato a capire come si costruisce una struttura dati. In un'applicazione Java, però, quasi sempre useremo le collezioni della libreria standard. Il **Java Collections Framework** raccoglie interfacce, implementazioni e algoritmi per gestire gruppi di oggetti. La sua ampiezza può invitare a memorizzare un catalogo di classi; è più utile partire dalle domande del problema. I duplicati sono ammessi? L'ordine conta? Accediamo per posizione, per chiave o da un'estremità? La collezione deve cambiare dopo la creazione?

Insieme, lista, pila, coda, deque e mappa rispondono a domande diverse. Prima di scegliere una classe concreta, correggiamo due scorciatoie frequenti. Una lista Java non è necessariamente una catena di nodi: `ArrayList` usa un array ridimensionabile, `LinkedList` usa collegamenti. Una `Map` appartiene al framework ma **non** estende `Collection`: associa chiavi a valori invece di contenere semplicemente elementi. La figura 17.1 confronta le due famiglie senza suggerire una falsa gerarchia unica.

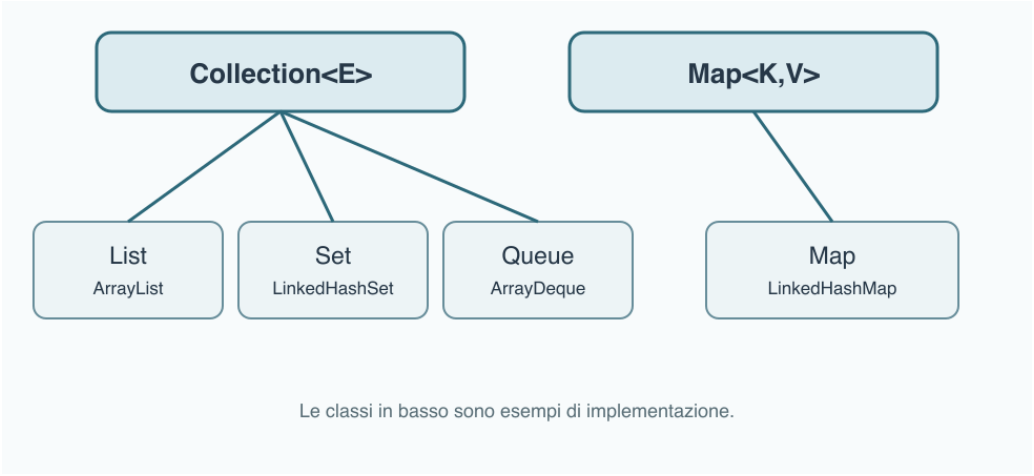


Figura 17.1 – `Collection` e `Map` rispondono a esigenze diverse. Le interfacce dichiarano il contratto; classi come `ArrayList`, `LinkedHashSet` e `LinkedHashMap` sono scelte concrete con proprietà proprie.

> **Definizione – Interfaccia e implementazione.** `List<String>` è un tipo di riferimento che promette operazioni di lista. `new ArrayList<String>()` crea un'implementazione concreta. Dichiarare una variabile con l'interfaccia rende visibile il contratto richiesto; scegliere la classe alla creazione stabilisce proprietà come costo delle operazioni e gestione dell'ordine.

Dalla pila costruita a una famiglia di soluzioni

Ripensiamo alla pila realizzata nei capitoli 7 e 8. Era utile perché ci costringeva a stabilire una regola: l'ultimo elemento inserito è il primo a uscire. Avevamo dovuto scegliere come memorizzare i dati, come riconoscere una pila piena e che cosa fare con una pila vuota. Ora il problema cambia. La biblioteca deve gestire non una sola struttura, ma titoli in sequenza, codici senza ripetizioni, prestiti in attesa e libri ricercabili per codice. Scrivere ogni volta una nuova classe significherebbe ripetere molto lavoro e moltiplicare gli errori. Il framework offre contratti comuni e implementazioni già pronte; a noi resta la responsabilità di scegliere il contratto adatto al problema.

Storicamente Java aveva già `Vector`, `Stack` e `Hashtable`, prima che il framework organizzasse le collezioni attorno a interfacce comuni. Da Java 5 i generics permettono inoltre di esprimere il tipo degli elementi, per esempio `List<Libro>`, senza affidare ogni lettura a un cast da `Object`. Incontreremo ancora quelle classi nei programmi esistenti, ma non dobbiamo usarle automaticamente per il codice nuovo. Il passaggio importante non è ricordare una data: è capire perché una famiglia di interfacce facilita il cambio di implementazione quando il bisogno resta lo stesso.

Considera `List<String> titoli = new ArrayList<>();`. La parte a sinistra dice che il codice che usa `titoli` chiede una sequenza indicizzata di stringhe; la parte a destra sceglie una rappresentazione con array ridimensionabile. Se domani cambiamo rappresentazione, possiamo farlo soltanto se il codice non dipende da proprietà specifiche di `ArrayList`. Dichiarare l'interfaccia non rende tutte le implementazioni equivalenti: ordine, mutabilità, ammissione di `null`, costo delle operazioni e comportamento concorrente restano scelte da verificare.

> **Per capire – Tre pezzi del framework.** Le *interfacce* definiscono il comportamento richiesto: `List` consente accesso per posizione, `Set` non ammette duplicati, `Map` associa chiavi e valori. Le *implementazioni* conservano i dati e realizzano quel comportamento: `ArrayList`, `HashSet` o `HashMap`, per esempio. Gli *algoritmi* lavorano sulle collezioni, come l'ordinamento di una lista tramite `Collections.sort`. Separare queste tre parti ci permette di chiederci prima **che cosa deve fare** la struttura, poi **come conviene realizzarla**.

Le operazioni comuni prima delle classi concrete

`Collection<E>` estende `Iterable<E>` ed è la radice della famiglia che comprende liste, insiemi e code. Per una collezione possiamo domandare `size()`, `isEmpty()` e `contains(elemento)`; possiamo attraversarla, aggiungere o togliere elementi se l'implementazione lo consente, oppure convertirne il contenuto in un array. `Map<K,V>` segue un ramo distinto: `size()` e `isEmpty()` hanno un significato analogo, ma un elemento della mappa è un'associazione e non un semplice `E`. Quando serve attraversare una mappa, otteniamo una vista delle chiavi, dei valori o delle coppie tramite `keySet()`, `values()` ed `entrySet()`.

Le operazioni che lavorano su gruppi interi sono importanti quanto quelle sul singolo elemento. `catalogo.containsAll(selezionati)` chiede se ogni elemento di `selezionati` compare nel catalogo; `addAll` trasferisce elementi da una raccolta

all'altra; `removeAll` elimina quelli presenti nell'altra; `retainAll` conserva soltanto quelli comuni. Il valore booleano restituito da queste modifiche indica se la collezione è cambiata, non quanti elementi sono stati mossi. Per esempio, chiamare `addAll` con un insieme di titoli già presenti in un `Set` lascia l'insieme uguale e restituisce `false`. Non tutte le collezioni permettono le modifiche: la firma del metodo esiste nell'interfaccia, ma una collezione non modificabile può lanciare `UnsupportedOperationException`.

> **Nota bene – Un tipo generico non promette mutabilità.** `Collection<Libro>` dice quale tipo di elemento possiamo leggere senza cast. Non dice che `add` riuscirà, che `null` sia ammesso o che l'iterazione abbia un ordine stabile. Queste promesse appartengono al contratto della collezione concreta e dell'API che ce la consegna. Per questo una funzione che riceve `Collection<Libro>` dovrebbe usare soltanto le operazioni di cui ha realmente bisogno e documentare se intende modificarla.

Per non confondere le strutture con il loro uso, pensiamo a due domande concrete. «Quali titoli compaiono nella lista di lettura, anche se ripetuti?» porta a una `List`. «Quali codici sono stati già visitati?» porta a un `Set`. La prima domanda attribuisce significato alla posizione e alla ripetizione; la seconda all'unicità. Solo dopo decidiamo se l'ordine d'incontro o la velocità di ricerca richiedano `ArrayList`, `LinkedHashSet` o un'altra implementazione. Questo ordine del ragionamento è un mattone che useremo anche quando arriveremo alle mappe.

Liste: sequenza, duplicati e posizione

Una `List<E>` mantiene una sequenza di elementi e può contenere duplicati. `get(indice)` legge una posizione; l'indice 0 è il primo elemento. Nel programma completo, `new ArrayList<>(List.of("Java", "Reti", "Java"))` crea una lista modificabile con tre elementi. L'output `[Java, Reti, Java]` conferma che il secondo `Java` non viene eliminato. Il costruttore riceve una lista iniziale, ma crea una nuova `ArrayList`: aggiungere o rimuovere elementi dalla nuova lista non cambia quella passata al costruttore.

`ArrayList` è spesso la prima scelta per una lista: l'accesso per indice è diretto e l'aggiunta in fondo è normalmente efficiente, pur potendo richiedere un ridimensionamento dell'array interno. Inserire o rimuovere vicino all'inizio richiede in genere lo spostamento di elementi. `LinkedList` ha una struttura diversa, ma non diventa automaticamente più veloce per ogni inserimento: raggiungere la posizione può costare una traversata e ogni nodo richiede memoria e collegamenti. Se il problema usa soprattutto le estremità, spesso conviene guardare a `Deque` prima di scegliere `LinkedList` per abitudine.

La documentazione Java 25 di `List` distingue le operazioni obbligatorie dalle operazioni di modifica che alcune implementazioni possono rifiutare. Un metodo che riceve `List` non può dedurre soltanto dal tipo statico se `add` sia consentito. Per questo il contratto dell'API che consegna la lista deve spiegare anche la sua mutabilità.

Due liste, stessa interfaccia, costi diversi

Una `ArrayList` conserva i riferimenti in un array che può crescere. `get(0)` o `get(500)` raggiunge direttamente la posizione richiesta: non deve attraversare i precedenti 500 elementi. Se aggiungiamo in fondo e la capacità interna basta, l'operazione è semplice; quando non basta, la lista deve predisporre più spazio e copiare i riferimenti. Questo costo occasionale spiega perché si parla di costo *ammortizzato* per le aggiunte in fondo: molte aggiunte economiche e qualche riallocazione più onerosa. Se inseriamo alla posizione zero, invece, gli elementi già presenti devono spostarsi per lasciare spazio. La capacità interna non coincide con `size()`: una lista appena creata può avere spazio disponibile pur contenendo zero elementi, e il valore preciso della capacità è un dettaglio di implementazione su cui il programma non deve fondarsi.

Una `LinkedList` conserva ogni elemento in un nodo collegato al precedente e al successivo. Aggiungere o togliere da un'estremità può evitare lo spostamento di molti riferimenti; raggiungere la posizione 500 richiede però di attraversare nodi. Anche l'inserimento «nel mezzo» è rapido solo **dopo** che il punto è stato raggiunto, per esempio con un `ListIterator` già collocato lì. Se per ogni inserimento chiamiamo `get(indice)` o `add(indice, elemento)` partendo dall'inizio, il costo della ricerca del punto può dominare. I nodi richiedono inoltre spazio e possono peggiorare la località dei dati in memoria. Non esiste dunque una regola «`LinkedList` è migliore quando si modifica una lista» senza dire da dove arriviamo al punto da modificare.

La figura 17.2 rende visibile perché le due liste hanno costi diversi: nell'array una posizione è individuata dall'indice, mentre nella lista collegata si seguono i riferimenti fra nodi.



Figura 17.2 – `ArrayList` raggiunge una posizione per indice; `LinkedList` deve attraversare i collegamenti per arrivare a un nodo interno. Il disegno rappresenta i riferimenti fra elementi, non le dimensioni fisiche in memoria.

Bisogno	`ArrayList`	`LinkedList`
Leggere spesso la posizione `i`	Accesso diretto; scelta naturale.	Richiede un percorso fra nodi.

Bisogno	`ArrayList`	`LinkedList`
Aggiungere in fondo	Normalmente economico; qualche crescita copia i riferimenti.	Aggiunge un nodo all'estremità.
Inserire nel mezzo conoscendo solo l'indice	Sposta una parte dei riferimenti.	Prima raggiunge il punto, poi modifica i collegamenti.
Usare una coda dalle estremità	Possibile con operazioni di lista, ma il tipo comunica poco l'intenzione.	Implementa anche `Deque`; spesso `ArrayDeque` è più semplice se non servono gli indici.

La tabella confronta *operazioni*, non dichiara un vincitore assoluto. Se la biblioteca mostra spesso la pagina 20 di un elenco, `ArrayList` è plausibile. Se ci serve una coda di richieste, il contratto giusto è `Deque` o `Queue`: la rappresentazione si sceglie dopo. `Vector` è un'altra lista con array ridimensionabile, nata prima del framework e con metodi storicamente sincronizzati. La sincronizzazione dei singoli metodi non risolve da sola un'operazione composta, come «controlla se il titolo manca e poi aggiungilo». Per il codice nuovo, scegliamo il contratto e la politica di concorrenza esplicitamente, senza usare `Vector` come scorciatoia.

Cercare e lavorare su una porzione

Una lista non serve soltanto per leggere un indice. `indexOf(titolo)` cerca la prima posizione di un elemento uguale secondo `equals`; `lastIndexOf(titolo)` cerca l'ultima. Se il titolo non c'è, il risultato è `-1`, quindi va controllato prima di usarlo come indice. `set(i, valore)` sostituisce l'elemento in posizione `i` se la lista supporta la modifica; `add(i, valore)` inserisce e sposta i successivi; `remove(i)` elimina e restituisce l'elemento che occupava quella posizione. Per una lista di stringhe, `remove(1)` sceglie l'indice 1, mentre `remove("Java")` sceglie la prima occorrenza del valore. Con una `List<Integer>` la distinzione fra indice e valore merita ancora più attenzione: `remove(1)` indica l'indice, `remove(Integer.valueOf(1))` indica il valore.

`subList(inizio, fine)` descrive una porzione con limite finale escluso, come `substring` per le stringhe. Da `List.of("A", "B", "C", "D")`, la porzione da 1 a 3 contiene `B` e `C`. È una **vista** della lista: non equivale a un nuovo archivio indipendente. Se il chiamante deve conservare una fotografia dei titoli mentre la lista originale continua a cambiare, crea una copia esplicita. Quando l'elemento è mutabile, anche una nuova lista copia soltanto i riferimenti agli elementi, non ogni oggetto contenuto: la differenza fra copia del contenitore e copia dei libri torna qui in forma molto concreta.

Insiemi: unicità e ordine

Un `Set<E>` non contiene due elementi uguali secondo il criterio di uguaglianza usato dall'implementazione. `HashSet` usa `hashCode` ed `equals`: il contratto del capitolo 10 diventa quindi concreto. Se due oggetti risultano uguali con `equals`, devono produrre lo stesso hash; altrimenti l'insieme può non riconoscere correttamente i duplicati. Non promettere un ordine di iterazione per `HashSet`. `LinkedHashSet`, invece, mantiene un ordine d'incontro definito. Nel programma, dalla lista con due `Java` otteniamo `[Java, Reti]`, poi la vista `reversed()` si legge come `[Reti, Java]`.

`TreeSet` mantiene gli elementi ordinati secondo il loro ordine naturale oppure un `Comparator`. Nel programma il comparatore ordina prima per lunghezza e poi alfabeticamente. I valori `API`, `Java` e `Reti` vengono stampati in quell'ordine. Qui la seconda parte del comparatore è essenziale: se confrontassimo soltanto la lunghezza, `Java` e `Reti` sarebbero giudicati equivalenti per il set, che ne conserverebbe soltanto uno. L'ordine di un `TreeSet` non è una decorazione successiva: decide anche quando due elementi occupano la stessa posizione logica nell'insieme.

`EnumSet` è un insieme specializzato per i valori di un enum: quando il dominio è proprio un enum, rende esplicito il tipo e offre un'implementazione dedicata. Non è necessario usarlo per ogni insieme; la scelta segue il tipo dei dati e le operazioni richieste.

> **Nota bene – Tre significati di «ordine».** Una lista ha posizioni e un ordine della sequenza. Un `LinkedHashSet` preserva un ordine d'incontro definito, ma non consente duplicati né accesso indicizzato. Un `TreeSet` ordina secondo un comparatore. Dire soltanto che una collezione «è ordinata» lascia aperta la domanda decisiva: *secondo quale regola?*

La figura 17.3 mette a confronto le promesse delle tre implementazioni principali di `Set`: tutte eliminano duplicati, ma rispondono a esigenze diverse di visita.

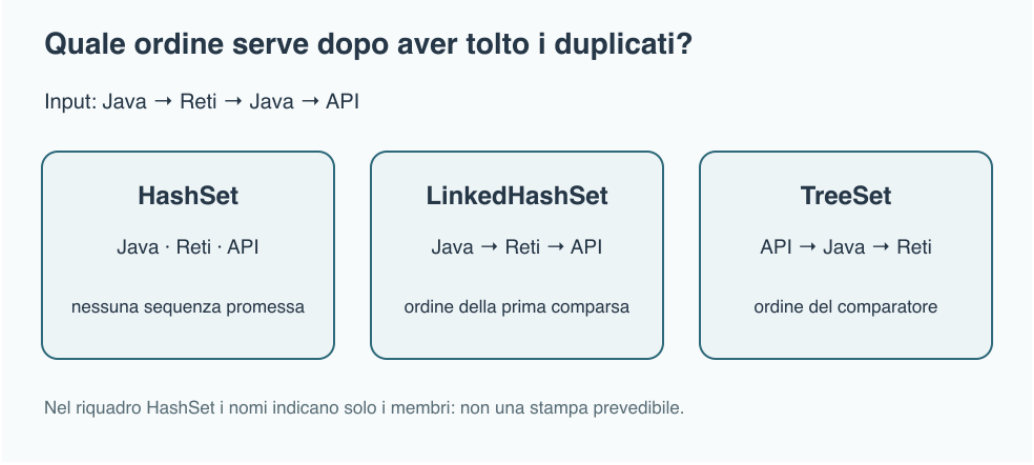


Figura 17.3 – `HashSet` non promette ordine, `LinkedHashSet` conserva la prima comparsa, `TreeSet` usa il comparatore. I nomi nel riquadro `HashSet` rappresentano i membri, non una sequenza di stampa.

Le operazioni sugli insiemi risolvono domande del dominio

Supponiamo di avere i codici dei libri posseduti da una biblioteca e quelli richiesti per una mostra. `posseduti.containsAll(richiesti)` risponde alla domanda «li abbiamo tutti?». Se vogliamo i codici che compaiono in almeno uno dei due gruppi, partiamo da una copia modificabile e chiamiamo `addAll`: è l'unione. Se vogliamo solo quelli comuni, chiamiamo `retainAll`: è l'intersezione. Se vogliamo i posseduti che non sono richiesti, chiamiamo `removeAll`: è la differenza. Le operazioni modificano il **set su cui sono invocate**; per conservare i due insiemi di partenza, creiamo prima una copia. Il risultato di `containsAll` è booleano e non cambia i dati.

> **Per capire – La notazione degli insiemi.** Se `A = {Java, Reti}` e `B = {Reti, API}`, allora l'unione `A ∪ B` contiene `Java`, `Reti` e `API`; l'intersezione `A ∩ B` contiene soltanto `Reti`; la differenza `A \ B` contiene soltanto `Java`. I simboli sono un modo breve per nominare una domanda concreta. In Java, su una copia modificabile di `A`, `addAll(B)`, `retainAll(B)` e `removeAll(B)` realizzano rispettivamente queste tre domande. Dopo ogni chiamata controlliamo quale insieme è cambiato: non vogliamo alterare per errore l'archivio originale.

Il fatto che `HashSet` non prometta ordine non significa che «ordini per hash»: l'ordine che osserviamo in una stampa può cambiare e non è un contratto da usare. `LinkedHashSet` preserva l'ordine d'incontro, utile quando vogliamo togliere duplicati da una lista senza perdere l'ordine della prima comparsa. `TreeSet` applica invece una regola di confronto: se la regola considera due elementi equivalenti, il secondo non trova una posizione distinta nell'insieme. Per questo un comparatore scelto soltanto per una bella stampa può cancellare informazione. Se il dominio considera diversi due titoli della stessa lunghezza, il comparatore deve saperli distinguere, per esempio aggiungendo il confronto alfabetico.

Un insieme costruito sui valori di un enum

Riprendiamo un enum `Giorno { LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI, VENERDI, SABATO, DOMENICA }`. Se una biblioteca apre dal lunedì al venerdì, `EnumSet.range(Giorno.LUNEDI, Giorno.VENERDI)` esprime la fascia inclusiva secondo l'ordine **di dichiarazione** delle costanti. `EnumSet.noneOf(Giorno.class)` crea l'insieme vuoto mantenendo noto il tipo; `allOf(Giorno.class)` prende tutti i giorni; `of(Giorno.SABATO, Giorno.DOMENICA)` sceglie due valori espliciti. `copyOf` parte da una collezione esistente, ma per una collezione vuota generica il tipo dell'enum non è ricavabile: in quel caso usiamo `noneOf` e aggiungiamo gli elementi. Queste factory evitano di rappresentare giorni inesistenti con stringhe digitate male.

`EnumSet` non accetta `null` ed è modificabile. Non diventa automaticamente sicuro per modifiche concorrenti solo perché è specializzato ed efficiente. Se l'orario di apertura dipende anche da festività, sede o stagione, un set statico dei giorni non basta: il modello dovrà includere quelle condizioni. Questo è il passaggio di generalizzazione: la soluzione `EnumSet` è adatta al problema «quali costanti dell'enum appartengono al gruppo?», non a qualunque regola temporale della biblioteca. La specifica Java 25 elenca le factory e le loro condizioni di errore.

Mappe: cercare per chiave

Una `Map<K,V>` associa una chiave `K` a un valore `V`. In un catalogo, la chiave potrebbe essere il codice del libro e il valore la sua scheda. Una chiave compare al più una volta; chiamare di nuovo `put` con la stessa chiave sostituisce il valore associato. `HashMap` offre ricerca basata su hash senza promettere un ordine d'iterazione. `LinkedHashMap` mantiene un ordine definito. `TreeMap` ordina le chiavi secondo un comparatore o il loro ordine naturale. `EnumMap` è una scelta specializzata quando le chiavi appartengono a un enum.

Nel programma usiamo `SequencedMap<String, Integer> prestiti = new LinkedHashMap<>();`. Dopo aver inserito `Java` e `Reti`, `firstEntry().getKey()` restituisce `Java`; la vista `reversed()` mostra `Reti` come prima voce. La parola **vista** conta: una vista espone gli stessi dati con una prospettiva diversa e può riflettere modifiche della mappa sottostante, secondo il contratto della specifica implementazione. Non va scambiata automaticamente per una copia indipendente.

Per `HashMap` le chiavi richiedono un `hashCode` coerente con `equals`, come per `HashSet`. Usare come chiave un oggetto mutabile e poi cambiare i campi coinvolti in uguaglianza e hash può rendere la voce difficile da ritrovare. Una chiave immutabile, come un record ben scelto, riduce questo rischio. Se la chiave può essere `null`, il comportamento va controllato nell'implementazione concreta; le factory `Map.of` non accettano chiavi o valori nulli.

La figura 17.4 riunisce le mappe più comuni: prima resta identica la regola delle chiavi uniche, poi cambia l’ordine con cui possiamo visitare le voci.



Figura 17.4 – `HashMap`, `LinkedHashMap`, `TreeMap` ed `EnumMap` distinguono quattro esigenze di visita. La figura non attribuisce un ordine a `HashMap`.

Dal codice del libro alla sua scheda

La domanda «il libro con codice J-17 è disponibile?» è diversa da «in quale posizione della lista si trova J-17?». Se il codice identifica univocamente il libro, una mappa descrive direttamente l'associazione. `put("J-17", scheda)` registra la voce; `get("J-17")` recupera il valore oppure `null` se non esiste una voce per la chiave. Se i valori nulli sono ammessi dalla mappa, un `null` restituito da `get` può anche significare che la chiave c'è ma il suo valore è nullo: `containsKey` distingue i due casi. Un'API di dominio può scegliere un modello ancora più chiaro, per esempio `Optional<Libro>` nel capitolo 19, se l'assenza è una risposta normale.

Una seconda `put` con la stessa chiave sostituisce il valore precedente e restituisce quel valore precedente, oppure `null` secondo le regole appena viste. La dimensione non aumenta. Questo è ciò che permette di aggiornare la scheda di un libro senza creare due identità concorrenti. I valori, invece, possono ripetersi: due codici diversi possono avere la stessa categoria o lo stesso autore. `containsValue` chiede se almeno una voce possiede quel valore, ma non è un sostituto di una ricerca per chiave quando il problema richiede prestazioni di accesso ripetuto.

`HashMap` è una scelta frequente quando non ci serve alcuna garanzia sull'ordine di visita. La stampa di quattro voci può apparire ordinata in un'esecuzione e diversa in un'altra: non dobbiamo dedurne che `HashMap` ordini le chiavi. `LinkedHashMap` conserva un ordine d'incontro definito, di norma quello d'inserimento: se inseriamo le chiavi `1, 3, 2, 4` e poi sostituiamo il valore di `4`, l'ordine resta `1, 3, 2, 4` nel caso ordinario. `TreeMap` ordina invece in base alle chiavi, usando il loro ordine naturale o un comparatore. Se i codici devono apparire sempre ordinati e il catalogo cambia di continuo, questo contratto può essere utile; se ci serve soltanto una stampa occasionale, potremmo ordinare una vista delle chiavi al momento della presentazione.

Domanda	Scelta iniziale	Garanzia da controllare
Trovo un libro dal codice senza un ordine di visita richiesto?	`HashMap`	Chiave stabile per `equals` e `hashCode`; accesso non sincronizzato.
Devo ricordare l'ordine di registrazione?	`LinkedHashMap`	L'ordine promesso è d'incontro, non automaticamente alfabetico.
Devo attraversare sempre le chiavi in ordine?	`TreeMap`	Il comparatore deve distinguere le chiavi che il dominio considera diverse.
Le chiavi sono costanti di un enum?	`EnumMap`	Ordine delle costanti e divieto di chiave `null`.

La tabella è una guida di scelta, non una misura delle prestazioni. Una mappa non corregge una chiave progettata male. Se una classe `CodiceLibro` usa campi modificabili per `equals` e `hashCode`, cambiarli dopo l'inserimento può impedire di ritrovare la voce. Un record che conserva un codice normalizzato è più facile da usare correttamente. Anche l'ordine di `TreeMap` deve essere coerente con il significato di identità scelto: se due codici diversi vengono confrontati come uguali, una mappa ordinata può trattarli come la stessa chiave.

Quando la chiave è una costante

Se registriamo per ogni `StatoPrestito` il numero di prestiti presenti, le chiavi appartengono a un insieme finito noto al compilatore. `EnumMap<StatoPrestito, Integer> conteggi = new EnumMap<>(StatoPrestito.class)` crea una mappa vuota del tipo corretto. Possiamo aggiungere una voce con `put`, rimuoverla con `remove` e creare una nuova mappa da una `EnumMap` esistente. Durante l'iterazione le chiavi seguono l'ordine di dichiarazione dell'enum, non l'ordine in cui abbiamo chiamato `put`. Una chiave `null` è vietata; un valore `null` è invece ammesso, anche se per un conteggio è più chiaro decidere se l'assenza significhi zero oppure «non ancora rilevato».

> **Nota bene –** `EnumMap` **non è un enum.** L'enum definisce le possibili chiavi; la mappa associa a ciascuna chiave un valore che può cambiare. `EnumSet` risponde alla domanda «quali stati sono presenti?», `EnumMap` alla domanda «quale valore corrisponde a ogni stato?». Questa differenza è il criterio per trasferire la soluzione a un problema nuovo. Le regole Java 25 di `EnumMap` precisano anche che le sue viste hanno iteratori debolmente consistenti: non lanciano `ConcurrentModificationException`, ma possono mostrare o non mostrare una modifica intervenuta durante il percorso. Non è una licenza per modificare la mappa da più thread senza coordinamento.

Pile, code e due estremità

Una **pila** estrae per primo l'ultimo elemento inserito: LIFO. Una **coda** estrae per primo il più vecchio: FIFO. `Deque<E>` supporta operazioni da entrambe le estremità e può rappresentare entrambi i comportamenti. `ArrayDeque` è una scelta ordinaria per una pila o una coda non concorrente. Nel programma chiamiamo `push("primo")`, poi `push("secondo"); pop()` stampa `secondo`. È lo stesso comportamento della pila didattica, ma con un tipo generico della libreria. La vecchia classe `Stack` esiste per compatibilità; per nuovo codice non è la scelta da insegnare come pila predefinita.

`Queue<E>` esprime la coda. I metodi `add`/`remove` e `offer`/`poll` differiscono nel modo in cui segnalano certe condizioni di fallimento o assenza; il chiamante deve conoscere il contratto dell'implementazione. `PriorityQueue` non conserva il FIFO generale: estrae secondo una priorità, definita dall'ordine naturale o da un comparatore. Il nome «queue» non basta a prevedere quale elemento uscirà.

Le interfacce **sequenced**, disponibili in Java 25, modellano collezioni con un ordine d'incontro definito e operazioni sulle estremità. `SequencedCollection`, `SequencedSet` e `SequencedMap` permettono anche una vista `reversed()`. La specifica API di `SequencedCollection` chiarisce il contratto. Non tutte le collezioni sono sequenced: chiedere il primo elemento di un `HashSet` come se avesse un ordine promesso sarebbe un errore di modello.

La figura 17.5 separa la classe `Stack`, che si può incontrare in programmi esistenti, dal contratto `Deque` che useremmo per una nuova pila.

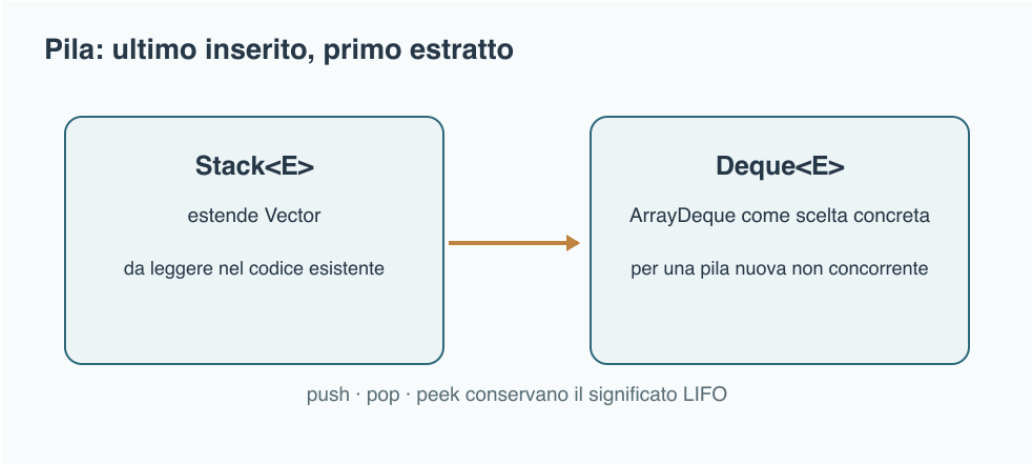


Figura 17.5 – Il comportamento LIFO resta lo stesso; nel codice nuovo `Deque` *esprime meglio la scelta della struttura.* `ArrayDeque` *è una possibile implementazione non concorrente.*

La pila della libreria e la pila del libro

La nostra `Pila` didattica aveva capacità fissa e metodi scelti per capire invarianti ed errori. `java.util.Stack<E>` esiste ancora: estende `Vector`, fornisce `push`, `pop`, `peek`, `empty` e `search`. `search(oggetto)` restituisce la distanza dalla cima contando da **uno**, non l'indice di una lista; se l'oggetto non compare, restituisce `-1`. Saper leggere una `Stack` è utile in codice esistente. Per nuovo codice la documentazione di `Deque` raccomanda il contratto `Deque`, spesso realizzato da `ArrayDeque`. `push` inserisce in testa, `pop` toglie dalla testa e `peek` osserva la testa senza rimuoverla: la stessa regola LIFO, espressa con una struttura più adatta al bisogno.

Non dobbiamo però confondere questa pila di oggetti con lo *stack* delle chiamate mostrato dalle eccezioni. Una `Deque<Libro>` è un contenitore che il nostro programma modifica; lo stack delle chiamate è parte del contesto di esecuzione di un thread. Il nome comune nasce dall'ordine LIFO, ma leggere `pop()` nella nostra collezione non rimuove un metodo dalla pila delle chiamate. La distinzione evita di trasferire un'analogia utile oltre il suo ambito.

Tre modi di chiedere il primo elemento di una coda

Per una coda di richieste, `add(r)` e `offer(r)` provano entrambe a inserire. La prima segnala con un'eccezione il rifiuto dovuto a limiti di capacità; la seconda può restituire `false`. `remove()` e `poll()` estraggono il primo elemento: su una coda vuota, `remove()` lancia un'eccezione e `poll()` restituisce `null`. `element()` e `peek()` guardano il primo senza toglierlo, con la stessa differenza fra eccezione e `null` quando la coda è vuota. La scelta dipende da come vogliamo rappresentare il caso vuoto nel contratto del chiamante. Non serve memorizzare sei nomi isolati: ricordiamo le tre operazioni — inserire, estrarre, osservare — e le due forme di segnalazione.

Con `ArrayDeque` non si inseriscono valori `null`, perciò `null` restituito da `poll` ha un significato non ambiguo. Una `PriorityQueue` risponde a una domanda diversa: «quale richiesta ha la priorità maggiore secondo un criterio?» e non «quale è

arrivata prima?». Se due richieste hanno la stessa priorità e l'ordine di arrivo conta, il comparatore deve includere anche un numero progressivo o un altro criterio di spareggio. Altrimenti il programma non dovrebbe promettere FIFO all'interno dello stesso livello di priorità.

Modificabile, non modificabile e immutabile

`List.of`, `Set.of` e `Map.of` creano collezioni non modificabili. `List.copyOf` produce una lista non modificabile dai valori di un'altra collezione; nel programma la copia dei tre titoli ha dimensione 3. Queste factory rifiutano `null`; `Set.of` rifiuta anche elementi duplicati già nella creazione. **Non modificabile** significa che non possiamo aggiungere, sostituire o rimuovere elementi attraverso quella collezione. Non significa che gli oggetti contenuti siano immutabili: se una lista contiene un oggetto mutabile, quello può ancora cambiare.

`Collections.unmodifiableList(lista)` crea invece una vista non modificabile della lista ricevuta: attraverso la vista non si può chiamare `add`, ma modifiche fatte alla lista originale possono apparire nella vista. Una copia non modificabile e una vista non modificabile rispondono quindi a esigenze differenti. Quando un'API promette uno snapshot, deve produrre una copia adeguata e considerare anche la mutabilità degli elementi, se rilevante.

`Arrays.asList` permette di costruire una lista in poche parole. È una scorciatoia utile, ma proprio la sua brevità può nascondere una differenza importante. Passando un array, otteniamo una lista **a dimensione fissa collegata a quell'array**: `set` può sostituire un elemento e il cambiamento appare nell'array, mentre `add` e `remove` non possono cambiare il numero degli elementi. Questa lista non è né una normale `ArrayList` ridimensionabile né una copia immutabile. Se il programma deve poter aggiungere titoli, costruiamo per esempio `new ArrayList<>(Arrays.asList(dati));` se deve conservare una fotografia non modificabile dei riferimenti correnti, scegliamo `List.copyOf` quando i valori non sono nulli.

Il programma `DemoVisteListe.java` fa cambiare un solo elemento e osserva tre percorsi. `fissa` modifica l'array; `copia` mantiene i riferimenti presenti quando è stata costruita; `vista` impedisce la modifica attraverso il proprio riferimento ma continua a mostrare il contenuto della lista sottostante. Usiamo stringhe, che sono immutabili, per isolare il comportamento della struttura. Con oggetti modificabili, anche una copia della lista potrebbe mostrare lo stato mutato di un oggetto condiviso: una copia della struttura non è automaticamente una copia profonda degli elementi.

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class DemoVisteListe {
    public static void main(String[] args) {
        String[] dati = {"Java", "Reti"};
        List<String> fissa = Arrays.asList(dati);
        List<String> copia = List.copyOf(fissa);
        List<String> vista = Collections.unmodifiableList(fissa);

        fissa.set(1, "API");
        System.out.println("Array: " + Arrays.toString(dati));
        System.out.println("Copia: " + copia);
        System.out.println("Vista: " + vista);
        try {

            fissa.add("JVM");
        } catch (UnsupportedOperationException errore) {
            System.out.println("Aggiunta: " + errore.getClass().getSimpleName());
        }
    }
}
```

Le righe attese sono `Array: [Java, API]`, `Copia: [Java, Reti]`, `Vista: [Java, API]` e `Aggiunta: UnsupportedOperationException`. Prima di eseguirlo, prova a prevederle: la differenza fra copia e vista diventa chiara solo se segui *quando* ciascuna osserva i dati. Ora cambia il requisito: un chiamante deve ricevere i titoli esistenti ma non deve vedere aggiunte future al catalogo. Una vista non modificabile non basta; occorre una copia con il contratto adatto. Se invece il chiamante deve osservare aggiornamenti in tempo reale senza poterli eseguire, la vista può essere appropriata, purché il codice dichiari che non è uno snapshot e definisca anche la disciplina di accesso concorrente.

Iterare e modificare con criterio

Un `Iterator<E>` permette di attraversare una collezione con `hasNext()` e `next()`. Il `for` esteso usa questo modello quando attraversa un `Iterable`. Se durante l'iterazione vogliamo rimuovere l'elemento corrente da una collezione che lo consente, possiamo usare `iterator.remove()` secondo il contratto, invece di modificare la collezione attraverso un altro riferimento nel mezzo del ciclo. Alcuni iteratori di implementazioni comuni rilevano modifiche strutturali inattese e lanciano `ConcurrentModificationException`; non sono però un meccanismo di sincronizzazione né una garanzia assoluta di individuazione in ogni circostanza. Nel capitolo 20 useremo `BlockingQueue` e `ArrayBlockingQueue` per coordinare un produttore e un consumatore: la coda consente di trasferire elementi fra thread e di attendere quando è piena o vuota.

> **Il nome e il problema.** Il pattern *Iterator* risponde alla domanda «come percorro gli elementi senza esporre la struttura interna della collezione?». Per questo, nella classificazione dei pattern, è un pattern **comportamentale**, non creazionale: creare l'oggetto iteratore è soltanto il modo per iniziare quel comportamento. L'interfaccia Java `Iterator<E>` applica questa idea con un contratto preciso; `Iterable<T>` è invece il contratto di chi sa fornire un nuovo iteratore. Non serve memorizzare l'etichetta prima di capire la coppia problema–soluzione: una collezione custodisce gli elementi, un iteratore governa un percorso.

Una piccola mappa di scelta riassume le domande del capitolo senza sostituire la spiegazione. Se servono posizioni e duplicati, si parte da `List` e spesso `ArrayList`; se serve unicità, da `Set`; se serve ricerca per chiave, da `Map`; se servono estremità, da `Deque`; se serve priorità, da `PriorityQueue`. Poi si decide l'implementazione chiedendo quali garanzie di ordine, modifica, nullità e costo siano necessarie. Il costo di un'operazione dipende anche dalla dimensione, dalla distribuzione delle chiavi, dal comparatore e dal modo d'uso: non basta uno slogan come «la mappa è sempre veloce».

Il catalogo della biblioteca, decisione per decisione

Proviamo a scegliere una collezione senza guardare prima l'elenco delle classi. Il catalogo deve trovare un libro dato il suo codice univoco. La domanda è una ricerca per chiave: `Map<CodiceLibro, Libro>` descrive bene il contratto. Se il codice è una stringa normalizzata, la mappa può usare `String` come chiave; se ha più parti, un record immutabile può riunirle. La scelta fra `HashMap`, `LinkedHashMap` e `TreeMap` dipende poi da una seconda esigenza: non ci serve un ordine, vogliamo quello d'inserimento o ci occorre quello dei codici? Scegliere un `TreeMap` soltanto perché «ordina» aggiunge un criterio che forse il catalogo non richiede.

Ora vogliamo mostrare gli ultimi prestiti in ordine di registrazione, anche se due prestiti riguardano lo stesso titolo. Una `List<Prestito>` conserva duplicati e posizione; una `Set<Prestito>` potrebbe eliminarne alcuni secondo `equals`, perdendo informazione. Se i prestiti crescono senza limite, una singola lista in memoria può non essere il modello completo del sistema: la collezione rappresenta il tratto caricato dall'applicazione, non sostituisce un archivio persistente. Le API di collezione non decidono da sole quanto storico conservare o come proteggere dati concorrenti.

Per tenere i codici dei libri già esposti in una vetrina, senza duplicati e nell'ordine in cui sono stati aggiunti, `LinkedHashSet<String>` combina le due proprietà. Se l'ordine non conta, `HashSet` esprime meno garanzie e può essere sufficiente. Se la vetrina deve mostrarsi sempre in ordine alfabetico, `TreeSet` o un ordinamento al momento della stampa possono essere valutati in base a quanto spesso cambiano i dati e quanto spesso vengono letti. Il criterio di confronto deve restare coerente con il significato di duplicato voluto dal dominio.

Per le richieste di prestito in attesa, una coda FIFO è naturale se l'ordine di arrivo decide chi viene servito per primo. Una `ArrayDeque<Richiesta>` può esprimere questo comportamento nel singolo thread. Se entrano richieste da più thread contemporaneamente, non basta cambiare il nome della variabile: serve una struttura concorrente e un contratto sull'ordine e sulla sincronizzazione, da

affrontare nel capitolo dedicato. Se alcune richieste hanno priorità, `PriorityQueue` può rappresentare la regola, ma occorre definire come gestire parità e arrivi successivi.

Questo percorso mostra perché il tipo della collezione non è soltanto una scelta di prestazioni. È parte del significato del programma: una lista dice che le ripetizioni contano, un set dice che no, una mappa identifica valori mediante chiavi, una deque espone le estremità. Dopo aver espresso il significato, si misurano i costi se il carico reale lo richiede. Non si sceglie `LinkedList` perché «inserire è veloce» senza specificare *dove*, *come si raggiunge il punto* e *quanti elementi ci sono*.

> **Buona pratica – Esporre soltanto la garanzia utile.** Un metodo che restituisce il catalogo non dovrebbe consegnare la `HashMap` interna modificabile se il chiamante deve soltanto leggere. Può restituire una vista o una copia con un contratto dichiarato. La scelta fra vista e copia determina se aggiornamenti successivi diventeranno visibili; va spiegata a chi usa l'API.

Attraversare non significa possedere

Proviamo il percorso di `Iterable` e `Iterator` con una lista di tre titoli: `Java`, `Reti`, `Java`. Chiamare `iterator()` crea un cursore per quell'attraversamento. `hasNext()` chiede se resta un elemento, `next()` lo restituisce e fa avanzare il cursore. Dopo tre chiamate valide, un'ulteriore `next()` non restituisce `null` come segnale di fine: lancia `NoSuchElementException`. Il `for (String titolo : titoli)` evita di scrivere a mano il cursore, ma percorre la stessa interfaccia `Iterable`. Non trasforma la lista in un array né produce una copia.

L'iteratore non è il proprietario dei dati: è un modo disciplinato di percorrerli. Su una lista modificabile, se vogliamo togliere durante il percorso ogni titolo vuoto, usiamo `iterator.remove()` dopo aver letto l'elemento, se l'implementazione consente quell'operazione. `ListIterator` aggiunge il movimento all'indietro e operazioni come `add` e `set` nel contesto di una lista. Queste capacità non appartengono a un semplice `Iterator` e non sono garantite da ogni collezione. Per una trasformazione di tutti i valori può essere più chiaro costruire una nuova lista; per una rimozione mirata `removeIf` esprime direttamente il predicato. La scelta dipende da quale risultato deve essere osservato dopo l'operazione.

Un cursore non è una copia della collezione

Immagina di segnare con un dito la riga che stai leggendo in un catalogo. Il dito non contiene il catalogo: indica soltanto dove ti trovi. Un `Iterator<E>` svolge questo ruolo. `hasNext()` controlla se un'altra lettura è possibile, `next()` restituisce l'elemento e avanza. Per questo `for (Libro libro : libri)` è comodo quando dobbiamo leggere tutti i libri, ma non quando serve una posizione precisa o un percorso all'indietro. `Iterable` è il contratto dell'oggetto che può fornire un iteratore; `Iterator` è il cursore ottenuto per un singolo percorso. Chiamare di nuovo `iterator()` crea un nuovo percorso, non riporta indietro quello già consumato.

La vecchia interfaccia `Enumeration<E>` compare ancora in alcune API e classi storiche. Offre `hasMoreElements()` e `nextElement()`, senza un metodo per

rimuovere l'elemento corrente. Se incontriamo codice che la restituisce, possiamo leggerlo come un cursore in avanti; `asIterator()` rende i suoi elementi rimanenti disponibili attraverso l'interfaccia `Iterator`. Non c'è motivo di introdurre `Enumeration` come prima scelta per una nuova collezione. Questo passaggio storico è utile perché i nomi delle API vecchie possono comparire in un progetto reale, anche se il percorso didattico usa il `for` esteso e `Iterator`.

Un `ListIterator<E>` aggiunge ciò che manca al cursore generico quando la struttura è una lista: `hasPrevious()` e `previous()` permettono il ritorno, mentre `nextIndex()` e `previousIndex()` indicano la posizione che verrebbe letta in ciascuna direzione. Il cursore si trova *fra* gli elementi. Se una lista contiene `A`, `B`, `C` e chiamiamo `listIterator(1)`, una chiamata a `next()` restituisce `B`, mentre `previous()` restituisce `A`. All'inizio `previousIndex()` vale `-1`; alla fine `nextIndex()` vale `size()`. Questi numeri sono posizioni di una prossima lettura, non elementi immaginari ai bordi della lista.

Quando l'implementazione consente modifiche, `ListIterator.add(e)` inserisce nel punto del cursore; `set(e)` sostituisce l'ultimo elemento restituito da `next()` o `previous()`, se la sequenza di chiamate lo permette. Non è corretto descrivere `set` come «sostituisce sempre l'elemento che verrà letto dopo»: il suo bersaglio dipende dalla **lettura appena compiuta**. Questa precisione serve quando trasformiamo una lista mentre la attraversiamo. Per una lista non modificabile, invece, i metodi di modifica possono lanciare `UnsupportedOperationException` anche se compaiono nella firma dell'interfaccia.

La modifica durante il percorso non è una gara fra thread

Un `ArrayList` può lanciare `ConcurrentModificationException` se, mentre lo percorriamo, cambiamo strutturalmente la lista attraverso un altro riferimento. L'eccezione aiuta a scoprire un errore di uso, ma non è una promessa di rilevarlo sempre e non rende la lista sicura per più thread. Se vogliamo rimuovere elementi durante una singola iterazione, usiamo `Iterator.remove()` dopo `next()`, quando l'implementazione lo supporta, oppure `removeIf` quando la regola è un predicato semplice. Se un altro thread modifica la stessa struttura, serve una politica di concorrenza definita per tutti i partecipanti; la sola presenza di un iteratore non la fornisce.

Le etichette «fail-fast» e «fail-safe» che si incontrano in alcuni testi sono scorciatoie per comportamenti diversi, non due interfacce standard da scegliere. Un iteratore di `ArrayList` tenta di segnalare alcune modifiche strutturali inattese. Le viste di `EnumMap` hanno iteratori *debolmente consistenti*: possono continuare senza eccezione e osservare o non osservare modifiche sopraggiunte. `CopyOnWriteArrayList` crea invece un percorso su una fotografia dell'array al momento dell'iterazione. Sono contratti differenti; dire che tutti gli iteratori che non falliscono lavorano «su un clone» sarebbe falso. Prima di modificare una collezione durante un percorso, leggiamo il contratto della sua implementazione e decidiamo quale risultato vogliamo osservare.

`forEach` e `forEachRemaining`: dove comincia il lavoro?

`Iterable.forEach(azione)` attraversa gli elementi forniti da un nuovo percorso e applica un `Consumer` a ciascuno. Possiamo scrivere `titoli.forEach(System.out::println)` quando l'azione è stampare ogni titolo. Un `Iterator` già ottenuto ha invece `forEachRemaining(azione)`: elabora soltanto gli elementi che restano **dal punto corrente**. Se `titoli` contiene `Java`, `Reti`, `API`, una chiamata a `iteratore.next()` consuma `Java`; poi `iteratore.forEachRemaining(System.out::println)` stampa `Reti` e `API`. Non stampa di nuovo `Java`, e non modifica i titoli. Il `for` esteso rende il controllo del percorso esplicito nel programma; `forEach` consegna l'azione all'oggetto iterabile. In entrambi i casi le eccezioni dell'azione vanno comprese: una lambda non rende innocua un'operazione che può fallire.

Una mappa offre `forEach((chiave, valore) -> ...)`, in cui l'azione è un `BiConsumer` perché riceve due dati collegati. È comodo per stampare una coppia. `entrySet()` resta preferibile quando dobbiamo passare le voci ad altri metodi o lavorare con ogni `Map.Entry` come oggetto. Se l'ordine delle stampe è parte del requisito, scegliamo una mappa che lo promette, come `LinkedHashMap` o `TreeMap`; una lambda non crea un ordine che la mappa non aveva.

Algoritmi, viste e costo delle scelte

Il Framework non contiene soltanto strutture: offre operazioni generali come ordinare una lista, cercare un valore o mescolare gli elementi. `Collections.sort(lista)` e `lista.sort(comparatore)` modificano l'ordine della lista ricevuta, se essa supporta la modifica; `lista.stream().sorted().toList()` produce invece una nuova lista risultato e lascia la sorgente com'era. Chiamare entrambe le forme «ordinamento» senza dire quale oggetto cambia renderebbe difficile prevedere il seguito del programma. Una ricerca binaria richiede che la sequenza sia già ordinata secondo lo stesso criterio usato dalla ricerca: su una lista in ordine casuale, il metodo non fornisce il risultato desiderato solo perché il nome contiene *search*.

Una `subList` è una vista su una porzione della lista, non necessariamente una copia indipendente. Se il chiamante conserva la vista e qualcun altro cambia strutturalmente la lista di origine, il comportamento pratico può sorprendere. Quando serve uno snapshot autonomo, costruire una nuova `ArrayList<>(lista.subList(inizio, fine))` esprime meglio l'intenzione. Anche `reversed()` delle interfacce sequenced è una vista invertita: iterare la vista parte dall'ultima posizione della collezione d'origine, ma non promette una copia dei dati. La differenza fra vista, copia e oggetti contenuti è una delle domande che ogni API dovrebbe risolvere nella propria documentazione.

Quanto costa una scelta? `ArrayList.get(i)` usa l'indice per raggiungere una posizione; per `LinkedList.get(i)` occorre percorrere collegamenti. Inserire alla fine di una `ArrayList` è normalmente economico, ma qualche inserimento può richiedere di riallocare e copiare l'array interno. Una `HashMap` offre in generale ricerche efficienti, ma il contratto non è la promessa che ogni input, su ogni macchina e con ogni funzione hash, costi lo stesso tempo. Il criterio pratico resta scegliere la struttura dal significato dei dati, poi controllare il costo delle operazioni dominanti e misurare se quel costo conta davvero.

Una mappa percorsa senza perdere la relazione

Se il catalogo usa `Map<String, Libro>`, attraversare soltanto `keySet()` restituisce i codici e attraversare soltanto `values()` restituisce i libri. Quando dobbiamo stampare il codice **insieme** al titolo corrispondente, `entrySet()` espone ogni coppia come `Map.Entry<String, Libro>`. Leggiamo `getKey()` e `getValue()` dalla stessa entry, senza cercare nuovamente il valore per ogni chiave. È un esempio in cui il tipo della vista comunica precisamente che cosa il ciclo sta percorrendo: non una lista generica di oggetti, ma associazioni fra chiavi e valori.

La regola delle chiavi uniche va osservata in sequenza. Dopo `put("J-01", libroA)`, la mappa contiene un'associazione; `put("J-01", libroB)` non aggiunge una seconda voce con la stessa chiave, ma sostituisce il valore e restituisce quello precedente secondo il contratto di `put`. Il numero delle voci resta uno. Se il problema richiede più libri per lo stesso autore, una semplice `Map<Autore, Libro>` non è adatta: possiamo usare `Map<Autore, List<Libro>>` o un modello diverso, dichiarando chi crea e modifica le liste interne. Non basta scegliere una mappa per risolvere automaticamente una relazione uno-a-molti.

Le viste `keySet`, `values` ed `entrySet` sono collegate alla mappa secondo il contratto dell'implementazione. Una vista ottenuta da una mappa modificabile può riflettere aggiornamenti successivi; non è uno snapshot. Al contrario, una copia esplicita conserva i dati osservati al momento della copia, almeno al livello delle associazioni e dei riferimenti copiati. Se i valori `Libro` sono mutabili, la copia della mappa non è una copia profonda di ogni libro. Questa distinzione riprende la differenza già incontrata per le liste e impedisce di promettere isolamento quando abbiamo copiato soltanto il contenitore esterno.

Laboratorio: scegliere, attraversare e conservare il risultato

Mettiamo insieme i pezzi senza nascondere i passaggi. La biblioteca ha una lista di titoli in cui `Java` compare due volte: la ripetizione è lecita perché la lista rappresenta una sequenza. Deve confrontare due gruppi di titoli per una mostra, rappresentare stati fissati da un enum e conservare una scheda per codice. Ogni esigenza sceglie un contratto diverso. Il programma completo `DemoPercorsiCollezioni.java` usa soltanto dati piccoli, così possiamo prevedere ogni risultato prima di eseguirlo.

```
import java.util.ArrayList;
import java.util.EnumMap;
import java.util.EnumSet;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;
import java.util.Set;
import java.util.StringJoiner;

public class DemoPercorsiCollezioni {
    enum Stato { DISPONIBILE, PRESTATO, RIENTRATO }

    public static void main(String[] args) {

        List<String> titoli = new ArrayList<>(List.of("Java", "Reti", "Java"));
        ListIterator<String> cursore = titoli.listIterator(titoli.size());
        StringJoiner indietro = new StringJoiner(" | ");
        while (cursore.hasPrevious()) {
            indietro.add(cursore.previous());
        }
        System.out.println("Lista a ritroso: " + indietro);

        Set<String> catalogo = new LinkedHashSet<>(List.of("Java", "Reti"));
        Set<String> mostra = new LinkedHashSet<>(List.of("Reti", "API"));
        Set<String> unione = new LinkedHashSet<>(catalogo);
        unione.addAll(mostra);
        Set<String> comuni = new LinkedHashSet<>(catalogo);
        comuni.retainAll(mostra);
        System.out.println("Unione: " + unione);
        System.out.println("Comuni: " + comuni);


        EnumSet<Stato> statiVisibili = EnumSet.of(
                Stato.DISPONIBILE, Stato.PRESTATO);
        EnumMap<Stato, Integer> conteggi = new EnumMap<>(Stato.class);
        conteggi.put(Stato.PRESTATO, 2);
        conteggi.put(Stato.DISPONIBILE, 5);
        System.out.println("Stati: " + statiVisibili);
        System.out.println("Conteggi: " + conteggi);

        Iterator<String> iteratore = titoli.iterator();
        iteratore.next();
        StringJoiner restanti = new StringJoiner(" | ");
        iteratore.forEachRemaining(restanti::add);
        System.out.println("Restanti: " + restanti);

        Map<String, String> libri = new LinkedHashMap<>();


        libri.put("J-01", "Java");
        libri.put("R-02", "Reti");
        libri.put("R-02", "Reti, seconda edizione");
        libri.forEach((codice, titolo) ->
                System.out.println(codice + " -> " + titolo));
    }
}
```

La prima riga stampata è `Lista a ritroso: Java | Reti | Java`. Il cursore parte da `titoli.size()`, cioè dopo l'ultimo elemento, e `previous()` torna indietro. Il fatto che le due parole `Java` siano uguali non le fonde: la lista conserva due posizioni. Per l'unione copiamo `catalogo` in un nuovo `LinkedHashSet` e poi chiamiamo `addAll(mostra)`: otteniamo `[Java, Reti, API]` senza cambiare i due gruppi iniziali. Per l'intersezione creiamo un'altra copia e chiamiamo `retainAll`: resta `[Reti]`. Se eliminassimo la copia, la prima operazione cambierebbe l'insieme originale e la seconda domanda partirebbe da dati diversi.

`EnumSet` produce gli stati `[DISPONIBILE, PRESTATO]`, mentre `EnumMap` stampa `{DISPONIBILE=5, PRESTATO=2}` benché nel programma `PRESTATO` sia stato inserito per primo. La mappa segue l'ordine delle costanti, non quello delle chiamate a `put`. Dopo che l'iteratore consuma `Java`, `forEachRemaining` raccoglie `Reti | Java`: il cursore non ricomincia da capo. Infine `LinkedHashMap` stampa due voci, `J-01 -> Java` e `R-02 -> Reti, seconda edizione`. La seconda `put` con la chiave `R-02` ha aggiornato una voce, non ne ha creata una terza. Ogni output conferma una proprietà diversa; il programma non è una gara fra implementazioni.

> **Per trasferire la soluzione.** Se il catalogo dovesse mostrare i libri in ordine alfabetico per titolo, cambiare soltanto `LinkedHashMap` in `TreeMap` non basterebbe: la mappa è indicizzata dal *codice*, quindi verrebbero ordinate le chiavi. Dovremmo decidere se costruire una vista ordinata delle schede per titolo o usare un altro indice. Il criterio riusabile è chiedersi *quale informazione identifica* il valore e *quale informazione stabilisce l'ordine di presentazione*.

Quando l'azione di `forEach` può fallire

L'esempio più semplice è una divisione: `divisori.forEach(n -> System.out.println(10 / n))` si interrompe con `ArithmeticException` se la lista contiene zero. Se mettiamo un `try` attorno all'intera chiamata, intercettiamo l'errore ma gli elementi dopo lo zero non vengono elaborati. Se il requisito è «prova ogni divisore e segnala quelli impossibili», il `try` va dentro l'azione oppure usiamo un `for` esplicito. La posizione del `catch` non è una questione di stile: decide se l'elaborazione continua. Questo è lo stesso ragionamento che abbiamo costruito nel capitolo 12 per le eccezioni, applicato ora a un'azione ripetuta.

Le eccezioni *checked* aggiungono un vincolo. `forEach` riceve un `Consumer<T>` il cui metodo `accept(T)` non dichiara `throws IOException`. Una lambda che chiama direttamente `leggi(nome)` se `leggi` dichiara `throws IOException` non compila. Possiamo gestire l'errore dentro la lambda; possiamo scegliere un `for` ordinario nel cui metodo chiamante propagare `IOException`; oppure possiamo adattare l'azione a un'eccezione unchecked, rendendo esplicita questa scelta nel contratto. Il prossimo programma completo confronta l'ultima scelta con la gestione elemento per elemento.

```
import java.io.IOException;
import java.io.UncheckedIOException;
import java.util.List;
import java.util.function.Consumer;

public class DemoEccezioniCollezioni {
    @FunctionalInterface
    interface AzioneIO<T> {
        void esegui(T valore) throws IOException;
    }

    static <T> Consumer<T> propaga(AzioneIO<T> azione) {
        return valore -> {
            try {
                azione.esegui(valore);
            } catch (IOException errore) {

                throw new UncheckedIOException(errore);
            }
        };
    }

    static void leggi(String nome) throws IOException {
        if (nome.equals("mancante")) {
            throw new IOException("archivio assente");
        }
        System.out.println("letto: " + nome);
    }

    public static void main(String[] args) {
        List<String> nomi = List.of("catalogo", "mancante", "storico");
        try {
            nomi.forEach(propaga(DemoEccezioniCollezioni::leggi));

        } catch (UncheckedIOException errore) {
            System.out.println("interrotto: " + errore.getCause().getMessage());
        }

        for (String nome : nomi) {
            try {
                leggi(nome);
            } catch (IOException errore) {
                System.out.println("saltato: " + nome);
            }
        }
    }
}
```

La prima visita stampa `letto: catalogo`, poi incontra `mancante` e il wrapper trasforma la `IOException` in `UncheckedIOException`, conservandola come causa. Il `catch` esterno stampa `interrotto: archivio assente`; `storico` non viene letto. La seconda visita usa un `for`: stampa di nuovo `letto: catalogo`, poi `saltato: mancante` e infine `letto: storico`. Abbiamo cambiato la politica del programma, non soltanto la sintassi del ciclo. La classe `UncheckedIOException` comunica ancora che il fallimento nasce da I/O; una generica `RuntimeException` perderebbe questa informazione specifica, pur potendo conservare la causa.

Il metodo generico `propaga` è un adattatore: riceve una `AzioneIO<T>` capace di dichiarare `IOException` e restituisce un `Consumer<T>` accettato da `forEach`. Non è una scorciatoia da applicare automaticamente a ogni eccezione checked. Se chi chiama deve sapere *quale* file è fallito e continuare con gli altri, il secondo percorso o un risultato strutturato è più chiaro. Se l'operazione deve fermarsi al primo errore, l'adattatore rende l'errore visibile all'esterno e mantiene la causa. Il mattone da riusare è la domanda «che cosa deve accadere agli elementi successivi quando uno fallisce?»; la risposta decide dove intercettare l'eccezione e quale API usare.

Per verificare

Compila `DemoCollezioni.java`, `DemoPercorsiCollezioni.java` e `DemoEccezioniCollezioni.java` con `javac --release 25 -Xlint:all -d build` seguito dai tre nomi dei file; poi esegui ciascuna classe. Prima dell'esecuzione, prevedi perché la lista conserva due `Java`, l'insieme uno solo e la mappa aggiorna `R-02` senza aumentare dimensione. Togli `thenComparing` dal comparatore del `TreeSet`: quanti titoli rimangono e quale informazione è stata persa? Prova a chiamare `add` sulla lista prodotta da `List.copyOf` e spiega l'errore. Infine cambia il requisito del laboratorio: dopo un file mancante bisogna continuare con i successivi. Indica quale dei due percorsi di `DemoEccezioniCollezioni` soddisfa la richiesta e perché. Se sai scegliere la struttura e la gestione dell'errore per questa variante, hai portato il mattone fuori dall'esempio iniziale.

18. Stream API: pipeline, riduzioni e Gatherer

Dalla collezione al calcolo

Riprendiamo il modello di lista del capitolo 17 e consideriamo i numeri `1, 2, 3, 4, 5, 6, 3`. Ora vogliamo una nuova lista senza i valori uguali a 3. Possiamo scrivere un ciclo, controllare ogni numero e aggiungere quelli ammessi a una lista nuova. La Stream API offre un'altra forma per descrivere quel calcolo: `dati.stream().filter(numero -> numero != 3).toList()`. `filter` esprime la condizione, `toList` produce il risultato. Il ciclo resta una soluzione valida; lo stream è utile quando la sequenza di trasformazioni rende più leggibile l'intenzione.

Uno stream non richiede al lettore di partire da concetti come le monadi e non promette, da solo, lavoro asincrono o maggiore velocità. Uno **stream Java** è una vista di elaborazione su una sorgente di elementi, con operazioni aggregate che possono essere composte. Non è, di per sé, una struttura che contiene gli elementi come una lista; non è automaticamente asincrono; una pipeline sequenziale lavora normalmente nel thread chiamante. Il parallelismo è una scelta distinta, con costi e vincoli propri.

> **Definizione – Pipeline.** Una pipeline comprende una sorgente, zero o più operazioni intermedie e un'operazione terminale. La sorgente può essere una collezione, un array, un generatore o una risorsa come un file. Le operazioni intermedie descrivono trasformazioni; quella terminale richiede un risultato o un effetto e consuma lo stream. Dopo il consumo, lo stesso stream non è riutilizzabile.

Nel programma completo, `dati.stream()` crea uno stream dalla lista, `filter` descrive gli elementi da mantenere e `toList()` produce `[1, 2, 4, 5, 6]`. La seconda stampa mostra ancora `[1, 2, 3, 4, 5, 6, 3]`: la pipeline non ha modificato la lista sorgente. `Stream.toList()` restituisce una lista non modificabile. Se vogliamo una nuova `ArrayList` modificabile, la costruiremo esplicitamente dal risultato o useremo un collector appropriato.

La figura 18.1 fa vedere l'ordine logico dei passaggi. Non va letta come promessa che ogni passaggio venga completato su tutti gli elementi prima di iniziare il successivo: l'implementazione può attraversare la pipeline elemento per elemento e, quando è lecito, evitare lavoro non necessario.

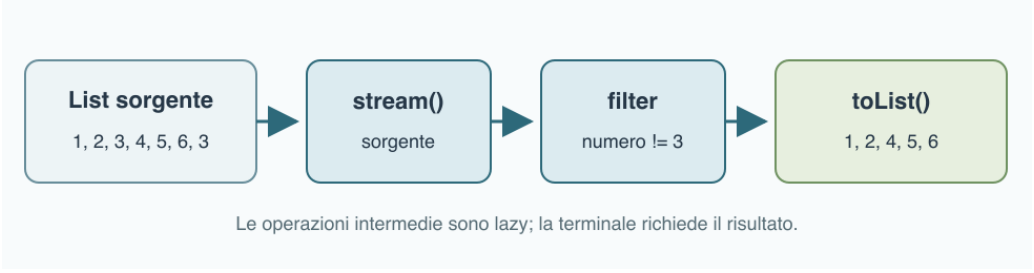


Figura 18.1 – La sorgente alimenta `filter`, la terminale `toList` materializza una nuova lista. Lo stream descrive l'elaborazione; la lista finale è un oggetto distinto dalla sorgente.

Dal ciclo alla pipeline, senza perdere il controllo

Riprendiamo il problema che apre il capitolo: eliminare dalla nuova lista tutti i valori uguali a 3. Nel ciclo `for` che segue vediamo tre operazioni: attraversare la sorgente, decidere se un elemento passa e aggiungerlo al risultato. La pipeline `filter(...).toList()` conserva le stesse tre responsabilità, ma affida l'attraversamento all'API. L'iterazione continua a esistere: il nostro codice descrive la condizione e la forma del risultato, mentre la libreria gestisce il percorso. Confrontiamo quindi i due risultati sugli stessi dati prima di scegliere la forma più leggibile. Lo stream è utile quando il calcolo si legge come una sequenza di trasformazioni; se ogni elemento richiede molti rami di errore con effetti diversi, un ciclo esplicito può rendere più facile seguire e verificare il lavoro.

```
List<Integer> origine = List.of(1, 2, 3, 4, 5, 6, 3);
List<Integer> copia = new ArrayList<>();
for (int valore : origine) {
    if (valore != 3) copia.add(valore);
}
List<Integer> filtrati = origine.stream()
        .filter(valore -> valore != 3)
        .toList();
```

I due risultati hanno gli stessi elementi `[1, 2, 4, 5, 6]`, ma non lo stesso contratto di modificabilità: `copia` è una `ArrayList` che possiamo modificare, mentre la lista restituita da `Stream.toList()` è non modificabile. La sorgente `origine` resta `[1, 2, 3, 4, 5, 6, 3]` in entrambi i casi. Questa verifica impedisce di scambiare una somiglianza di output per identità completa di comportamento. Se il risultato deve essere una lista mutabile, lo dichiariamo e la costruiamo; se non deve cambiare, la lista di `toList` rende utile quel vincolo.

Una terza versione con `origine.forEach(x -> { if (...) copia.add(x); })` usa una lambda ma conserva l'accumulatore esterno. Su un ciclo sequenziale piccolo può funzionare; non offre il vantaggio di un risultato espresso dalla pipeline e diventa pericolosa se qualcuno sostituisce la sorgente con un parallel stream. La presenza di una lambda non basta a rendere il codice "funzionale". È la relazione fra input, trasformazioni ed effetti a stabilire se il modello è adatto.

Sorgente, passaggi intermedi, terminale

Una pipeline inizia da una sorgente. `origine.stream()` produce uno stream associato alla lista. `filter` è un'operazione intermedia: definisce quali elementi potranno proseguire e restituisce un altro stream. `toList` è terminale: richiede l'elaborazione e restituisce una lista. Anche `count`, `sum`, `findFirst`, `anyMatch`, `collect` e `forEach` sono terminali, ma non producono tutti lo stesso tipo di risultato. Alcune restituiscono un valore, altre un `Optional`, altre un effetto. Prima di scegliere la terminale, formuliamo la domanda: vogliamo tutti gli elementi, il primo, un conteggio, una somma o una stampa?

La distinzione non è una convenzione grafica. Una catena che termina con `filter` ha soltanto descritto un lavoro; se non viene consumata, non produce la lista filtrata. Una catena che chiama `count` ha consumato lo stream: lo stesso oggetto non può essere riattraversato con `toList`. Possiamo invece chiedere alla lista sorgente un nuovo `stream()` per una seconda domanda. Questa caratteristica è

simile a un `Iterator` che, una volta avanzato fino alla fine, non torna automaticamente all'inizio. Non dobbiamo confondere la consumazione della pipeline con la chiusura automatica di una risorsa esterna: uno stream di file richiede ancora la gestione del file.

> **Box – La pigrizia non è una promessa di chiamate.** L'API garantisce il risultato della pipeline secondo il contratto delle operazioni, non che una lambda intermedia venga invocata esattamente una volta per ogni elemento in ogni caso. Un'implementazione può evitare lavoro inutile. Per questo `map(x -> { registra(x); return x; }).count()` è un posto sbagliato per un'azione necessaria come `registra`. Se l'azione deve avvenire, rendiamola esplicita e controlliamo il suo ciclo di vita; se vogliamo soltanto contare, usiamo `count` senza effetto nascosto.

Costruire stream da sorgenti diverse

La `List` è la sorgente più familiare, ma uno stream può arrivare da un array, da valori dichiarati nel codice, da un generatore o da un file. `Stream.of("rosso", "blu")` è pratico per pochi valori noti; `Stream.<String>empty()` rappresenta nessun elemento. `Stream.ofNullable(valore)` produce zero elementi quando il valore è `null` e uno quando è presente. Non va confuso con `Stream.of(valore)`: nel caso di un singolo riferimento `null`, l'overload e il significato desiderato possono essere ambigui per il lettore; scegliere `ofNullable` rende esplicita l'assenza.

`Stream.builder()` permette di aggiungere valori uno alla volta prima di costruire lo stream. È utile quando gli elementi vengono selezionati durante una breve preparazione; dopo `build`, il builder non è un contenitore da continuare a modificare. Per dati già dentro una collezione, creare un builder e copiare ogni valore aggiunge lavoro senza beneficio. `Arrays.stream(array)` usa un array esistente; con `int[]` produce `IntStream`, con `String[]` produce `Stream<String>`. Lo stesso nome del metodo nasconde overload adatti ai diversi tipi, perciò leggiamo sempre il tipo di ritorno che il compilatore ha scelto.

Un testo separato da virgole può essere letto con `Pattern.compile(",").splitAsStream(testo)`. Qui il separatore è una **espressione regolare**. La virgola semplice non richiede escape, ma un punto significherebbe “qualunque carattere” se non fosse protetto. Dopo la divisione possiamo applicare `map(String::strip)` e `filter(s -> !s.isEmpty())` per pulire gli elementi. Una pipeline non convalida automaticamente il contenuto: se una voce è malformata, serve una regola di dominio e una decisione sull'errore.

`Stream.generate(supplier)` chiede nuovi elementi al supplier ogni volta che servono. Senza un limite può produrre una sequenza senza fine. `Stream.iterate(seme, passo)` costruisce il primo valore dal seme e i successivi applicando il passo al precedente. La forma con predicato `Stream.iterate(seme, continua, passo)` ferma la sequenza quando la condizione non è più vera. Nel programma completo `DemoCatalogoStream.java`, `Stream.iterate(2, n -> n <= 8, n -> n + 2).toList()` produce `[2, 4, 6, 8]`: il valore 2 è il seme, `n <= 8` decide se emetterlo, `n + 2` prepara il successivo. L'ordine dei tre argomenti non è arbitrario; segue la domanda “da dove parto, quando continuo, come avanzo?”.

Una sorgente infinita richiede una terminale che possa fermarsi oppure un'operazione intermedia che la renda finita. `Stream.generate(() -> "x").limit(3).toList()` termina con tre valori; `Stream.generate(() -> "x").toList()` non ha un termine naturale e può esaurire le risorse. `findFirst()` può terminare anche senza `limit`, purché la ricerca sia in grado di trovare un elemento e le operazioni precedenti lo lascino passare. La presenza di una operazione di arresto anticipato non garantisce da sola che ogni pipeline finisca: se il predicato non è mai soddisfatto, la ricerca può continuare senza fine.

La valutazione è pigra

Molte operazioni intermedie sono **lazy**, cioè pigre: costruire `dati.stream().filter(...)` non attraversa immediatamente tutti i dati. Il lavoro richiesto avviene quando una terminale, come `toList`, `count`, `findFirst` o `forEach`, consuma la pipeline. Questo permette a operazioni di **short circuit**, o arresto anticipato, di non esaminare necessariamente tutta la sorgente. Per esempio, su uno stream ordinato, `filter(...).findFirst()` può fermarsi appena trova il primo elemento adatto.

La figura 18.2 separa il momento in cui prepariamo la pipeline da quello in cui chiediamo il risultato. Il caso di Anna è volutamente minimo: consente di verificare perché Andrea e Luca non devono necessariamente essere esaminati.

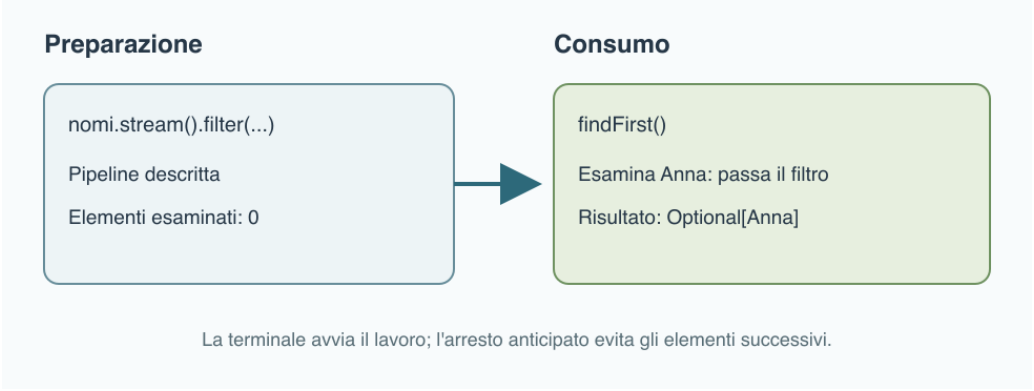


Figura 18.2 – Costruire `filter` non visita i nomi; `findFirst` avvia la ricerca e può fermarla sul primo risultato.

La pigrizia non autorizza a nascondere effetti in `filter` o `map` aspettandosi che vengano sempre eseguiti una volta per elemento. Alcune pipeline possono evitare o riorganizzare chiamate quando ciò non cambia il risultato richiesto dalla terminale. Un predicato dovrebbe descrivere una condizione, non registrare una fattura o incrementare un contatore necessario alla correttezza del programma. La documentazione Java 25 del package `java.util.stream` chiama **non-interfering** i comportamenti che non modificano la sorgente durante l'elaborazione e **stateless** quelli che non dipendono da uno stato variabile fra gli elementi.

Un oggetto `Stream` si usa per una sola pipeline consumata una volta. Se dobbiamo effettuare due calcoli sulla stessa collezione, possiamo creare due stream dalla collezione: `dati.stream()` a ogni calcolo. Conservare il riferimento allo stream e chiamare due terminali su di esso è un errore di ciclo di vita. L'eccezione che può comparire è una conseguenza, non il principio da memorizzare.

Osservare la pigrizia in esecuzione

Verifichiamo la distinzione appena introdotta senza usare una pausa artificiale. Prepariamo una lista di tre nomi e un filtro che stampa una riga diagnostica quando viene chiamato. Subito dopo aver costruito la pipeline, stampiamo `preparata`; poi invochiamo `findFirst`. L'output inizia da `preparata`, perché la semplice costruzione della pipeline non attraversa la lista. Solo la terminale mette in moto il lavoro. Se il primo nome soddisfa il predicato, la ricerca può fermarsi lì. La stampa nel predicato è uno strumento temporaneo per osservare l'esecuzione, non una parte della logica applicativa.

```
List<String> nomi = List.of("Anna", "Andrea", "Luca");
Stream<String> ricerca = nomi.stream().filter(nome -> {
    System.out.println("Esamino " + nome);
    return nome.startsWith("A");
});
System.out.println("Pipeline preparata");
String primo = ricerca.findFirst().orElse("nessuno");
System.out.println("Trovato " + primo);
```

Con questo ordine dei dati, ci aspettiamo prima `Pipeline preparata`, poi `Esamino Anna`, infine `Trovato Anna`. Il programma ci permette di associare il *problema* “trovare il primo nome adatto” alla *soluzione* `filter` più `findFirst`, alla *verifica* dell'arresto anticipato e al *trasferimento* verso una lista più grande. Se sostituiamo `findFirst` con `toList`, la terminale deve produrre tutti i nomi che passano e quindi attraversa anche gli elementi successivi. Se inseriamo `sorted()` prima di `findFirst`, l'operazione di ordinamento può dover conoscere tutti gli elementi prima di restituire il primo ordinato. La presenza di una terminale breve non cancella il costo delle operazioni precedenti.

L'ordine delle operazioni va ragionato con attenzione. Filtrare prima di una trasformazione costosa può risparmiare lavoro, **se** il filtro ha già le informazioni necessarie e le due operazioni possono essere spostate senza cambiare il significato. Se il filtro deve valutare il risultato della trasformazione, allora la trasformazione deve venire prima. Anche `distinct` e `sorted` possono cambiare quali elementi una `limit` vede: spostare `limit(10)` prima di `sorted` significa ordinare solo i primi dieci elementi incontrati, non scegliere i dieci minori dell'intera sorgente. Le regole di prestazione non sono permessi per modificare silenziosamente la risposta al problema.

Uno stream fresco per ogni domanda

Una lista di città può servire a rispondere prima “quante sono?” e poi “quale viene per prima in ordine alfabetico?”. Sono due terminali e richiedono due stream. La forma più semplice è chiamare `citta.stream()` due volte. Se la costruzione della sorgente è incapsulata in un metodo, possiamo usare `Supplier<Stream<Citta>>` e chiamare `get()` per avere una pipeline nuova a ogni domanda. Il supplier non rigenera necessariamente i **dati**: può restituire un nuovo stream sopra la stessa lista. Se invece apre un file a ogni chiamata, porta anche una responsabilità di chiusura e un costo I/O che non vanno nascosti.

```
Supplier<Stream<String>> sorgente = () -> nomi.stream();
long quanti = sorgente.get().count();
String primo = sorgente.get().sorted().findFirst().orElse("nessuno");
```

Questa soluzione è utile quando più funzioni ricevono la capacità di creare uno stream nuovo; nel codice ordinario, due chiamate dirette a `nomi.stream()` restano più immediate. Una pipeline non è una collezione da conservare per riuso. Il messaggio di errore `stream has already been operated upon or closed` segnala che abbiamo cercato di attraversare di nuovo un oggetto consumato; ricreare lo stream dalla sorgente è la risposta, purché la sorgente sia ancora disponibile e valida.

Trasformare, appiattire e interrompere

`map` trasforma ogni elemento in un altro valore. Da titoli possiamo ottenere lunghezze con `map(String::length)`. Se la trasformazione di un elemento produce a sua volta più elementi, `flatMap` permette di appiattirli in un solo stream. Nel programma, le righe `"Java,Reti"` e `"API"` diventano `[Java, Reti, API]` attraverso `flatMap(riga -> Stream.of(riga.split(",")))`. La trasformazione è visibile: prima dividiamo ogni riga, poi uniamo i piccoli stream risultanti.

`mapMulti` è un'altra forma per produrre zero o più risultati da un elemento mediante un consumatore fornito alla funzione. Può essere utile quando l'apertura di molti piccoli stream in `flatMap` renderebbe la pipeline meno adatta al problema, ma la scelta va fatta dopo aver verificato che il corpo resti leggibile. `Stream.ofNullable(x)` produce zero elementi se `x` è `null`, uno altrimenti; è utile per evitare un controllo separato in piccoli adattamenti, senza trasformare `null` in una scelta di modello sempre desiderabile.

Su una sorgente con ordine d'incontro, `takeWhile` conserva il prefisso finché il predicato è vero e si ferma al primo elemento che lo rende falso. `dropWhile` scarta quel prefisso e lascia il resto. Non sono sinonimi di `filter`, che valuta la condizione anche su elementi successivi. `Stream.iterate(seme, predicato, passo)` può generare una sequenza finita quando il predicato stabilisce quando fermarsi; se manca un limite, una terminale che richiede tutti gli elementi può non terminare. `limit` e `skip` sono altri strumenti di taglio, ma il loro costo dipende dall'ordine e dalla sorgente.

Gli stream primitivi come `IntStream`, `LongStream` e `DoubleStream` evitano il boxing di ogni elemento e offrono operazioni numeriche come `sum` e `average`. Non sempre conviene trasformare tutto in primitivi: se gli elementi hanno un significato di dominio, perdere il tipo dell'oggetto può rendere il codice meno chiaro. La scelta corretta unisce contratto, leggibilità e misurazione quando le prestazioni contano davvero.

Operazioni intermedie: seguire un elemento alla volta

`filter` mantiene o scarta un elemento secondo un `Predicate`. Su `["Roma", "Rovigo", "Bari"]`, il filtro `citta -> citta.startsWith("R")` produce le prime due città. `map` applica una `Function` a ciascun elemento che arriva: `map(String::toUpperCase)` cambia il testo, mentre `map(String::length)` cambia anche il tipo dello stream da `Stream<String>` a `Stream<Integer>`. Se vogliamo sommare le lunghezze, `mapToInt(String::length).sum()` esprime direttamente un calcolo numerico senza una lista intermedia di wrapper.

`distinct` elimina duplicati secondo la nozione di uguaglianza degli elementi. Per stringhe la distinzione è sensibile alle maiuscole: `"Roma"` e `"ROMA"` non sono uguali. Se vogliamo città distinte indipendentemente dalla grafia, dobbiamo decidere come normalizzarle, e `map(String::toUpperCase).distinct()` è una possibile soluzione. Ma normalizzare prima di conservare il nome originale significa anche perdere la grafia iniziale. Questo è un problema di modello, non un parametro nascosto di `distinct`.

`sorted` ordina gli elementi della pipeline. Per stringhe usa l'ordine naturale, che può non coincidere con un ordinamento linguistico locale; per oggetti di dominio serve un `Comparator` oppure un ordine naturale definito dalla classe. `sorted(Comparator.comparing(Libro::titolo))` dice quale proprietà guida il confronto. Poiché per ordinare bisogna conoscere gli elementi da confrontare, `sorted` può richiedere di trattenere molti dati. Su una sorgente potenzialmente infinita, `sorted().findFirst()` non è un trucco innocuo per trovare il minimo: una riduzione `min(comparatore)` esprime meglio la richiesta e può lavorare senza ordinare tutti gli elementi.

`skip(n)` lascia cadere i primi `n` elementi, `limit(n)` non ne lascia passare più di `n`. Su una sorgente ordinata sono utili per una finestra di posizioni, ma non rappresentano automaticamente una paginazione stabile di dati che cambiano. Se nuovi elementi vengono inseriti all'inizio fra due richieste, la seconda pagina può ripetere o perdere valori. Qui il trasferimento dal piccolo esempio alla applicazione reale richiede un ordine stabile e un contratto sullo stato della sorgente. `takeWhile` e `dropWhile` lavorano invece su un **prefisso** definito da un predicato: dopo il primo elemento che interrompe il prefisso, non riprendono come farebbe un filtro generale.

`flatMap` entra in gioco quando un elemento della sorgente produce a sua volta una sequenza. Una lista di righe `"Java,Reti"`, `"API"` diventa una lista di parole: `map(riga -> riga.split(","))` darebbe uno stream di array, non ancora un unico flusso di parole; `flatMap(riga -> Arrays.stream(riga.split(",")))` unisce gli elementi dei piccoli stream in un flusso solo. La parola “appiattire” indica proprio il passaggio da un livello annidato a uno lineare. `mapMulti` permette di emettere zero o più elementi senza costruire esplicitamente uno stream per ogni ingresso; è utile soprattutto quando questa scelta semplifica un caso concreto, non per sostituire a priori ogni `flatMap`.

Le operazioni intermedie **stateless**, come i normali `filter` e `map` con funzioni senza stato, non devono ricordare elementi precedenti. Operazioni **stateful** come `distinct` e `sorted` hanno invece bisogno di informazione sull'insieme già visto o sull'insieme da ordinare. Questo spiega una differenza di costo che una lista di nomi di metodi non mostra. Una pipeline può essere pigra e tuttavia dover

accumulare molti elementi quando arriva alla terminale. La pigrizia riguarda *quando* si lavora; la memoria necessaria dipende anche da *quale* operazione abbiamo richiesto.

Le terminali rispondono a domande diverse

Se chiediamo “quante città iniziano con R?”, `filter(...).count()` produce un `long`. Se chiediamo “ne esiste almeno una?”, `anyMatch(...)` produce un booleano e può fermarsi al primo successo. `allMatch` chiede se tutte soddisfano la condizione; `noneMatch` se nessuna la soddisfa. Su uno stream vuoto `anyMatch` è `false`, mentre `allMatch` e `noneMatch` sono `true`: nel caso di `allMatch` non esiste un controesempio, e nel caso di `noneMatch` non esiste un esempio che violi la condizione. Questa regola spesso sorprende, perciò va verificata con un piccolo programma prima di usarla in una validazione importante.

`findFirst` restituisce il primo elemento secondo l'ordine d'incontro, quando l'ordine esiste; `findAny` permette di scegliere un elemento qualsiasi. Entrambi restituiscono `Optional<T>` perché la sorgente o il filtro possono non produrre nulla. Il metodo `orElse("nessuna città")` rende esplicita la risposta in quel caso. `min` e `max` restituiscono anch'essi un `Optional` di oggetto: non c'è un minimo di una sequenza vuota. `count`, al contrario, ha un risultato definito, zero. Vedere il tipo di ritorno aiuta a riconoscere se l'assenza deve essere trattata.

`forEach` esegue un'azione, quindi serve quando l'effetto è proprio il risultato richiesto, per esempio inviare ogni riga a un consumer. Non restituisce una collezione. Per ottenere una lista, usiamo `toList` o `collect`; per ottenere un array tipizzato, `toArray(String[]::new)` fornisce una fabbrica della dimensione giusta. `Collectors.joining(", ")` concatena stringhe con un separatore; scegliere un separatore non aggiunge automaticamente regole di quoting o escape per un formato CSV. Se il risultato deve essere un vero file CSV, occorre gestire i campi che contengono virgole, virgolette e a capo.

Riduzione e raccolta, con un dominio concreto

Il programma `DemoCatalogoStream.java` usa quattro libri, ciascuno con titolo, reparto, numero di pagine e disponibilità. La prima domanda è “quali titoli disponibili mostro in ordine?”. Il programma filtra i disponibili, estrae i titoli, rimuove gli spazi ai bordi, ordina e produce una lista. Il risultato è `[Algebra, Algoritmi, Java]`. La sequenza dei metodi segue la domanda: prima selezioniamo libri, poi trasformiamo i libri in testi, poi puliamo e ordiniamo i testi. Se facessimo `map(Libro::titolo)` prima di filtrare sulla disponibilità, avremmo perso proprio la proprietà che serve al filtro.

La seconda domanda è “quanti libri ci sono in ciascun reparto?”. `Collectors.groupingBy(Libro::reparto, Collectors.counting())` produce una mappa: la chiave è il reparto, il valore è il conteggio. `perReparto.get("Informatica")` vale 3. Un `groupingBy` semplice produrrebbe invece una mappa di liste di libri. Il collector a valle `counting` cambia il tipo del valore e evita di costruire liste quando vogliamo soltanto un numero. Il programma stampa solo la chiave che ci interessa: non facciamo dipendere una verifica dall'ordine di iterazione di una mappa per cui non abbiamo chiesto una garanzia d'ordine.

La terza domanda riguarda le pagine complessive. `mapToInt(Libro::pagine)` porta da oggetti `Libro` a valori primitivi `int`; `summaryStatistics()` calcola in una sola terminale conteggio, somma, minimo, massimo e media. Il programma usa conteggio e somma e stampa `4 libri, 1150 pagine`. È un caso pratico del problema "voglio più riepiloghi ma non posso riusare lo stesso stream": le statistiche sono un oggetto risultato che contiene più misure ottenute dal consumo della pipeline. Se un capitolo ha zero libri, alcune statistiche come la media non hanno il significato che avrebbero su una popolazione non vuota; il chiamante deve decidere come rappresentare quell'assenza.

La quarta domanda è "qual è il libro disponibile con meno pagine?". Dopo il filtro, `min(Comparator.comparingInt(Libro::pagine))` confronta i libri secondo il campo pagine e restituisce un `Optional<Libro>`. `map(Libro::titolo)` trasforma il libro trovato nel titolo; `orElse("nessuno")` copre il caso di catalogo senza libri disponibili. Il risultato del programma è `Algebra`. La forma completa applica l'idea di `Optional` anticipata nel capitolo 16: un minimo potrebbe mancare e il chiamante deve esprimere che cosa fare. Il capitolo 19 svilupperà questo tipo e le scelte per gestire l'assenza.

`reduce` serve quando possiamo combinare due valori dello stesso problema in un risultato del medesimo tipo. Con `reduce(0, Integer::sum)` lo zero è l'identità e la somma combina risultati parziali. Se usiamo `reduce` per concatenare molte stringhe con `a + b`, creiamo potenzialmente molti oggetti intermedi; `Collectors.joining` comunica meglio la finalità. `collect` è invece pensato per accumulare elementi in una struttura risultato con un protocollo di supplier, accumulator e combiner. Le due operazioni non sono sinonimi universali. Un collector ben scelto evita di condividere una `ArrayList` esterna fra le lambda della pipeline, soprattutto se la pipeline diventa parallela.

Ridurre e raccogliere risultati

Una **riduzione** combina più elementi in un risultato. `count()` restituisce quanti elementi attraversano la pipeline; `reduce` può combinare numeri o altri valori con un'operazione compatibile; `collect` usa un **collector** per costruire un risultato, per esempio una mappa di gruppi. Nel programma, `Collectors.groupingBy(String::length)` raggruppa parole per lunghezza. Leggendo la chiave 4 otteniamo `[Java, Reti]`. Non stampiamo l'intera mappa affidandoci al suo ordine, perché il contratto generale di `groupingBy` non promette un ordine d'iterazione specifico della mappa restituita.

`Collectors.teeing` applica due collector agli stessi elementi e combina i risultati. L'esempio conta sette numeri e ne somma i valori, stampando `7 elementi, somma 24`. Il nome non implica due thread o due passaggi obbligati sulla sorgente: indica una combinazione di due raccolte logiche. Per una somma semplice useremmo `mapToInt(...).sum()`, più breve. Il collector doppio è utile quando servono davvero due aggregazioni correlate nello stesso risultato.

> **Nota bene – Identità e associatività.** Per usare `reduce` correttamente, il valore iniziale deve comportarsi da identità dell'operazione e la combinazione deve essere associativa se vogliamo che suddivisioni e riaggregazioni producano lo stesso risultato. Per la somma degli interi, 0 è l'identità: `0 + x` vale `x`. Per una sottrazione, cambiare il raggruppamento può cambiare il risultato. Questo limite diventa decisivo quando si considera il parallelismo.

`findFirst` e `findAny` restituiscono `Optional`, anticipato nel capitolo 16 e approfondito nel capitolo 19; qui basta sapere che il risultato può essere assente e va gestito. `anyMatch`, `allMatch` e `noneMatch` rispondono a domande booleane e possono arrestarsi prima della fine. `sorted` ordina gli elementi della pipeline secondo un comparatore o l'ordine naturale; se abbiamo bisogno di conoscere l'ordine della collezione originale, non possiamo dedurlo soltanto dall'ordine prodotto da `sorted`.

Una riduzione letta elemento per elemento

Per sommare 1, 2 e 3 possiamo scrivere `reduce(0, Integer::sum)`. Il valore iniziale 0 è l'identità della somma: aggiungerlo non altera il primo elemento. La riduzione combina 0 con 1, ottiene 1; combina 1 con 2, ottiene 3; combina 3 con 3, ottiene 6. Non è una scorciatoia magica che esegue sempre esattamente tre chiamate nell'ordine descritto: questa lettura mostra il caso sequenziale, mentre una pipeline parallela può suddividere il lavoro e ricombinare risultati parziali. La somma è associativa nell'aritmetica matematica, ma con `int` l'overflow resta possibile; per quantità che possono superare il limite occorre un tipo o un controllo appropriato.

Se vogliamo raccogliere i valori in una lista, la domanda cambia: non cerchiamo un numero unico ma una struttura risultato. `toList()` esprime direttamente la richiesta di una lista non modificabile; `collect` permette strategie di raccolta più articolate, come un raggruppamento. Accumulare elementi in una `ArrayList` esterna con effetti in `forEach` può sembrare simile su tre numeri, ma introduce uno stato condiviso difficile da controllare se la pipeline cambia. La distinzione fra riduzione a un valore, raccolta in una struttura ed effetto esterno prepara una scelta corretta anche quando la pipeline diventa lunga.

Raccolte più ricche: scegliere la struttura finale

`Collectors.toCollection(ArrayList::new)` costruisce esplicitamente una lista mutabile del tipo richiesto; `Collectors.toCollection(TreeSet::new)` produce un insieme ordinato che elimina duplicati secondo il suo comparatore. Non è una conversione neutra: due testi che il comparatore considera equivalenti possono diventare un solo elemento, anche se erano due elementi nella sorgente. Se occorre preservare duplicati, un `TreeSet` non rappresenta il risultato giusto. `Collectors.toMap` richiede una scelta quando due elementi producono la stessa chiave: senza una funzione di fusione, la collisione è un errore, non una decisione nascosta della libreria.

Supponiamo di voler associare il codice di ciascun libro al suo titolo. Se il catalogo può contenere due edizioni con lo stesso codice, dobbiamo stabilire se la chiave è davvero unica, se va conservata la prima, l'ultima o una lista di entrambe.

`groupingBy(Libro::codice)` esprime una mappa da codice a lista; `toMap(Libro::codice, Libro::titolo, (primo, secondo) -> primo)` dichiara la politica “tieni il primo”. Quest'ultima è una decisione di dominio che non andrebbe nascosta in una lambda priva di spiegazione. La struttura finale racconta come intendiamo trattare le collisioni.

`partitioningBy(predicato)` crea due gruppi indicizzati da `true` e `false`, adatto a una divisione binaria come disponibili e non disponibili. `groupingBy` può invece produrre molti gruppi, uno per reparto. `mapping` come collector a valle permette di raccogliere soltanto i titoli dentro ciascun gruppo; `summingInt` e `averagingDouble` calcolano valori per gruppo senza conservare ogni elemento. Non serve memorizzare un elenco di combinazioni: formuliamo la domanda, scegliamo la chiave o il predicato e poi decidiamo che cosa vogliamo dentro ogni gruppo.

> **Box – Sommare importi monetari.** Gli esempi numerici del capitolo usano pagine e conteggi, per cui `int` o `long` sono naturali entro limiti dichiarati. Per denaro, `double` può introdurre approssimazioni binarie indesiderate. Se il dominio richiede importi decimali esatti e regole di arrotondamento, rappresentiamo gli importi con un tipo adatto come `BigDecimal` e riduciamoli con una somma coerente con il contratto. Lo stream non decide il modello numerico al posto nostro.

Dal valore numerico allo stream primitivo

`IntStream`, `LongStream` e `DoubleStream` lavorano con valori primitivi; la differenza rispetto a `Stream<Integer>` si capisce meglio osservando un calcolo. Il programma `DemoStreamNumerici.java` parte dagli interi da 1 a 4. `IntStream.rangeClosed(1, 4)` include entrambi gli estremi e produce quattro valori; `range(1, 4)` si fermerebbe prima di 4. La prima terminale `sum()` stampa `10`. Per fare un secondo calcolo si crea **un nuovo stream** con una seconda chiamata a `rangeClosed`: non si riusa quello già consumato. `average()` calcola la media, cioè la somma 10 divisa per i quattro elementi, e produce un `OptionalDouble`, perché una sequenza vuota non avrebbe una media. In questo caso sappiamo che non è vuota e `orElseThrow()` restituisce `2.5`.

Nella figura 18.3 le quattro famiglie condividono il contratto `BaseStream`, ma non lo stesso tipo di elemento. Le tre specializzazioni primitive non sono sottoclassi di `Stream<T>`: la conversione da oggetti a primitivi e viceversa avviene con metodi come `mapToInt` e `boxed`.

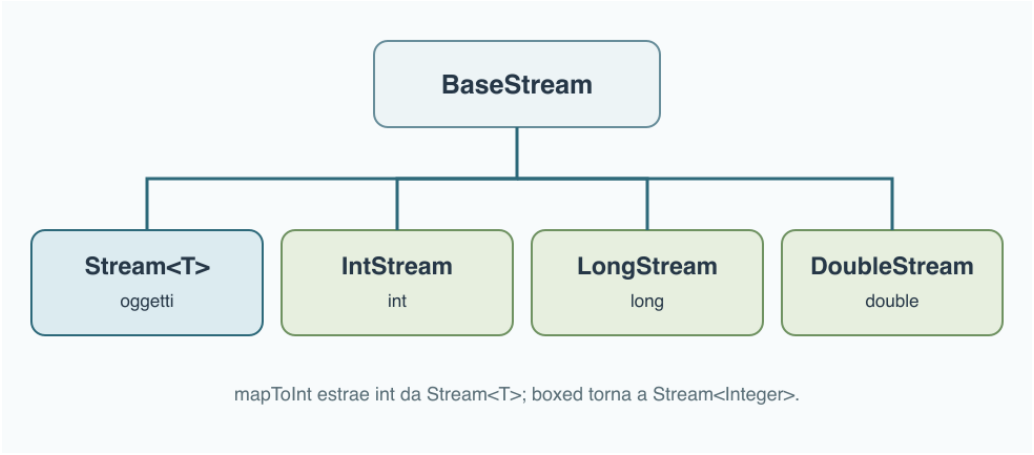


Figura 18.3 – `Stream<T>` elabora oggetti; `IntStream`, `LongStream` e `DoubleStream` elaborano i tre primitivi supportati dalle API specializzate.

La terza pipeline applica `map(numero -> numero * numero)` e ottiene i quadrati 1, 4, 9 e 16. `boxed()` trasforma i valori primitivi dello `IntStream` in `Integer`, così `toList()` produce una `List<Integer>` stampata come `[1, 4, 9, 16]`. Nella quarta pipeline il percorso va nella direzione opposta: `Stream.of("2", "3")` contiene stringhe, `mapToInt(Integer::parseInt)` le interpreta come interi e `sum()` restituisce 5. Se una stringa non rappresenta un intero valido, `parseInt` lancia un'eccezione; lo stream non la converte automaticamente in zero né la scarta. È il chiamante a decidere se validare prima o gestire l'errore.

Gli stream primitivi evitano il boxing per ogni elemento del percorso numerico e offrono terminali come `sum`, `average`, `min` e `max`. Non sono un obbligo di stile: se lavoriamo con oggetti del dominio, trasformarli subito in numeri può far perdere il significato delle operazioni. Anche il possibile vantaggio di prestazioni dipende dal lavoro reale. Il primo criterio resta rendere chiaro ciò che entra, ciò che esce e la ragione per cui il tipo cambia.

Scegliere lo stream primitivo quando cambia il problema

L'esempio precedente ha mostrato `IntStream` su numeri già disponibili. Nel catalogo, invece, partiamo da oggetti `Libro` e li conserviamo come tali finché la domanda riguarda titolo e disponibilità. `mapToInt(Libro::pagine)` rende numerico il percorso quando chiediamo una statistica sulle pagine. Per costruire direttamente uno stream primitivo possiamo usare `IntStream.of(1, 2, 3)`, che parte da valori espliciti, oppure `IntStream.range(1, 4)`, che produce 1, 2, 3; `rangeClosed(1, 4)` include anche 4. `Arrays.stream(int[])` produce ancora `IntStream`. Metodi analoghi esistono per `long` e `double`; la libreria non offre uno stream distinto per ogni primitivo Java.

`sum()` restituisce la somma; `average()` produce `OptionalDouble`; `min()` e `max()` producono `OptionalInt` per `IntStream`. Il carattere opzionale ricorda che una sequenza vuota non ha media, minimo o massimo. Chiamare `getAsInt()` senza verificare la presenza su uno stream vuoto causa un errore; `orElse`, `orElseThrow` o `isPresent` rendono visibile la scelta. `count()` restituisce un `long`, anche per gli

stream primitivi. Se i valori possono superare la capacità di `int`, non basta che il numero di elementi sia `long`: anche la **somma** deve usare un tipo o una strategia adatta. `mapToLong(...).sum()` può essere una scelta, ma un overflow di `long` resta comunque possibile per dati estremi.

`boxed()` converte `IntStream` in `Stream<Integer>` quando abbiamo bisogno di una collezione di wrapper o di un'API per oggetti. `mapToInt` percorre la direzione opposta da `Stream<T>` verso un valore numerico estratto o calcolato. I due passaggi hanno un costo e un significato: se facciamo `mapToInt(...).boxed()` immediatamente senza ragione, probabilmente stiamo complicando il percorso. Nel programma, `mapToInt(Libro::pagine)` è motivato da statistiche sulle pagine, mentre i libri restano oggetti fino al punto in cui il numero è necessario.

Gatherer: una trasformazione con memoria

Alcune trasformazioni hanno bisogno di ricordare elementi precedenti. `Gatherer` è un'astrazione per operazioni intermedie più ricche; la classe `Gatherers` offre implementazioni pronte. Le API sono stabili da Java 24 e presenti in Java 25, come indica la documentazione ufficiale di `Gatherers`. Non richiedono opzioni preview nel nostro programma.

Nel programma, `dati.stream().gather(Gatherers.windowFixed(3)).toList()` produce `[[1, 2, 3], [4, 5, 6], [3]]`. Ogni finestra raccoglie fino a tre elementi consecutivi; l'ultima può essere più corta. È un caso in cui una semplice `map` non basta, perché il risultato dipende dal raggruppamento di più input. `windowSliding` produce finestre sovrapposte; `scan` emette accumuli progressivi; `fold` costruisce un accumulo finale secondo il proprio contratto. `mapConcurrent` usa concorrenza controllata per trasformazioni adatte a quel modello, ma va affrontato insieme ai costi, agli effetti e alle regole del capitolo sulla concorrenza. Per imparare gli stream non occorre partire da un Gatherer personalizzato: prima si dominano sorgente, trasformazioni comuni e terminale.

Immagina che le tre misure di una finestra rappresentino i dati di un giorno. Con `windowFixed(3)` ogni misura appartiene a un solo gruppo, salvo la finestra finale incompleta. Se invece vogliamo osservare una media mobile, la misura di oggi deve comparire anche accanto a quelle dei giorni vicini: serve una finestra scorrevole e dobbiamo decidere che cosa fare quando mancano ancora abbastanza osservazioni. Prima di cambiare metodo, scriviamo i gruppi attesi per una sequenza piccola e confrontiamoli con l'output. Il criterio per riusare `Gatherer` è che la trasformazione dipenda da stato o da più elementi del flusso; per convertire ogni titolo in maiuscolo resta sufficiente `map`. Una soluzione più potente, applicata a un problema che non la richiede, rende soltanto più difficile prevedere il risultato.

Ordine d'incontro: la prima garanzia da dichiarare

Uno stream può avere un **ordine d'incontro** determinato dalla sorgente, come l'ordine di una `List`. Altre sorgenti non ne promettono uno. Alcune terminali e alcuni collector preservano un ordine, altri possono non farlo, specialmente in parallelo. `forEach` su uno stream parallelo non è lo strumento per richiedere una stampa ordinata; `forEachOrdered` mantiene l'ordine d'incontro quando esiste, ma questa garanzia può ridurre le possibilità di elaborazione parallela.

L'ordine d'incontro è una proprietà della sorgente che influenza la risposta a `findFirst` e la stampa con `forEachOrdered`. Non compare automaticamente perché abbiamo scritto una pipeline. Quando introdurremo il parallelismo, questa distinzione resterà decisiva: imporre un ordine può limitare il lavoro simultaneo. La responsabilità di chiudere una risorsa, invece, dipende dalla sorgente e verrà applicata alla pipeline su file alla fine del capitolo.

Parallelismo, ordine e risorse nel programma reale

Il modello degli stream permette di richiedere esecuzione parallela con `parallelStream()` o `parallel()`, ma non rende ogni pipeline più veloce. La libreria può dividere la sorgente in porzioni, lavorarle su più thread e combinare i risultati. Questo lavoro ha costi. Un `filter` molto breve su poche stringhe può andare peggio; un calcolo pesante su una sorgente grande e facilmente divisibile può beneficiare della macchina multicore. L'ordine richiesto e le operazioni stateful, come `sorted`, possono ridurre il vantaggio. Non esiste una soglia universale di elementi oltre cui “conviene”: bisogna misurare su dati e hardware rappresentativi.

Il parallelismo non significa che il chiamante ottenga subito il controllo come in una API asincrona. Una terminale come `toList()` ritorna quando il risultato è pronto, anche se il lavoro interno è stato distribuito. La pipeline deve inoltre rispettare le regole di **non interferenza**: le lambda non devono modificare la sorgente non concorrente durante l'attraversamento. Devono normalmente essere senza stato variabile fra elementi. Un `ArrayList` esterna a cui ogni elemento aggiunge un risultato è problematica: in parallelo gli aggiornamenti possono correre l'uno contro l'altro, e anche in sequenziale l'effetto è nascosto rispetto al contratto della terminale. `toList()` e i collector esprimono la raccolta senza quel rischio.

Una `List` possiede un ordine d'incontro; una `HashSet` non promette l'ordine in cui vedremo gli elementi. `findFirst` ha quindi un significato preciso quando la sorgente è ordinata; su una sorgente non ordinata non possiamo prevedere un primo elemento stabile. `findAny` comunica esplicitamente che qualunque elemento adatto è sufficiente. `forEach` su una pipeline parallela non promette che le azioni appaiano nell'ordine d'incontro, mentre `forEachOrdered` lo preserva quando esiste, pagando il costo di quel vincolo. Se l'ordine della stampa è parte del risultato, una lista ordinata raccolta e poi stampata può essere più semplice da verificare.

Uno stream creato da `Files.lines(percorso)` usa una risorsa del sistema operativo. Il blocco `try (Stream<String> righe = Files.lines(percorso)) { ... }` ne garantisce la chiusura anche in caso di eccezione. È un'applicazione diretta del capitolo 12: la pipeline elabora righe, il `try-with-resources` gestisce il ciclo di vita del file. `Stream.toList()` non sostituisce quel blocco. Al contrario, lo stream di una lista in memoria non richiede di chiudere un file. Il tipo `Stream` è lo stesso, ma la **sorgente** determina la responsabilità sulla risorsa.

> **Metodo per verificare una pipeline.** Scrivi prima la domanda in italiano e il risultato atteso su quattro elementi. Segna il tipo dopo ogni passaggio: `Stream<Libro>`, poi `Stream<String>`, poi `List<String>` oppure `IntStream` e infine un numero. Controlla l'ordine d'incontro se è rilevante, la presenza di effetti esterni e il caso di sorgente vuota. Cambia un dato e prevedi il nuovo risultato prima di eseguire. Solo dopo valuta se una pipeline parallela ha senso e misurala.

Una pipeline per una domanda reale

Supponiamo che il catalogo contenga schede di libri e che vogliamo i titoli dei volumi disponibili, senza spazi superflui e ordinati per la stampa. La domanda può diventare una pipeline: partire dalle schede, filtrare quelle disponibili, estrarre il titolo con `map`, pulirlo con `String::strip`, ordinarlo con `sorted` e raccoglierlo con `toList`. Ogni passaggio deve avere un significato distinto. Se la disponibilità può cambiare mentre la pipeline è in corso, occorre anche un contratto sullo snapshot dei dati o sulla sincronizzazione: lo stream non rende coerente una sorgente che altri stanno modificando senza regole.

Se dobbiamo soltanto trovare un libro disponibile, costruire e ordinare tutta la lista sarebbe lavoro inutile. Una ricerca con `filter(...).findFirst()` su una sorgente ordinata esprime meglio il risultato richiesto e può fermarsi al primo elemento adatto. Se l'ordine non conta, `findAny()` comunica quella libertà, soprattutto in parallelo. La differenza fra i due metodi è una scelta semantica, non una micro-ottimizzazione da applicare senza chiedersi quale risposta sia corretta.

Anche il ciclo tradizionale può restare la soluzione più leggibile. Se ogni elemento richiede una sequenza di controlli con errori differenti e aggiornamenti di stato, comprimere tutto in cinque lambda può nascondere il percorso che dobbiamo diagnosticare. La Stream API è forte quando le trasformazioni e le aggregazioni sono chiare come operazioni sui dati. Non è un obbligo stilistico imposto dal Java moderno.

Effetti e ordine di osservazione

`peek` permette di osservare elementi che attraversano una pipeline ed è utile soprattutto per diagnosi limitate. Non lo usiamo per aggiornare uno stato da cui dipende il risultato, perché un'operazione intermedia può non essere invocata come ci aspettiamo in ogni pipeline. `forEach` è terminale e produce un effetto per gli elementi elaborati, ma in parallelo l'ordine di esecuzione delle azioni non coincide necessariamente con l'ordine d'incontro. Se l'output deve avere un ordine stabile, una raccolta in lista seguita da una stampa ordinata è spesso più semplice da ragionare; `forEachOrdered` è disponibile quando quel vincolo appartiene davvero alla pipeline.

Una riduzione parallela richiede particolare cura con contenitori mutabili. Scrivere da più lambda nella stessa `ArrayList` esterna non è un modo corretto di raccogliere risultati: introduce stato condiviso e può perdere dati o produrre comportamenti imprevedibili. Un collector adatto gestisce l'accumulo secondo il proprio contratto. Ma anche un collector che funziona in parallelo non garantisce che la pipeline sarà più rapida. Il criterio resta prima correttezza, poi misurazione.

> **Concetto chiave – Dichiarare le garanzie.** Per ogni pipeline chiediamoci quale sia la sorgente, se l'ordine conta, quali funzioni leggono soltanto i dati, quale terminale produce il risultato e chi chiude eventuali risorse. Queste cinque domande sono più utili di una lista a memoria di tutti i metodi di `Stream`.

Una pipeline su file: dalla risorsa al risultato

Immaginiamo un file con una parola per riga e la richiesta di contare quelle che iniziano con `J`. La parte di calcolo può essere `righe.map(String::strip).filter(s -> s.startsWith("J")).count()`. Prima, però, dobbiamo aprire il file con charset e percorso adeguati e gestire un possibile errore I/O. `Files.lines(percorso)` restituisce uno stream collegato a una risorsa; il blocco `try-with-resources` è quindi parte della soluzione, non un'aggiunta facoltativa per i casi sfortunati. Il conteggio finale è un `long`; non serve materializzare tutte le righe in una lista.

Il programma completo `DemoStreamFile.java` prepara i dati in una cartella temporanea, dichiara esplicitamente UTF-8 e tiene l'apertura dello stream dentro `contaParoleConJ`. `try-with-resources` chiude la risorsa anche se il filtro o la lettura lanciano un'eccezione. La cartella viene rimossa nel blocco `finally`: il programma non lascia un file di prova nel progetto. Il caso mancante è gestito nel chiamante, dove si può decidere se informare l'utente, ritentare o interrompere il lavoro.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.NoSuchFileException;
import java.nio.file.Path;
import java.util.stream.Stream;

public class DemoStreamFile {
    private static long contaParoleConJ(Path percorso) throws IOException {
        try (Stream<String> righe = Files.lines(percorso, StandardCharsets.UTF_8)) {
            return righe.map(String::strip)
                    .filter(parola -> parola.startsWith("J"))
                    .count();
        }
    }
```

```
    public static void main(String[] args) throws IOException {
        Path cartella = Files.createTempDirectory("stream-file-");
        Path parole = cartella.resolve("parole.txt");
        Path vuoto = cartella.resolve("vuoto.txt");
        try {
            Files.writeString(parole, "Java\nReti\n JVM \n", StandardCharsets.UTF_8);
            Files.writeString(vuoto, "", StandardCharsets.UTF_8);
            System.out.println("Parole con J: " + contaParoleConJ(parole));
            System.out.println("File vuoto: " + contaParoleConJ(vuoto));
            try {
                contaParoleConJ(cartella.resolve("assente.txt"));
            } catch (NoSuchFileException errore) {
                System.out.println("File assente: " + errore.getClass().getSimpleName());
            }
        } finally {
            Files.deleteIfExists(parole);

            Files.deleteIfExists(vuoto);
            Files.deleteIfExists(cartella);
        }
    }
}
```

Compila con `javac --release 25 -Xlint:all -d build DemoStreamFile.java` ed esegui `java -cp build DemoStreamFile`. Ottieni `Parole con J: 2`, `File vuoto: 0` e `File assente: NoSuchFileException`. Il primo risultato segue le tre righe `Java`, `Reti`, `JVM` : `strip` elimina gli spazi esterni e la seconda parola con `J` diventa `JVM`. Il file vuoto non fornisce elementi alla pipeline. Nel terzo caso la risorsa non si apre: `NoSuchFileException` non è un conteggio pari a zero. La prova trasferisce il filtro già noto a una sorgente I/O e aggiunge responsabilità nuove, cioè charset, apertura, chiusura e gestione dell'assenza. Se leggiamo lo stesso file due volte per due domande, lo apriamo due volte oppure definiamo una strategia di conservazione dei dati; non riusiamo lo stream chiuso.

Per verificare

Compila `DemoStream.java` con `javac --release 25 -Xlint:all -d build DemoStream.java` ed eseguilo. Prima, prevedi le sei righe: la lista filtrata, la sorgente ancora integra, le parole appiattite, le finestre, il gruppo di lunghezza 4 e il risultato delle due aggregazioni. Compila anche `DemoCatalogoStream.java` e prevedi i titoli disponibili, il conteggio del reparto Informatica e la somma delle pagine. Poi cambia la finestra da 3 a 2 e spiega perché l'ultimo gruppo è più corto. Infine prova a conservare uno stream in una variabile e a invocare due volte `toList()` su di essa: spiega quale regola del ciclo di vita hai violato.

19. `Optional`: quando una ricerca non trova nulla

Il problema nascosto in una catena di chiamate

Immaginiamo di cercare un libro tramite il suo codice. Il catalogo può trovare la scheda, ma può anche non trovarla. Se il metodo restituisce `Libro` e usa `null` per dire «assente», chi lo chiama deve ricordarsi di controllare il risultato. Un solo controllo dimenticato può portare a una `NullPointerException` in una riga lontana dalla ricerca. Nel capitolo precedente abbiamo visto che `findFirst()` restituisce un valore che può non esserci: è lo stesso problema, espresso attraverso un tipo della libreria standard.

Alcune funzioni non possono produrre un risultato per ogni input. Una ricerca, per esempio, può non trovare nulla: spesso si restituisce `null` e le chiamate successive si riempiono di `if` per evitare un errore. In Java, `Optional<T>` rappresenta un risultato che contiene un valore non nullo di tipo `T` oppure è vuoto. Il tipo restituito avverte chi legge la firma che l'assenza è una possibilità ordinaria del dominio. Non elimina ogni controllo: lo colloca in un punto esplicito della composizione, dove possiamo scegliere se trasformare, fornire un'alternativa, lanciare un'eccezione o non fare nulla.

> **Definizione – Valore assente.** Un `Optional<Libro>` vuoto è un oggetto `Optional` esistente che descrive l'assenza di un libro. Non è `null`, e non contiene un libro il cui valore sia `null`. La variabile stessa non dovrebbe essere `null`: restituire `null` da un metodo dichiarato `Optional<Libro>` vanificherebbe il contratto. La documentazione Java 25 indica `Optional` soprattutto come tipo di ritorno di un metodo quando «nessun risultato» è una risposta normale e `null` sarebbe fonte di errori.

La composizione delle trasformazioni si collega al concetto di *monade*, introdotto nel capitolo 16. Per capire la ricerca, però, partiamo dal contratto concreto di `Optional`. Non attribuiamo a `Optional` il potere di correggere automaticamente un programma: un risultato vuoto può ancora essere ignorato, un effetto collaterale può ancora nascondersi in una funzione, e un `get()` sul valore vuoto può ancora lanciare un'eccezione. La proprietà concreta che ci interessa è la composizione di trasformazioni che mantengono visibile l'assenza.

Costruire i due casi

La prima distinzione da padroneggiare è fra `Optional.empty()`, `Optional.of(valore)` e `Optional.ofNullable(valore)`. La prima forma produce un risultato vuoto. La seconda richiede un valore non nullo e lancia `NullPointerException` se gli passiamo `null`: è adatta quando il programma sa che il valore deve esserci e vuole scoprire subito la violazione dell'invariante. La terza accetta un riferimento che può essere `null`: se lo è, restituisce un `Optional` vuoto, altrimenti un `Optional` presente.

Nel programma completo, il metodo `cerca` interroga una piccola `Map` del catalogo. `Map.get(codice)` restituisce il libro associato oppure `null` se la chiave non esiste. Proprio al confine con questa API, il metodo converte il possibile `null` in un `Optional<Libro>`:

```
static Optional<Libro> cerca(String codice) {
    return Optional.ofNullable(CATALOGO.get(codice));
}
```

Quando cerchiamo `"J-01"`, il valore è presente; quando cerchiamo `"X"`, il risultato è vuoto. `cerca("X").isEmpty()` stampa `true`. Si può chiedere anche `isPresent()`, che nel medesimo caso darebbe `false`. Sono domande sul risultato, non metodi per estrarre direttamente il libro. Possiamo provarle entrambe, ma se l'intero percorso diventasse una successione di `if (isPresent())` seguiti da `get()`, torneremmo a gestire manualmente ogni ramo nel punto meno espressivo.

> **Nota bene – Vuoto non significa errore.** Se il codice libro non compare nel catalogo, «nessun libro» può essere una risposta prevista. Se invece il catalogo è indisponibile per un guasto, restituire `Optional.empty()` farebbe sembrare assente un libro che non abbiamo potuto cercare. Quel guasto richiede un altro canale, per esempio un'eccezione o un tipo di risultato che distingua assenza e fallimento. La forma del tipo deve corrispondere alle risposte reali dell'operazione.

Presenza, azione e valore finale

Quando vogliamo compiere un'azione soltanto se il valore esiste, `ifPresent` riceve un `Consumer<T>`. Nel nostro caso potremmo scrivere `cerca("J-01").ifPresent(libro -> System.out.println(libro.titolo()));`. Se la ricerca è vuota, la lambda non viene eseguita. `ifPresentOrElse` accetta una seconda azione per il caso vuoto: è utile quando entrambi i rami sono effetti, come stampare un messaggio o aggiornare un'interfaccia. Non restituisce un titolo; se dobbiamo produrre un valore per un'espressione successiva, è più adatto un metodo della famiglia `orElse`.

`orElse("assente")` restituisce il titolo presente oppure il testo di ripiego. La prima riga del programma passa da `Optional<Libro>` a `Optional<String>` con `map(Libro::titolo)` e poi a `String` con `orElse("assente")`: stampa `Java`. La chiamata non costruisce un libro fittizio quando la ricerca fallisce; sceglie soltanto quale stringa mostrare al lettore. Se il chiamante deve conservare la distinzione fra titolo vero e titolo non trovato, non deve applicare troppo presto `orElse`: può mantenere `Optional<String>` fino al confine in cui serve una risposta concreta.

Il metodo `get()` estrae il valore presente ma lancia `NoSuchElementException` se il risultato è vuoto. Una traccia d'errore ci aiuterebbe a vedere il punto del fallimento, ma non renderebbe sicura la chiamata. In Java 25 `get()` è ancora nell'API; la documentazione indica `orElseThrow()` come alternativa preferita quando vogliamo esprimere esplicitamente che l'assenza, in quel punto, è eccezionale. La differenza non è un trucco per evitare un'eccezione: `orElseThrow()` senza argomenti lancia anch'esso `NoSuchElementException` sul vuoto. Il nome fa leggere meglio la decisione. Se invece abbiamo un'alternativa valida, scegliamo quella; se non dobbiamo fare nulla, manteniamo l'assenza.

Un ripiego calcolato subito o soltanto quando serve?

Il confronto fra `orElse` e `orElseGet` merita un esperimento completo. Consideriamo `Optional.of("Java")`: il valore esiste, quindi entrambi i metodi restituiscono `"Java"`. Eppure la valutazione dell'argomento è diversa. In `presente.orElse(nomeDiDefault())`, Java calcola `nomeDiDefault()` **prima** di entrare in `orElse`, perché gli argomenti di un metodo vengono valutati prima della chiamata. In `presente.orElseGet(DemoOptional::nomeDiDefault)`, passiamo una funzione; `Optional` la invoca soltanto se è vuoto.

Nel programma `nomeDiDefault()` incrementa un contatore per rendere osservabile la chiamata. La riga `default calcolati: 1` mostra che il ripiego di `orElse` è stato calcolato anche se non usato, mentre quello di `orElseGet` non è stato calcolato. Se l'alternativa fosse una costante come `"sconosciuto"`, la forma `orElse` sarebbe semplice e leggibile. Se richiedesse una query, una lettura da file o un'operazione costosa, `orElseGet` eviterebbe lavoro inutile quando il valore è presente. L'effetto del nostro contatore serve solo alla prova: in codice applicativo una funzione chiamata come ripiego dovrebbe avere un comportamento documentato e non dipendere da un ordine di chiamata nascosto.

Forma	Argomento passato	Quando si calcola il ripiego	Risultato con valore presente
`orElse(alternativa)`	un valore `T`	prima della chiamata a `orElse`	restituisce il valore presente
`orElseGet(fornitore)`	un `Supplier<T>`	solo se l'`Optional` è vuoto	restituisce il valore presente senza invocare il fornitore

La tabella non dice che `orElseGet` sia sempre migliore. Se il valore alternativo è già disponibile, creare una lambda per restituirlo non aggiunge chiarezza. D'altro canto, una chiamata a un servizio remoto non diventa sicura solo perché la mettiamo in un `Supplier`: quando l'`Optional` è vuoto, l'operazione avverrà comunque e potrà fallire. Il contratto del metodo deve spiegare che cosa accade in quel caso.

Quando l'assenza deve diventare un'eccezione

Il catalogo di un negozio può rispondere tranquillamente «libro non trovato» a una ricerca pubblica. La stessa risposta può essere inaccettabile mentre prepariamo una spedizione già pagata: in quel punto il codice si aspetta che il libro esista. `orElseThrow()` traduce questa decisione in modo evidente. Senza argomenti lancia `NoSuchElementException`; con un `Supplier` possiamo costruire un'eccezione più pertinente al dominio. Nell'esempio completo, `cerca("X").orElseThrow(() -> new LibroAssenteException("X"))` produce un errore che porta con sé il codice cercato. Il fornitore dell'eccezione non viene invocato se la ricerca ha successo.

```
Libro libro = cerca(codice)
        .orElseThrow(() -> new LibroAssenteException(codice));
```

Qui la variabile `libro` ha tipo `Libro`, non `Optional<Libro>`: dopo questa riga il flusso normale prosegue solo con un libro presente. Se chiamiamo `orElseThrow` subito dopo ogni ricerca, però, abbiamo perso la ragione per cui abbiamo scelto `Optional`: l'assenza ordinaria diventa sistematicamente un'eccezione. Il metodo appartiene ai punti in cui il contratto cambia, non a ogni riga che vuole leggere un valore.

> **Approfondimento – Eccezione controllata.** La versione con fornitore è generica anche rispetto al tipo dell'eccezione. Possiamo fornire un'eccezione controllata e dichiararla nella firma del metodo chiamante, oppure usarne una non controllata. La scelta dipende dal contratto dell'operazione. Una ricerca fallita non obbliga di per sé a scegliere una delle due famiglie: prima si decide se è davvero un fallimento in quel contesto.

È facile confondere `orElseThrow` con `get()` perché entrambi possono estrarre il valore e fallire sul vuoto. La differenza pratica è la leggibilità della scelta; la versione con fornitore permette anche di dare al fallimento un significato specifico. Il fallimento con `NoSuchElementException` è utile da osservare in un esperimento: si può eseguire `Optional.<Libro>empty().get()` in un test isolato, osservare l'eccezione e poi sostituirlo con una gestione intenzionale. Non mettiamo quell'esperimento nel flusso normale del programma, dove interromperebbe gli esempi successivi.

Una seconda ricerca con `or`

Talvolta vogliamo conservare il risultato come `Optional` ma cercare altrove se il primo catalogo non contiene il libro. `or` riceve un `Supplier<Optional<? extends T>>`: il fornitore deve restituire un altro `Optional`, non un `Libro` nudo. Se il primo è presente, il fornitore non viene chiamato e il risultato è il primo `Optional`. Se è vuoto, viene usato il risultato della seconda ricerca, che può essere ancora vuoto. Non è dunque una promessa di successo; è un piano di ricerca alternativo.

Nel programma la ricerca di `"R-02"` nel primo catalogo è vuota; il fornitore consulta il catalogo di riserva e trova *Reti*. La riga di output è `Reti`. Se la prima ricerca trovasse già un libro, non pagheremmo il costo della seconda consultazione. Se entrambe fallissero, il successivo `map(Libro::titolo).orElse("assente")` renderebbe esplicito il valore da mostrare.

```
String titolo = cerca("R-02")
        .or(() -> cercaNelCatalogoDiRiserva("R-02"))
        .map(Libro::titolo)
        .orElse("assente");
```

Il contrasto con `orElseGet` chiarisce i tipi. `orElseGet` esce da `Optional` e restituisce un `T`; `or` conserva l'involucro e restituisce un `Optional<T>`. Se la fase successiva deve ancora distinguere il caso assente, `or` mantiene l'informazione. Se invece occorre un valore concreto alla fine del percorso, `orElseGet` è adatto. Il codice diventa leggibile quando ogni operazione corrisponde a una decisione reale del dominio, non quando si sommano metodi solo per evitare un `if`.

Filtrare un risultato trovato

Supponiamo che il catalogo contenga il libro, ma la schermata debba mostrare soltanto volumi disponibili. `filter` non interroga il catalogo: riceve l'`Optional` già prodotto e verifica un `Predicate` sul valore presente. Se il risultato era vuoto, non chiama il predicato e resta vuoto. Se il risultato era presente ma il predicato restituisce `false`, diventa vuoto. Se restituisce `true`, il valore rimane presente.

```
Optional<Libro> disponibile = cerca("J-01")
        .filter(Libro::disponibile);
```

La semplicità di questa forma ha un prezzo informativo: dopo `filter` un risultato vuoto può significare «codice inesistente» oppure «libro esistente ma indisponibile». Se il lettore dell'applicazione deve distinguere i due casi, non dobbiamo fonderli. Possiamo modellare i due esiti con un tipo dedicato o mantenere la verifica della disponibilità nel ramo in cui il libro è presente. È un esempio importante del limite di `Optional`: trasporta presenza o assenza, non il motivo di ogni assenza.

Il predicato va scelto per il suo significato e deve essere comprensibile al punto d'uso. `filter(libro -> libro.prezzo() > 0)` ha senso solo se abbiamo spiegato perché il prezzo positivo è la condizione desiderata. In generale, un predicato con effetti collaterali rende meno facile ragionare sulla composizione; soprattutto, non affidiamoci a `filter` per modificare il libro mentre fingiamo di limitarci a verificarlo.

Cambiare il contenuto con `map`

Con `map` trasformiamo il valore presente e lasciamo invariato il caso vuoto. Da `Optional<Libro>` e una funzione `Libro -> String` otteniamo `Optional<String>`. In `cerca("J-01").map(Libro::titolo)` la funzione viene chiamata sul libro trovato. In `cerca("X").map(Libro::titolo)` non viene chiamata, perché non c'è un libro a cui applicarla. Possiamo proseguire con altre trasformazioni senza introdurre un controllo a ogni passaggio:

```
String titoloPerSchermo = cerca("J-01")
        .map(Libro::titolo)
        .map(String::toUpperCase)
        .orElse("NON DISPONIBILE");
```

La catena va letta da sinistra a destra. Prima cerchiamo un libro; se lo troviamo, ne prendiamo il titolo; se il titolo esiste, lo convertiamo in maiuscolo; infine scegliamo cosa mostrare quando l'intero percorso non ha prodotto una stringa. Il valore di ripiego compare una sola volta, alla fine, invece di essere duplicato in ogni trasformazione.

`map` applica internamente una logica equivalente a `ofNullable` al risultato della funzione: se la funzione restituisce `null`, la trasformazione produce `Optional.empty()`. È una proprietà dell'API, non un invito a restituire `null` senza spiegarlo. In un modello curato, un titolo obbligatorio non dovrebbe diventare `null`; se accade, è più utile correggere l'invariante che nascondere il problema

dietro un vuoto. Quando invece la funzione interroga davvero un dato facoltativo, conviene dichiarare quella facoltatività nel suo tipo di ritorno e usare `flatMap`.

> **Schema di lettura – `map`.** `Optional<Libro>` seguito da `Libro -> String` produce `Optional<String>`. Il contenitore mantiene la possibilità di assenza, mentre cambia il tipo del valore eventualmente presente. La freccia descrive una funzione: riceve un `Libro` e restituisce una `String`; non indica che il libro venga modificato.

Comporre due ricerche con `flatMap`

Un autore può avere un nome facoltativo. Nel nostro modello, `autore(Libro)` restituisce `Optional<String>`. Se usassimo `map(DemoOptional::autore)`, la trasformazione avvolgerebbe un risultato già facoltativo e otterremmo `Optional<Optional<String>>`. Per leggere il nome dovremmo aprire due livelli: il libro potrebbe mancare e, anche se presente, il nome potrebbe mancare. `flatMap` applica la funzione e appiattisce il risultato in un solo `Optional<String>`.

```
Optional<String> nomeAutore = cerca("J-01")
        .flatMap(DemoOptional::autore);
```

Seguiamo una catena di ricerche: dalla chiave troviamo forse un oggetto; dall'oggetto troviamo forse un dato; soltanto al termine decidiamo come rappresentare il vuoto. Se la ricerca del libro è vuota, `autore` non viene chiamato. Se il libro esiste ma il campo autore non è disponibile, `autore` restituisce `Optional.empty()` e il risultato della catena è vuoto. Se entrambi esistono, il nome viene passato al passo seguente. Nel nostro programma il nome ottenuto è `Ada`.

`flatMap` richiede che la funzione restituisca un `Optional` non nullo. Scrivere una funzione che restituisce `null` invece di `Optional.empty()` viola il contratto e può provocare una `NullPointerException`. È coerente con l'idea del metodo: il risultato deve poter dire in modo esplicito se il secondo passo ha trovato qualcosa. `map` e `flatMap` non sono varianti stilistiche intercambiabili; scegliamo guardando il tipo restituito dalla funzione che passiamo.

> **Approfondimento – Due assenze, un solo tipo.** Appiattire non conserva automaticamente la causa. `Optional.empty()` dopo `flatMap` può significare che mancava il libro o che mancava il suo autore. Quando la causa conta, un semplice `Optional` non basta. Si può mantenere la prima decisione in un ramo esplicito, oppure definire un tipo risultato che descriva i casi. La composizione è utile quando entrambi i vuoti conducono davvero alla stessa scelta finale.

Da un risultato facoltativo a uno stream

`Optional.stream()` produce uno stream con zero elementi quando il risultato è vuoto e con un elemento quando è presente. Questo ponte diventa utile quando abbiamo *molti* risultati facoltativi. Una lista di codici può generare una sequenza di `Optional<Libro>`; `flatMap(Optional::stream)` elimina i risultati vuoti e lascia scorrere i libri trovati. Nel programma, i codici `"J-01"`, `"X"` e `"R-02"` vengono cercati prima nel catalogo principale e poi, se serve, in quello di riserva. L'output è `[Java, Reti]`.

```
List<String> titoli = codici.stream()
        .map(codice -> cerca(codice)
                .or(() -> cercaNelCatalogoDiRiserva(codice)))
        .flatMap(Optional::stream)
        .map(Libro::titolo)
        .toList();
```

Qui i due `flatMap` che abbiamo incontrato appartengono a tipi diversi. Quello di `Optional` compone una funzione che restituisce un altro `Optional`; quello di `Stream` prende, per ciascun elemento, uno stream e ne concatena gli elementi nel flusso risultante. `Optional::stream` è l'adattatore fra le due forme. Questo dettaglio evita l'impressione che si tratti di una formula da memorizzare: la catena rappresenta tre scelte comprensibili, cercare, ignorare le assenze e prendere i titoli.

Una stream pipeline è consumata dall'operazione terminale `toList()`. Prima di quella chiamata le trasformazioni sono definite ma non hanno ancora prodotto la lista. Per una sola ricerca, costruire uno stream sarebbe prolisso; per una collezione di ricerche consente di esprimere senza sentinelle la regola «mantieni soltanto i successi». Se la posizione dei codici assenti o il motivo dell'assenza è importante, non scartiamoli con `flatMap(Optional::stream)`: conserviamo un risultato per ogni codice.

Dove conviene usare `Optional`

Un metodo di ricerca che può non trovare un oggetto è un buon candidato per `Optional<T>`. Una variabile locale che rappresenta un esito già prodotto può esserlo quando rende il flusso più chiaro. La documentazione dell'API lo presenta soprattutto come tipo di ritorno, non come sostituto universale di ogni riferimento. Un campo `Optional<Libro>` in un'entità, un parametro `Optional<Libro>` imposto a tutti i chiamanti o una collezione piena di `Optional` possono aggiungere un involucro dove basterebbe un contratto più diretto. Sono scelte da valutare nel contesto, non divieti assoluti.

`Optional` è una classe basata sul valore. Non confrontiamo due istanze con == per stabilire se descrivono lo stesso risultato e non usiamole come monitor di sincronizzazione. Per confrontare i contenuti, `equals` rispetta il contratto documentato. Non assumiamo che `Optional.empty()` sia sempre il medesimo oggetto in memoria: ciò che conta è che sia vuoto. L'API offre anche `OptionalInt`, `OptionalLong` e `OptionalDouble` per risultati facoltativi primitivi, evitando il boxing del valore quando quella scelta è appropriata; il modo di comporli non coincide sempre con l'API generica e va letto nei rispettivi contratti.

Proviamo infine a formulare il contratto prima di scegliere il metodo. «Il codice potrebbe non esistere» suggerisce `Optional<Libro>`. «Se manca nel primo catalogo, consulta il secondo» suggerisce `or`. «Se esiste, mostra il titolo» suggerisce `map` e un valore finale per la vista. «Se manca durante una spedizione già confermata, l'operazione è incoerente» suggerisce `orElseThrow` nel punto della spedizione. Così `Optional` non diventa una catena decorativa: rende esplicite le decisioni che in una sequenza di `null` e `if` resterebbero sparse.

Una difficoltà frequente emerge quando il ripiego è esso stesso assente. `orElse(null)` è permesso dal metodo, ma riconduce il chiamante alla necessità di gestire `null`; non abbiamo guadagnato chiarezza. `orElseGet` può a sua volta ricevere un fornitore che restituisce `null`, e `or` può restituire un `Optional` vuoto. Prima di scrivere la catena, chiediamoci se il punto di arrivo ammette ancora l'assenza. Se sì, manteniamo `Optional` e lasciamo la decisione al livello che possiede il contesto; se no, scegliamo un ripiego significativo oppure un'eccezione appropriata. Il valore di ripiego non deve fingere che la ricerca sia riuscita: una stringa come `"sconosciuto"` può andare bene per una vista, ma sarebbe un pessimo codice ISBN da salvare come dato reale.

Intenzione del chiamante	Metodo di partenza	Tipo del risultato
Sapere se il valore esiste	`isPresent`, `isEmpty`	`boolean`
Eseguire un'azione quando esiste	`ifPresent`	`void`
Eseguire una delle due azioni	`ifPresentOrElse`	`void`
Trasformare un valore presente	`map`	`Optional<U>`
Comporre una ricerca facoltativa	`flatMap`	`Optional<U>`
Escludere un valore che non soddisfa una regola	`filter`	`Optional<T>`
Cercare un'altra fonte	`or`	`Optional<T>`
Ottenere un valore o un ripiego	`orElse`, `orElseGet`	`T`
Ottenere un valore o segnalare un errore	`orElseThrow`	`T`
Inserire il risultato in una pipeline	`stream`	`Stream<T>`

La tabella è una guida alla lettura dei tipi, non una scorciatoia per scegliere il metodo solo dal nome. Se si parte da `Optional<Libro>` e si applica `map(Libro::titolo)`, la lettera `U` della tabella è `String`. Se si applica `flatMap` a un metodo che restituisce `Optional<String>`, il risultato resta un solo `Optional<String>`. Questo piccolo esercizio di tipi permette di controllare la catena

prima ancora di eseguirla e di individuare subito un `Optional<Optional<...>>` comparso per errore.

> **Per verificare la comprensione.** Si modifichi il programma aggiungendo un libro presente senza autore e si osservi il risultato di `flatMap(DemoOptional::autore)`. Poi si sostituisca `orElseGet` con `orElse` per una ricerca vuota e una presente, annotando quante volte viene calcolato il ripiego. Infine si decida se un codice inesistente e un catalogo indisponibile debbano produrre la stessa risposta. La spiegazione della scelta conta più della sintassi usata.

Una migrazione ragionata da `null`

Immaginiamo un metodo esistente `Libro trova(String codice)` che restituisce `null` quando la chiave non è nel catalogo. Prima di cambiare la firma chiediamoci chi lo chiama e che cosa fa con il risultato. Un chiamante usa `if (libro != null)` per mostrare la scheda; un altro passa subito il risultato a un metodo che non accetta `null`; un terzo interpreta il vuoto come un errore. Cambiare la firma in `Optional<Libro> cerca(String codice)` rende esplicita la possibilità di assenza, ma non rende identiche le tre decisioni. Ogni chiamante deve scegliere il proprio punto di uscita: mostrare un messaggio, richiedere un'altra fonte oppure segnalare l'incoerenza con un'eccezione.

La migrazione va fatta ai confini dell'API che conosce il significato di `null`. Nel nostro esempio quel confine è `Map.get`: `Optional.ofNullable(CATALOGO.get(codice))` traduce la risposta della mappa. Non è utile disseminare `Optional.ofNullable` dopo ogni chiamata se il metodo precedente ha già promesso un risultato non nullo; in quel caso un `null` inatteso segnala un contratto rotto. Al contrario, se una libreria esterna usa `null` come esito ordinario, la conversione in un `Optional` nel nostro adapter rende il resto del codice più leggibile. La scelta richiede di conoscere il contratto della fonte, non soltanto la sintassi di `ofNullable`.

Un cambiamento della firma è visibile anche nei test. Prima un test poteva asserire che la ricerca assente restituisse `null`; dopo deve asserire che l'`Optional` sia vuoto. Per il caso presente può verificare che il titolo sia quello atteso, senza basarsi sull'identità dell'oggetto `Optional`. Un terzo test dovrebbe chiedere cosa accade quando il catalogo stesso fallisce: restituire vuoto in quel caso sarebbe una falsa risposta «codice non trovato». Una buona suite misura queste tre risposte distinte, perché sono proprio quelle che i chiamanti devono saper gestire.

> **Approfondimento – Assenza di un numero.** Se una ricerca calcola facoltativamente un `int`, `Optional<Integer>` può descrivere l'esito, ma comporta il boxing del valore primitivo. `OptionalInt` rappresenta presenza o assenza di un `int` senza usare `Integer` come contenuto. Ha metodi propri, per esempio `empty`, `of`, `isPresent`, `orElse` e `stream`; non assumiamo che ogni metodo generico di `Optional<T>` abbia una copia identica. La scelta fra le due forme si fa per chiarezza del contratto e profilo del lavoro, non per trasformare ogni numero del programma in un involucro facoltativo.

Due casi ulteriori: email e studente

Il catalogo ci ha permesso di seguire un solo problema per tutto il capitolo. Due esempi indipendenti ci aiutano a verificare che i metodi non dipendano dal nome `Libro`: una email facoltativa e l'età di uno studente. Nel programma `StudenteEFiltri`, un piccolo `record Studente(String nome, Integer eta)` rende visibili le proprietà necessarie senza richiedere librerie aggiuntive. La scelta del record non cambia le regole di `Optional`; serve soltanto a concentrare l'attenzione sul valore che può mancare.

Nel primo caso un `Optional<String>` contiene un indirizzo email. `filter` applica un predicato costruito con una espressione regolare. Con `ada@example.org` il predicato restituisce `true` e l'`Optional` resta presente; con `ada.example.org` restituisce `false` e il risultato è vuoto. Il programma stampa `true` e `false`. Il predicato controlla qui una forma elementare, non certifica che la casella esista o che l'indirizzo soddisfi ogni regola possibile di consegna. Questa precisazione conta: `filter` applica esattamente la condizione che gli diamo, non una validazione universale nascosta nella classe `Optional`.

```
Optional<String> email = Optional.of("ada@example.org");
boolean formaAccettata = email
        .filter(valore -> FORMA_EMAIL.matcher(valore).matches())
        .isPresent();
```

Se il valore iniziale fosse vuoto, il predicato non verrebbe eseguito e `isPresent()` restituirebbe comunque `false`. Dopo questa pipeline, dunque, `false` può significare «non era stato fornito alcun indirizzo» oppure «l'indirizzo fornito non rispetta la forma scelta». Se la schermata deve mostrare due messaggi diversi, occorre conservare la distinzione prima del filtro. Lo stesso limite era apparso con la disponibilità del libro: una trasformazione che porta più cause al medesimo vuoto è corretta solo quando quelle cause hanno davvero lo stesso esito per il chiamante.

Nel secondo caso il campo `eta` è di tipo `Integer` e può essere nullo nel piccolo modello didattico. `studente.map(Studente::eta)` produce `Optional<Integer>`: quando il campo è presente, il nuovo `Optional` contiene il numero; quando il campo è nullo, il risultato è vuoto. Così passiamo da `Optional<Studente>` a `Optional<Integer>` senza perdere il significato dell'assenza. Il programma stampa `eta presente: true` per lo studente di vent'anni. Se cambiamo la costruzione in `new Studente("Ada", null)`, la stampa diventa `false`. Non abbiamo «riparato» un'età obbligatoria mancante: abbiamo descritto un modello in cui l'età può davvero non essere nota. Se nel dominio l'età è richiesta, il record dovrebbe impedirne l'assenza invece di affidare l'errore a una pipeline.

Il metodo `nomeFacoltativo(Studente)` restituisce `Optional<String>`. Partendo da `Optional<Studente>`, `flatMap(StudenteEFiltri::nomeFacoltativo)` evita il doppio involucro e produce direttamente `Optional<String>`. Il programma ne estrae `Ada` con un ripiego dichiarato per il caso vuoto. In questo modo l'esempio dello studente mostra, con dati diversi dal catalogo, la ragione dei tipi `Optional<Integer>` e `Optional<String>` e la differenza fra `map` e `flatMap`. Il lettore può disegnare i due percorsi come trasformazioni di tipi prima di guardare l'output.

Un parametro facoltativo è sempre una buona idea?

Che cosa cambia se il valore facoltativo è un parametro del metodo che filtra gli studenti? La documentazione presenta `Optional` soprattutto come tipo di ritorno; un parametro `Optional<Integer>` obbliga ogni chiamante a costruire un involucro per dire «nessun filtro» o «filtra per questa età». Nel programma usiamo un parametro `Integer eta` che può essere `null` *al confine di quel metodo*, e traduciamo la scelta in una condizione esplicita: `eta == null || eta.equals(studente.eta())`. È un esempio locale di contratto, non una raccomandazione a diffondere `null` senza controllo nel resto del programma.

Se i criteri crescono, per esempio nome, età minima, sezione e disponibilità, molti parametri facoltativi diventano difficili da leggere. Un tipo `CriteriRicerca` con campi e regole dichiarate può essere più chiaro; un builder può permettere al chiamante di specificare soltanto i criteri desiderati. La decisione dipende dall'API e dai suoi chiamanti. La regola didattica da conservare non è «mai `Optional` nei parametri» come divieto assoluto, ma «usalo quando il tipo chiarisce davvero il contratto e non obbliga a un involucro superfluo». Anche qui la chiarezza della chiamata conta quanto la concisione del corpo del metodo.

> **Esercizio di lettura.** Si prenda la lista con due studenti chiamati Ada, uno di venti e uno di ventuno anni. `filtraStudenti(lista, "Ada", 20)` restituisce un solo elemento. Con il terzo argomento `null` il metodo restituisce entrambi. Il lettore deve spiegare perché la condizione `eta == null || eta.equals(studente.eta())` evita di dereferenziare il parametro assente e perché il confronto del nome resta necessario in entrambi i casi. Poi può decidere se questo contratto è abbastanza chiaro per l'applicazione immaginata o se merita un tipo di criteri dedicato.

Per verificare

Riprendi il catalogo e distingui tre esiti: codice trovato, codice assente e catalogo non disponibile. Scrivi quale tipo restituiresti per una ricerca riuscita o assente e dove renderesti visibile il guasto della fonte. Solo dopo scegli fra `map`, `flatMap`, `or` e `orElseThrow` per costruire una risposta al chiamante: per ogni passaggio annota il tipo del valore prima e dopo. Una prova con un codice presente e una con un codice assente devono confermare le tue previsioni; una terza prova deve impedire che il guasto venga presentato come semplice assenza. L'attività del capitolo 19 permette di trasferire la stessa distinzione a un problema nuovo.

20. Thread e concorrenza in Java

Quando un programma svolge più lavori

Un'applicazione può dover attendere una risposta dalla rete mentre continua a servire altre richieste, leggere dati mentre aggiorna una vista o distribuire una computazione su più unità di elaborazione. Un programma concorrente sembra trovarsi logicamente in più punti nello stesso periodo. Questa immagine aiuta a iniziare, purché non ci faccia immaginare un ordine unico delle istruzioni. Le attività concorrenti avanzano secondo tempi che il nostro programma non controlla del tutto. Su più core possono eseguire davvero nello stesso istante; su un solo core possono alternarsi abbastanza rapidamente da fare progressi nello stesso intervallo di tempo.

Un *processo* è un'istanza di programma con proprie risorse gestite dal sistema operativo. I thread di un processo condividono parte dello spazio di memoria e possono accedere agli stessi oggetti Java. Passare loro un riferimento permette di raggiungere lo stesso dato, ma non basta a comunicare correttamente: se un thread lo modifica e un altro lo legge, dobbiamo stabilire quando la modifica può essere osservata. La memoria condivisa non richiede di trasferire ogni valore tramite un messaggio fra processi, ma la comunicazione conserva costi e regole di sincronizzazione. Due incrementi sullo stesso contatore, per esempio, possono perdere un aggiornamento se non sono coordinati; lo vedremo nel programma del capitolo. Scegliere fra più processi e più thread richiede anche di valutare isolamento, affidabilità e distribuzione, oltre al costo di avvio e al lavoro necessario per condividere i dati.

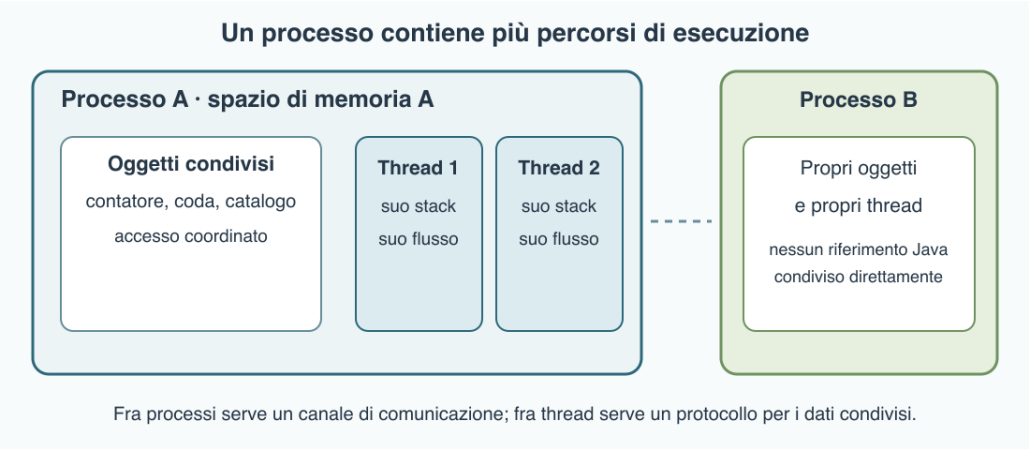


Figura 20.1 – I thread condividono gli oggetti del loro processo, ma mantengono percorsi di esecuzione distinti. I processi hanno spazi separati.

Definizione – Concorrenza e parallelismo. Concorrenza significa che più attività sono in corso e possono progredire in un periodo sovrapposto. Parallelismo significa che almeno due operazioni vengono eseguite materialmente nello stesso istante su risorse distinte. Un programma concorrente può essere eseguito senza parallelismo; un programma parallelo è necessariamente concorrente. Questa distinzione impedisce di promettere un'accelerazione solo perché abbiamo creato due thread.

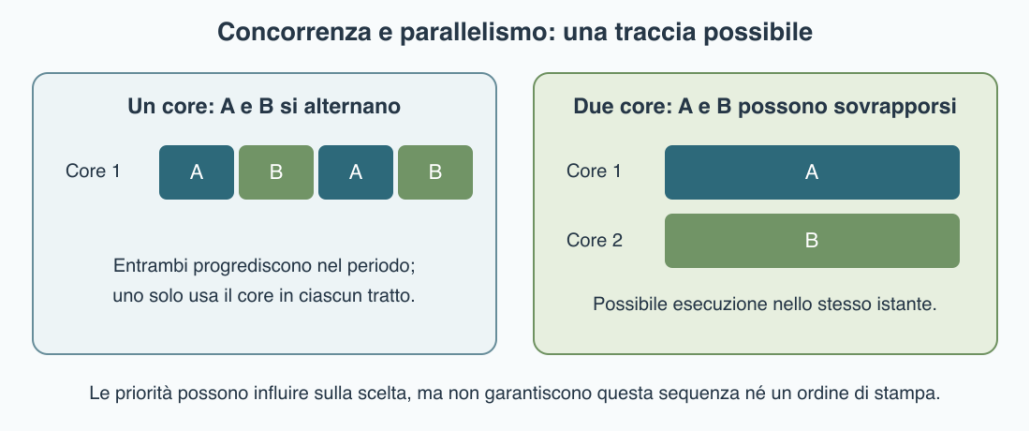


Figura 20.2 – Due possibili esecuzioni: i riquadri non promettono turni fissi. La priorità non determina l'ordine del programma.

I thread offrono vantaggi, ma comportano anche costi: una risposta può rimanere pronta mentre un'altra operazione attende; il lavoro su dati indipendenti può sfruttare più core; risorse condivise possono essere riutilizzate. Ma ogni accesso condiviso può introdurre race condition, attese, deadlock o risultati difficili da riprodurre. Aumentare il numero dei thread non rende automaticamente il programma più veloce. Se il collo di bottiglia è una singola risorsa seriale, aggiungere lavoro concorrente può peggiorare il tempo totale.

Dalla richiesta di avvio alla fine

L'oggetto `Thread` rappresenta un thread di esecuzione. Un `Runnable` descrive il lavoro da svolgere e non restituisce un valore. Quando chiamiamo `start()`, chiediamo l'avvio di un nuovo thread che eseguirà `run()`. Chiamare direttamente `run()` esegue invece il metodo nel thread corrente: è una normale chiamata di metodo. Questa differenza è il primo esperimento da compiere, stampando `Thread.currentThread().getName()` nei due casi. Nel codice moderno spesso è più chiaro separare il lavoro dalla gestione del thread, usando una lambda `Runnable` anziché estendere `Thread`, a meno che la sottoclasse rappresenti davvero un comportamento specifico.

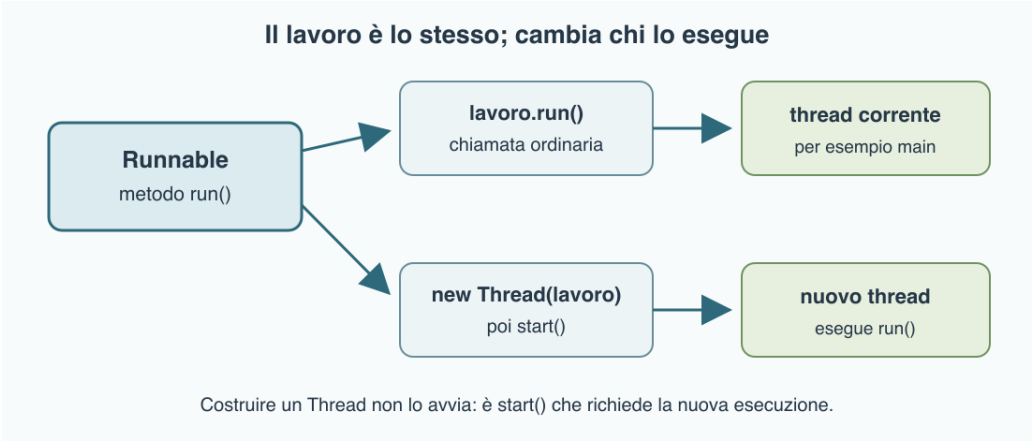


Figura 20.3 – `run()` chiamato direttamente resta nel chiamante; `start()` richiede un nuovo thread che eseguirà lo stesso lavoro.

L'API espone gli stati `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING` e `TERMINATED`. `NEW` descrive un oggetto non avviato; `TERMINATED` un'esecuzione conclusa. `BLOCKED` indica attesa per acquisire un monitor; `WAITING` e `TIMED_WAITING` coprono forme di attesa senza o con limite temporale. `RUNNABLE` include sia il thread eseguito dal processore sia quello pronto a essere scelto. A volte si chiama «running» il thread che sta usando la CPU, ma `RUNNING` non compare nell'enumerazione `Thread.State`: teniamo separato lo stato pubblico dalla descrizione intuitiva dello scheduler.

Il programma `AvvioEJoin` rende visibile la distinzione fra oggetto e attività. Prima di `start`, lo stato è `NEW`. Una chiamata diretta a `lavoratore.run()` stampa `esegue: main`, perché il lavoro viene svolto dal thread principale; lo stato dell'oggetto `lavoratore` rimane `NEW`. Dopo `start`, il `Runnable` stampa `esegue: lavoratore`. `main` chiama `join` e solo dopo osserva `TERMINATED`. Il programma non si affida all'ordine casuale fra le due stampe: la prima avviene prima di `start`; la seconda deve terminare prima che `join` ritorni. Un thread non può essere avviato due volte con `start`: dopo la prima partenza, tentare di ripartire con lo stesso oggetto produce `IllegalThreadStateException`. Per un nuovo lavoro si crea un nuovo `Thread` o si affida l'attività a un esecutore.

```
NEW -- start() --> RUNNABLE -- run() termina --> TERMINATED
                     |
                     +-- attende monitor --> BLOCKED --> RUNNABLE
                     +-- attende evento ----> WAITING --> RUNNABLE
                     +-- attende tempo -----> TIMED_WAITING --> RUNNABLE
```

Schema testuale degli stati. Le frecce indicano casi tipici, non un calendario prevedibile. Uno stesso thread può tornare più volte a `RUNNABLE` prima di terminare.

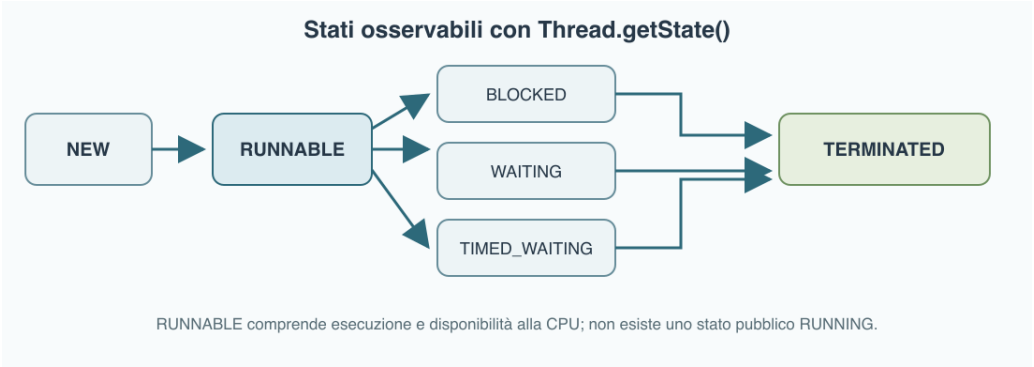


Figura 20.4 – Stati e transizioni principali di `Thread`.

Il metodo `join()` permette a un thread di attendere la fine di un altro. Se il thread corrente viene interrotto mentre attende, `join()` lancia `InterruptedException`. `sleep` sospende il thread corrente per almeno l'intervallo richiesto nelle condizioni previste dall'API, ma non assicura che riprenda in un istante esatto. `yield()` è un suggerimento allo scheduler, che può ignorarlo. Le priorità dei thread di piattaforma sono anch'esse suggerimenti dipendenti dall'implementazione e dal sistema operativo: non sono uno strumento per ottenere un ordine corretto di output. Se due thread stampano senza fine «priorità alta» e «priorità bassa», il risultato fotograferebbe una sola esecuzione e il programma potrebbe consumare risorse senza terminare; non dimostrerebbe una garanzia del linguaggio.

Che cosa possiamo e non possiamo dedurre dal tempo

Se `main` avvia un lavoratore e chiama `join(4000)`, l'attesa può terminare perché il lavoratore è finito, perché è trascorso il tempo indicato o perché `main` è stato interrotto, nel qual caso riceve un'eccezione. Dopo una `join` con limite temporale occorre dunque controllare se il lavoro è ancora vivo prima di dichiararlo completato. La differenza fra `join` senza limite e `join` con timeout riguarda la possibilità di tornare prima della conclusione del lavoratore, non l'ordine esatto delle stampe in un'esecuzione. Un timeout può aiutare a evitare un'attesa indefinita, ma non annulla automaticamente il lavoro rimasto attivo.

`sleep` è utile per simulare attese in un esempio, ma non è un meccanismo di coordinamento fra thread. Scrivere «attendo cento millisecondi e quindi il dato sarà pronto» lega la correttezza alla velocità di una macchina e al carico del sistema. Occorre invece attendere una condizione con strumenti appropriati: `join` per la conclusione di un thread, una coda bloccante per la disponibilità di un elemento, un `Future` per il risultato di un'attività. La specifica chiarisce anche che né `sleep` né `yield` introducono da soli le relazioni di sincronizzazione richieste per rendere visibile una scrittura condivisa.

La priorità ha un problema analogo. `setPriority` accetta valori nell'intervallo definito da `MIN_PRIORITY`, `NORM_PRIORITY` e `MAX_PRIORITY` per i thread ai quali si applica, ma il modo in cui influisce sulla pianificazione dipende dalla piattaforma. Un test che passa solo se il thread ad alta priorità stampa per primo è sbagliato: sta misurando un comportamento che l'API non garantisce. Possiamo mostrare la lettura e l'impostazione della priorità come proprietà del thread di piattaforma, senza dedurne una regola di precedenza semantica.

La memoria condivisa e l'operazione ++

Immaginiamo un contatore `int valore` su cui due thread eseguono `valore++` diecimila volte ciascuno. Il risultato atteso sul piano aritmetico è ventimila, ma l'operazione ++ sul campo non è un'azione indivisibile. Occorre leggere il valore, sommare uno e scrivere il nuovo valore. I due thread possono leggere lo stesso numero prima di scrivere, perdendo un incremento. Il problema non si risolve aggiungendo una pausa casuale o scegliendo una priorità diversa: l'ordine casuale resta possibile e il problema di visibilità fra thread resta aperto.

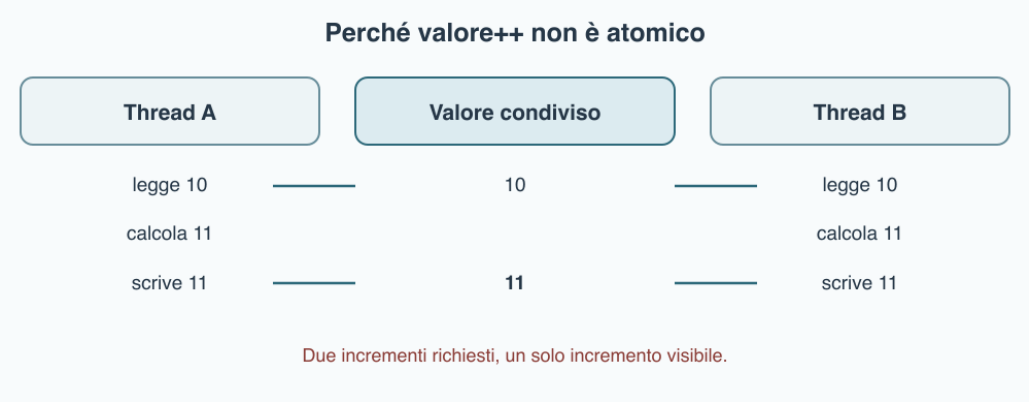


Figura 20.5 – Due letture dello stesso valore prima delle scritture producono un incremento perso.

Nel programma `ContatoreProtetto`, `incrementa()` e `leggi()` sono `synchronized`. I due metodi usano il monitor dell'istanza `contatore`. Soltanto un thread alla volta può eseguire una sezione protetta da quel medesimo monitor; l'uscita e la successiva acquisizione stabiliscono anche una relazione di visibilità secondo il Java Memory Model. Dopo aver avviato i due thread, `main` chiama `join()` su entrambi e poi legge il valore. L'output verificato è `20000`. Senza le condizioni di sincronizzazione non potremmo usare quella cifra come prova di correttezza, anche se qualche esecuzione fortunata la stampasse.

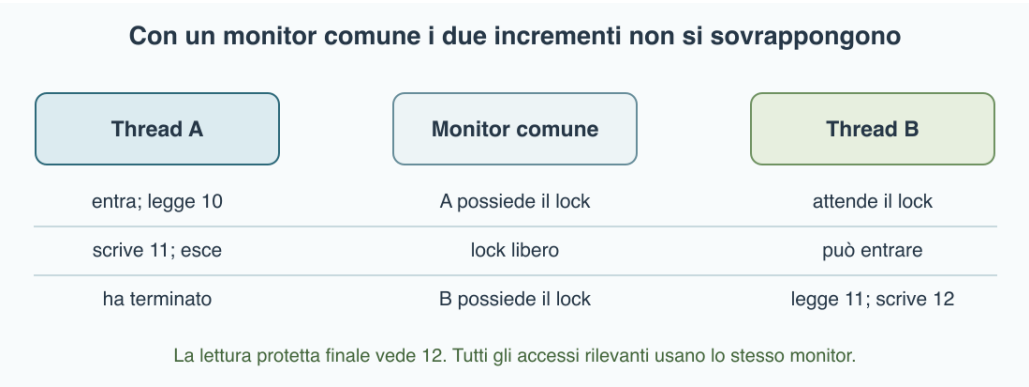


Figura 20.6 – La vista accoppiata alla figura precedente: con lo stesso monitor la seconda lettura avviene dopo la prima scrittura.

Un metodo di istanza `synchronized` usa il monitor di `this`. Un metodo `static synchronized` usa invece il monitor dell'oggetto `Class` che rappresenta la classe: non blocca automaticamente i monitor delle singole istanze. Un blocco `synchronized(lock)` permette di scegliere un oggetto dedicato e limitare la sezione critica alle sole istruzioni che ne hanno bisogno. Il lock deve essere condiviso dai thread che proteggono la stessa invariante. Due blocchi che usano oggetti diversi non si escludono, anche se nel sorgente entrambi contengono la parola `synchronized`.

```
private final Object lock = new Object();
private int valore;

void incrementa() {
    synchronized (lock) {
        valore++;
    }
}
```

Qui il campo `lock` è privato e finale: il codice esterno non può scegliere di acquisirlo e interferire con il protocollo interno. Il carattere `final` impedisce di sostituire il riferimento, non trasforma automaticamente in immutabili gli oggetti a cui un riferimento può puntare. Il valore protetto resta mutabile e deve essere letto con la stessa disciplina. Se aggiungessimo `int leggi() { return valore; }` senza sincronizzazione o altro meccanismo appropriato, avremmo rotto la regola di visibilità del contatore pur avendo protetto le scritture.

Quando due lock si aspettano a vicenda

Un *deadlock* può nascere se il thread A possiede il lock `primo` e aspetta `secondo`, mentre il thread B possiede `secondo` e aspetta `primo`. Nessuno può proseguire fino a rilasciare il lock che l'altro richiede. Il sistema non garantisce che il programma riconosca e risolva automaticamente questa situazione. Una regola semplice è acquisire più lock sempre nello stesso ordine documentato, oppure ridurre il numero di lock necessari. Il problema non è che i thread siano «lenti»: è che le condizioni logiche per avanzare sono diventate circolari.

```
Thread A: possiede primo  → attende secondo
Thread B: possiede secondo → attende primo
```

Questo schema descrive una situazione da evitare, non un esempio da eseguire nel percorso normale: due thread bloccati per sempre renderebbero la prova poco utile. Possiamo studiarla con una traccia o in un test isolato con un limite di tempo e diagnostica dei thread. In una revisione del codice si cercano coppie di sezioni che acquisiscono gli stessi lock in ordine opposto. Anche chiamare codice sconosciuto mentre si tiene un lock è rischioso, perché quel codice potrebbe cercare un altro lock o impiegare molto tempo.

> **Approfondimento – Tre proprietà diverse.** *Mutua esclusione* significa che due sezioni protette dallo stesso lock non si eseguono contemporaneamente. *Atomicità* significa che un'operazione appare indivisibile rispetto agli altri thread. *Visibilità* significa che una scrittura eseguita da un thread può essere osservata correttamente dall'altro. `volatile` fornisce particolari garanzie di visibilità e ordine, ma non rende atomico `valore++`. `synchronized` su un monitor comune può offrire sia esclusione sia le relazioni di memoria necessarie. Per un contatore indipendente esistono anche tipi atomici, come `AtomicInteger`; la scelta dipende dalle invarianti da proteggere.

La parola *invariante* indica una condizione che deve restare vera nei punti osservabili del programma. Se una coppia di campi `saldo` e `totale` deve essere aggiornata insieme, proteggere con un'operazione atomica un solo campo non basta. Il lock deve coprire il cambiamento coerente di entrambi. Questo è il motivo per cui la sincronizzazione non può essere aggiunta alla cieca: prima si identifica quale stato è condiviso e quale proprietà deve mantenere.

Un conto corrente letto da due persone

Un conto corrente fa vedere quanto una race condition possa essere più seria di un numero del contatore sbagliato. Supponiamo che il saldo sia 500 euro e che due richieste di prelievo da 400 euro arrivino quasi insieme. Ogni richiesta potrebbe leggere il saldo, verificarne la sufficienza e decidere di procedere. Se entrambe leggono 500 prima che una scriva il nuovo saldo, entrambe ritengono consentito il prelievo. A seconda della forma del codice si può registrare due volte un'uscita pur conservando un saldo incoerente, o perdere una delle modifiche. Non basta rendere atomica la sola sottrazione: il controllo «saldo sufficiente» e l'aggiornamento devono costituire un'unica operazione dal punto di vista degli altri thread.

```
synchronized (lock) {
    if (saldo < importo) {
        return false;
    }
    saldo -= importo;
    return true;
}
```

Questo frammento assume un `lock` condiviso e privato e un importo già validato come positivo. Il suo punto centrale è il confine della sezione critica, non una soluzione completa per un conto bancario: transazioni su database, persistenza e più processi richiedono garanzie ulteriori. Se il saldo è memorizzato su un server, un monitor Java di un singolo processo non coordina altre istanze dell'applicazione. Un libro di base deve mostrare il meccanismo senza estendere la promessa oltre il confine in cui vale.

> **Definizione – Sezione critica.** È la parte del programma che accede a stato condiviso e deve rispettare una regola comune. L'esclusione ha valore solo se tutti i percorsi che modificano o leggono quello stato in modo rilevante usano lo stesso protocollo. Una sezione critica troppo ampia aumenta l'attesa; una troppo stretta lascia scoperte parti dell'invariante. La sua dimensione si decide dalla proprietà da mantenere, non dal numero di righe che sembra comodo racchiudere fra parentesi.

Il ruolo preciso di `volatile`

Un campo `volatile` permette a una scrittura e a una lettura successive del medesimo campo di partecipare alla relazione *happens-before* descritta dal Java Memory Model. In un segnale semplice, un thread legge `fermati` finché un altro lo pone a `true`. Il programma `SegnaleVolatile` usa questa forma e stampa `fermato` dopo la richiesta. Non usa `sleep` per rendere visibile il cambiamento: il valore volatile è il mezzo di comunicazione. `Thread.onSpinWait()` è solo un suggerimento di ottimizzazione del ciclo di attesa attiva; non sostituisce la dichiarazione `volatile` e non è adatto quando il lavoro potrebbe attendere a lungo.



Figura 20.7 – La freccia rappresenta la relazione di memoria fra scrittura e lettura del medesimo campo `volatile`, senza imporre una particolare architettura delle cache.

Confrontiamo il ciclo con un booleano ordinario e poi con `volatile`. Il punto è che il compilatore e la CPU possono trattare le letture non sincronizzate in modi che sorprendono chi immagina una sola memoria immediatamente condivisa. Non promettiamo però che la versione difettosa si blocchi in ogni esecuzione o su ogni macchina. La prova empirica di una sola corsa non definisce la semantica: è la relazione di memoria stabilita dalla specifica a consentirci di ragionare sulla versione corretta.

`volatile` non protegge un'operazione composta. Con `volatile int valore`, la lettura di `valore`, il calcolo di `+1` e la scrittura sono ancora passi separati. Due thread possono leggere lo stesso vecchio valore e scrivere lo stesso nuovo valore. Se l'unica invariante è un contatore indipendente, `AtomicInteger` offre `incrementAndGet()`, che esprime un incremento atomico. Il programma avvia due thread, ne attende la conclusione e stampa `20000`. È una soluzione diversa dal monitor di `ContatoreProtetto`: non rende automaticamente atomiche altre modifiche che abbiamo dimenticato di includere.

> **Approfondimento – `happens-before` non è il cronometro.** La relazione indica quali effetti di memoria devono risultare visibili e in quale ordine logico per un programma correttamente sincronizzato. Non significa semplicemente che un'istruzione è stata eseguita pochi millisecondi prima di un'altra. `Thread.start` stabilisce una relazione dalle azioni precedenti all'avvio alle azioni del nuovo thread; la conclusione osservata con `join` stabilisce una relazione dalle azioni del lavoratore a quelle successive del chiamante. Questo spiega perché `main`, dopo `join`, può leggere il risultato finale di un lavoro svolto dal thread terminato. Per accessi condivisi mentre entrambi lavorano, occorre comunque un protocollo adatto.

Interruzione: una richiesta cooperativa

`interrupt()` segnala a un thread che dovrebbe interrompere il lavoro o uscire da un'attesa. Non equivale a terminare forzatamente il thread in un punto arbitrario. Se il thread è in una chiamata bloccante interrompibile, come `sleep`, `join` o `BlockingQueue.take`, può ricevere `InterruptedException`. In quel momento il segnale di interruzione viene normalmente cancellato dalla chiamata che lancia l'eccezione. Se il livello corrente non sa concludere l'operazione e deve restituire il controllo, può ripristinare il segnale con `Thread.currentThread().interrupt()` e terminare il proprio ciclo, come fanno gli esempi di questo capitolo.

Consideriamo un blocco `catch (InterruptedException e) {}` vuoto. Un blocco vuoto può lasciare il thread al lavoro dopo che un altro componente ha chiesto la chiusura. In un piccolo programma dimostrativo potrebbe passare inosservato; in un servizio con risorse reali rende la terminazione incerta. Se il metodo può dichiarare `throws InterruptedException`, spesso è meglio lasciare propagare l'eccezione. Se non può, documentiamo come la richiesta viene rispettata. Non esiste una risposta unica per ogni applicazione, ma ignorare il segnale per distrazione non è una strategia.

> **Nota – Interruzione e stato.** `isInterrupted()` legge il segnale senza azzerarlo. `Thread.interrupted()` è statico e consulta il thread corrente azzerandone il segnale. La differenza conta nei cicli che aspettano una richiesta di arresto. Non usiamo i due metodi come sinonimi; prima decidiamo se il codice deve soltanto osservare l'interruzione o anche consumarne l'indicazione.

Produttore e consumatore, dal difetto alla soluzione

Il caso produttore-consumatore merita un percorso graduale, perché ogni soluzione fa emergere un nuovo requisito. Un produttore genera valori; un consumatore li usa. Se il consumatore legge un campo prima che il produttore inserisca il prossimo valore, può ottenere un dato non pronto o riutilizzare il precedente. Una lista condivisa senza protocollo non risolve la questione. Anche con metodi `synchronized`, un controllo `if (lista.isEmpty()) wait()` è insufficiente: dopo il risveglio la condizione va ricontrollata in un `while`, perché un altro consumatore può aver preso l'elemento oppure il risveglio può non corrispondere alla disponibilità attesa.

`wait()` richiede il monitor dell'oggetto su cui è chiamato; rilascia quel monitor durante l'attesa e lo riacquisisce prima di proseguire. `notifyAll()` risveglia i thread in attesa sullo stesso monitor, ma non consegna loro immediatamente la risorsa: devono competere per il monitor e verificare di nuovo la condizione. Con due consumatori può emergere un errore: occorre osservare che la condizione di attesa non è un permesso permanente a rimuovere un elemento. Un'implementazione manuale corretta richiede un protocollo completo per disponibilità, chiusura e interruzione.

La libreria standard fornisce `BlockingQueue<E>` per questo caso. Una coda limitata può bloccare `put` quando è piena e `take` quando è vuota. Nel programma `ProduttoreConsumatore` usiamo `ArrayBlockingQueue<Integer>` con capacità due. Il produttore inserisce i numeri da zero a quattro; il consumatore ne legge uno alla volta. Dopo i dati, il produttore inserisce la sentinella `-1`, che indica la fine. La sentinella è sicura qui perché i dati validi sono solo da zero a quattro. In un sistema reale occorre verificare che un valore speciale non possa confondersi con un dato e che la terminazione funzioni anche quando il produttore fallisce.

Costruire il protocollo a mano per capirlo

Prima di usare la coda pronta, costruiamo un deposito a un solo posto e correggiamo passo dopo passo la sua sincronizzazione. Nel programma `DepositoConAttesa` il campo `prodotto` vale `null` quando il posto è vuoto. `inserisci` attende con `while (prodotto != null)` fino a quando il consumatore ha prelevato il valore precedente; `preleva` attende con `while (prodotto == null)` fino a quando arriva un valore. Entrambi i metodi sono `synchronized` sull'istanza del deposito. Cambiata la condizione, chiamano `notifyAll()` perché i thread che aspettano possano ricontrollarla.

La forma `while` non è una preferenza di stile. La specifica permette risvegli senza una corrispondente `notify`, e un altro thread può cambiare la condizione prima che il thread risvegliato riacquisti il monitor. Con `if`, il controllo avviene una sola volta e il metodo può proseguire quando il deposito è ancora vuoto o ancora pieno. Con due consumatori, questo difetto può manifestarsi anche quando una prova con un solo consumatore sembra funzionare. Nel nostro primo programma manuale c'è un solo consumatore, così il lettore può isolare il protocollo di un posto prima di ragionare sulla competizione fra molti consumatori.

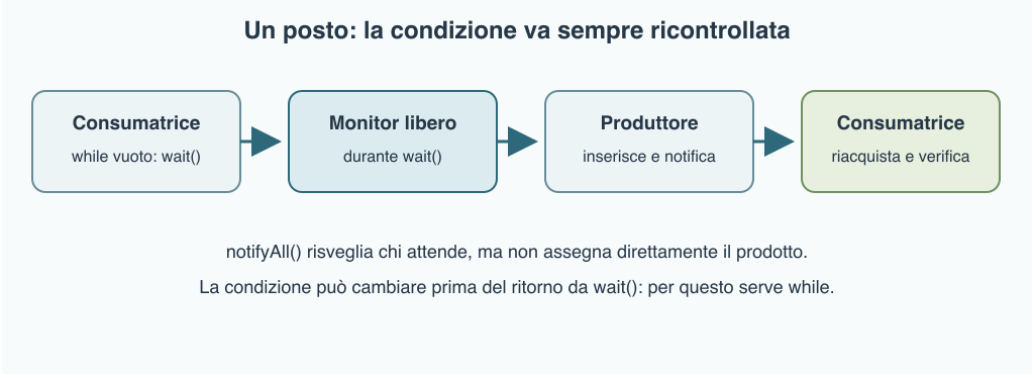


Figura 20.8 – `wait` rilascia il monitor; dopo la notifica il thread riacquisisce il monitor e ricontrolla la condizione.

```
public synchronized int preleva() throws InterruptedException {
    while (prodotto == null) {
        wait();
    }
    int numero = prodotto;
    prodotto = null;
    notifyAll();
    return numero;
}
```

La chiamata a `wait()` appartiene all'oggetto deposito e avviene mentre il thread ne possiede il monitor. Durante l'attesa quel monitor viene rilasciato: altrimenti il produttore non potrebbe entrare in `inserisci` e la condizione non cambierebbe mai. Prima del ritorno da `wait`, il thread riacquisisce il monitor; solo allora controlla di nuovo il `while`. `notifyAll` non trasferisce direttamente il valore al consumatore e non fa uscire subito dal metodo chi lo chiama. È una notifica a chi attende sul medesimo oggetto.

Il valore `-1` conclude il flusso, come nella versione con `BlockingQueue`. La stampa dei valori da zero a quattro è deterministica perché un solo produttore li inserisce nell'ordine previsto e un solo consumatore li preleva nell'ordine del deposito. L'esempio mostra il funzionamento normale, ma non ancora ogni caso di produzione. Se il produttore viene interrotto prima di inserire la sentinella, il consumatore potrebbe attendere per sempre. Un servizio robusto richiede una procedura di chiusura anche per gli errori; una coda bloccante semplifica trasferimento e capacità, ma da sola non definisce la politica di chiusura dell'applicazione.

Le quattro forme di inserimento e prelievo

L'interfaccia `BlockingQueue` offre più modi di reagire a pieno e vuoto. `add` può lanciare un'eccezione se non c'è spazio; `offer` restituisce `false`; `put` aspetta. La forma di `offer` con timeout aspetta al più il tempo dichiarato e restituisce un booleano. Per leggere, `remove` può lanciare un'eccezione sul vuoto, `poll` restituisce `null`, `take` aspetta; esiste anche `poll` con timeout. `element` e `peek` osservano l'elemento in testa senza rimuoverlo, con differenze analoghe sul caso vuoto. Il chiamante sceglie in base al contratto: «non c'è spazio» può essere un errore, una risposta ordinaria o una condizione da attendere.

Operazione	Fallimento immediato con eccezione	Risposta immediata speciale	Attesa	Attesa con timeout
Inserire	`add(e)`	`offer(e)` → `false`	`put(e)`	`offer(e, tempo, unità)`
Prelevare	`remove()`	`poll()` → `null`	`take()`	`poll(tempo, unità)`
Osservare la testa	`element()`	`peek()` → `null`	non prevista	non prevista

Il `null` di `poll` non può essere scambiato con un elemento valido, perché `BlockingQueue` non permette di inserire `null`. Questo dettaglio fa parte del contratto e rende interpretabile la risposta speciale. Una coda limitata come `ArrayBlockingQueue` applica una capacità definita; una coda senza limite fisso può accettare elementi finché ci sono risorse disponibili, quindi «senza limite» non significa memoria infinita. La capacità è una decisione di progetto: limita la pressione del produttore sul consumatore e rende osservabile un rallentamento invece di far crescere indefinitamente la memoria occupata.

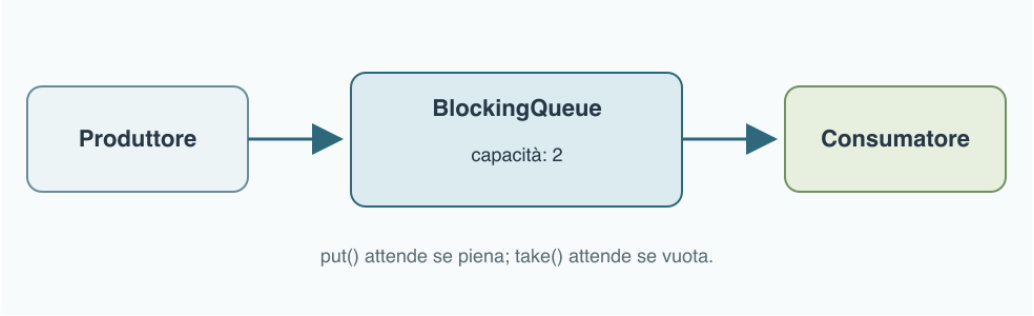


Figura 20.9 – La coda trasferisce i valori e coordina l'attesa quando è piena o vuota.

L'output contiene sempre i cinque valori consumati in ordine, perché una singola coda FIFO alimenta un solo consumatore, ma non possiamo dedurre da quelle righe quando esattamente i due thread hanno ottenuto CPU. Con più consumatori, una stampa che alterna produttore e lavoratori rappresenta una possibile intercalazione, non l'unico «output atteso». Con più consumatori il reparto dei valori fra i nomi dei thread non è deterministico; si verificano invece proprietà come «ogni valore valido è consumato una volta» e «nessun consumatore resta bloccato dopo la chiusura». Se ci fossero due consumatori, una sola sentinella non basterebbe a concluderli entrambi.

Thread di piattaforma e thread virtuali

Fin qui abbiamo visto due forme di avvio: il contatore crea thread di piattaforma con `Thread.ofPlatform()`, mentre il produttore-consumatore usa `Thread.ofVirtual()`. Entrambi i programmi attendono la conclusione con `join`. Le chiamate a `BlockingQueue.put` e `take` mantengono lo stesso significato; per capire perché scegliere una forma anziché l'altra dobbiamo guardare al lavoro che il thread svolge e al tempo che trascorre in attesa.

> **Approfondimento – Attività e durata.** Se un server riceve molte richieste indipendenti che attendono servizi esterni, un thread virtuale per richiesta può essere una scelta leggibile. Se invece dobbiamo moltiplicare grandi matrici, il lavoro utile è limitato dai core disponibili e dalla memoria; creare un numero enorme di thread non produce core aggiuntivi. Per scegliere bene bisogna misurare il tipo di attesa, la durata delle attività e la quantità di stato condiviso.

Perché è servito un altro tipo di thread

Immagina un servizio che riceve richieste di prestito. Per ognuna controlla il catalogo, interroga un archivio remoto e infine prepara una risposta. Il codice più naturale segue quest'ordine: chiama il catalogo, aspetta la risposta, chiama l'archivio, aspetta ancora e compone il risultato. Durante le due attese non sta calcolando: la CPU potrebbe occuparsi di un'altra richiesta. Con un thread di piattaforma per richiesta, però, ogni attesa trattiene anche un thread del sistema operativo per tutta la durata della richiesta. Quando le richieste simultanee diventano numerose, quel costo limita la capacità del servizio prima ancora che la CPU sia piena.

Un pool di thread riutilizza i thread e risparmia il costo di crearli ripetutamente, ma non risolve il vincolo: se il pool contiene cento lavoratori e cento richieste sono ferme ad aspettare risposte esterne, la richiesta successiva resta in coda. Un'altra strada è spezzare ogni richiesta in callback o `CompletableFuture`, così i lavoratori sono liberi mentre l'I/O è in corso. Il prezzo è che il percorso logico di una richiesta si distribuisce fra più funzioni e diventa più difficile seguire valori, errori e cancellazione. I thread virtuali sono stati introdotti per conservare il modello leggibile «un thread per richiesta» anche quando molte richieste aspettano contemporaneamente. Sono diventati una funzionalità stabile in Java 21; in questa edizione li usiamo con Java 25. La motivazione del progetto e la guida Java 25 descrivono questo obiettivo in termini di capacità di servire più lavoro concorrente, non di maggiore velocità del singolo calcolo.

> **Per capire – Concorrenza, throughput e latenza.** La *concorrenza* è quante richieste sono in corso nello stesso intervallo. Il *throughput* è quante richieste finiscono in un secondo. La *latenza* è quanto aspetta una singola richiesta prima del risultato. Se ogni richiesta dura in media 50 millisecondi e il servizio completa 200 richieste al secondo, in media circa 10 richieste sono in corso: `200 × 0,050 = 10`. Per completarne 2.000 al secondo alla stessa durata, ne servono circa 100 contemporanee. È una relazione fra medie, non una promessa di prestazioni. Ridurre il costo dei thread che aspettano può permettere più richieste in corso; non fa diventare più breve la risposta del database.

Che cosa esegue il codice mentre il thread aspetta

Anche un thread virtuale esegue istruzioni su un thread del sistema operativo: non esiste una CPU «virtuale» aggiuntiva. La JVM assegna temporaneamente il thread virtuale a un thread di piattaforma, chiamato *carrier*. Finché il codice Java calcola, il carrier lo esegue. Quando il thread virtuale incontra un'operazione bloccante supportata dal runtime, per esempio l'attesa di dati da un socket, la JVM può sospenderlo e liberare il carrier. Il carrier esegue allora un altro thread virtuale. Quando i dati arrivano, il thread sospeso diventa nuovamente eseguibile e può riprendere su un carrier, anche diverso dal precedente. Il programma continua dopo la chiamata bloccante come se avesse semplicemente aspettato.

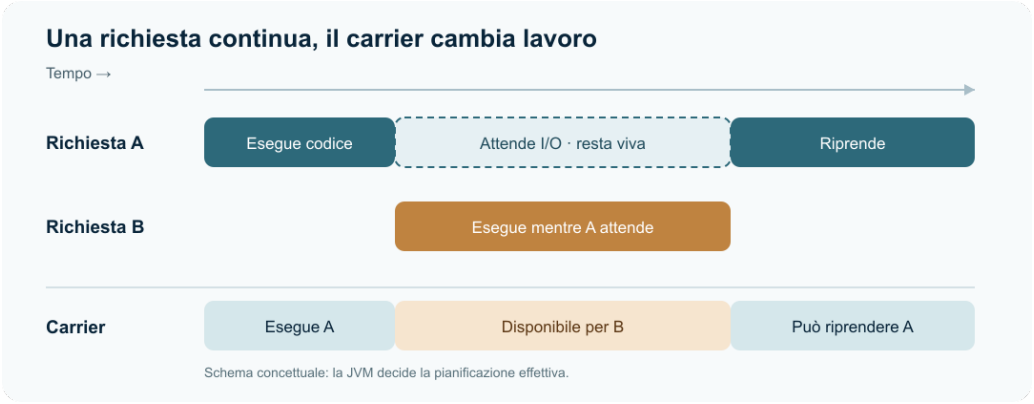


Figura 20.10 – Un thread virtuale rappresenta la richiesta per tutta la sua durata; il carrier è occupato soltanto mentre ne esegue il codice. Quando la richiesta A attende l'I/O, la JVM può usare lo stesso carrier per la richiesta B. La figura rappresenta il meccanismo, non una sequenza temporale garantita dal programma.

> **Definizione – Sospensione e carrier.** *Sospendere* un thread virtuale significa conservare il punto da cui riprenderà, inclusa la sua catena di chiamate. Il *carrier* è il thread di piattaforma che lo sta eseguendo in quel momento. Il carrier non appartiene per sempre al thread virtuale: dopo un'attesa può eseguirne un altro. `Thread.currentThread()` nel codice della richiesta continua a identificare il thread virtuale, non il carrier.

Questa distinzione spiega anche che cosa non cambia. `synchronized`, `volatile`, le eccezioni, l'interruzione e le regole di visibilità della memoria restano regole dei thread Java. Un incremento non protetto può andare perso anche se entrambi i partecipanti sono virtuali. Inoltre, non ogni chiamata bloccante libera sempre il carrier: in Java 25 il codice nativo o una foreign function possono trattenere l'associazione, situazione chiamata *pinning*. Un'attesa lunga mentre il thread è bloccato così può ridurre la capacità del servizio. Non è un motivo per evitare automaticamente i thread virtuali; è un motivo per osservare il carico e le operazioni effettive. La guida Java 25 distingue questo caso dal normale uso dei blocchi `synchronized`.

Una richiesta, un thread virtuale, un confine di vita

Per una piccola prova si può creare il thread direttamente: `Thread.ofVirtual().name("prestito-1").start(lavoro)` lo avvia e restituisce un `Thread` su cui chiamare `join()`. In un servizio che riceve molte richieste, `Executors.newVirtualThreadPerTaskExecutor()` crea invece un thread virtuale nuovo per ogni attività inviata. Non è un pool di thread virtuali da riempire e riutilizzare. La separazione è utile perché l'applicazione può descrivere il lavoro con un `Callable` e poi gestire il risultato tramite `Future`, come nel programma `EsecutoreVirtuale` che segue. Il `try` con risorse chiude l'esecutore dopo il lavoro e rende esplicito quando le attività devono essere terminate.

Il vantaggio atteso emerge se molte attività indipendenti aspettano spesso I/O. Non emerge nel piccolo calcolo di 22 più 33 del nostro esempio: quel programma serve a leggere l'API, non a misurare una velocità. La risorsa esterna ha comunque i

propri limiti. Se il catalogo remoto accetta venti richieste contemporanee, creare diecimila thread virtuali non gli conferisce una capacità maggiore. Un `Semaphore` con venti permessi può limitare gli accessi al catalogo, mentre i thread virtuali in attesa di un permesso non occupano inutilmente un carrier. Il semaforo limita l'accesso alla risorsa scarsa; il numero di thread descrive quante attività esistono. Sono due decisioni diverse.

> **Quando servono davvero.** Scegli un thread virtuale per un'attività indipendente che passa una parte rilevante della sua vita ad aspettare I/O: una richiesta HTTP, una query o una lettura di rete sono esempi tipici. Se il lavoro calcola continuamente, il limite resta il numero di core. Se condividi stato modificabile, devi ancora coordinare gli accessi. Se avvii un'attività che deve finire prima della risposta o della chiusura del programma, devi ancora attenderla e gestire il fallimento. Nessuno di questi compiti viene svolto automaticamente dalla parola *virtuale*.

Due limiti che restano: stato condiviso e risorse esterne

Il meccanismo del carrier spiega il costo di molte attese, ma non modifica il contratto del Java Memory Model. Gli esempi di `synchronized`, `volatile` e `BlockingQueue` mantengono la stessa semantica se passiamo da `Thread.ofPlatform()` a `Thread.ofVirtual()`. Un aggiornamento non protetto resta una race condition; il programma deve scegliere il protocollo di accesso allo stato condiviso.

Anche l'attesa economica non rende illimitata la risorsa attesa. Ogni attività conserva dati, riferimenti e lavoro da completare; un database o un servizio remoto può accettare meno richieste di quante la JVM possa avviare. Se cento mila attività chiedono una connessione a un database con poche connessioni disponibili, il problema diventa la pressione sul servizio e sulle code. Il semaforo dell'esempio precedente limita gli accessi al catalogo, mentre una coda limitata può impedire di accumulare richieste senza fine. Il thread virtuale rende conveniente aspettare; non crea capacità nel database.

Un ciclo che occupa continuamente la CPU resta limitato dai core e dalla memoria. Possiamo allora distinguere tre domande prima di scegliere la forma concorrente: l'attività aspetta soprattutto I/O, calcola senza pause oppure contende uno stato condiviso? I thread virtuali rispondono soprattutto alla prima; la seconda richiede una misura del carico e la terza un protocollo corretto. Questa distinzione prepara il confronto con gli stream paralleli.

> **Collegamento – Stream paralleli e thread.** Una stream pipeline può essere richiesta in forma parallela, ma il codice al suo interno deve comunque rispettare le regole dello stato condiviso. Un accumulatore mutabile catturato da una lambda può avere una race condition tanto quanto un campo aggiornato da due `Thread`. La parallelizzazione di una pipeline non risolve la semantica dell'operazione né assicura velocità maggiore. Le attività con molte attese di I/O e le trasformazioni dati CPU hanno profili diversi; per questo non sostituiamo automaticamente un esecutore di thread virtuali con `parallelStream()`.

Servizi di esecuzione e confini del lavoro

Gestire a mano un oggetto `Thread` per ogni operazione è utile per capire il modello, ma applicazioni più grandi spesso separano la *descrizione del lavoro* dalla *politica di esecuzione*. Un `ExecutorService` accetta attività; un `Future<T>` può rappresentare un risultato da attendere, distinguendo il valore, un'eccezione e una cancellazione. Un pool di thread di piattaforma può limitare la concorrenza; un esecutore per thread virtuali può offrire un thread virtuale a ogni attività. La chiusura dell'esecutore è parte del contratto: non lasciamo risorse aperte dopo l'uso.

Il programma stabile che segue mostra il confine di vita con `Future` e la chiusura dell'esecutore. Subito dopo esamineremo la domanda che questa coppia lascia al chiamante: come trattare più lavori figli come una sola richiesta. L'API `StructuredTaskScope` di Java 25 affronta quel problema, ma è ancora preview; il suo stato viene chiarito nell'approfondimento dedicato.

Il programma `EsecutoreVirtuale` usa `Executors.newVirtualThreadPerTaskExecutor()` e passa due `Callable<Integer>` attraverso `submit`. Il tipo `Callable`, diversamente da `Runnable`, produce un risultato e può lanciare un'eccezione. Ogni chiamata restituisce un `Future<Integer>`, sul quale `get()` attende il completamento e restituisce il valore oppure segnala il fallimento. I due lavori calcolano 22 e 33; il programma stampa 55. L'esecutore viene chiuso dal costrutto `try` con risorse, che rende visibile la durata del servizio di esecuzione. L'esempio non mostra un beneficio prestazionale, perché i due calcoli sono minuscoli; mostra come rappresentare attività con risultato e un confine di vita esplicito.

Se un'attività fallisce, `Future.get()` può lanciare `ExecutionException`, la cui causa è l'eccezione prodotta dal lavoro. Se chi aspetta viene interrotto, riceve `InterruptedException`. Queste risposte sono diverse da «il risultato è zero» e non vanno cancellate con un `catch` vuoto. Anche `Future.cancel` richiede una scelta: può richiedere un'interruzione del lavoro, ma il lavoro deve cooperare e la cancellazione non annulla retroattivamente gli effetti già avvenuti. Un esercizio serio sugli esecutori deve quindi includere il percorso di errore e di chiusura, oltre al risultato felice.

Uno sguardo alle attività figlie in Java 25

Una richiesta applicativa può aver bisogno di due risposte indipendenti prima di costruire una pagina: per esempio il profilo di un libro e la disponibilità del magazzino. Con due `Future` sappiamo avviare i lavori e aspettarne i valori, ma dobbiamo ancora definire insieme cosa fare se uno fallisce o se il chiamante viene interrotto. Il problema non è soltanto «eseguire in parallelo»: è far coincidere la durata dei lavori figli con la durata della richiesta che li ha creati. Se la richiesta non serve più, lasciare in giro un lavoro che continua a occupare risorse può essere uno spreco o un errore.

La *concorrenza strutturata* affronta questa relazione tra attività padre e figlie. In Java 25 `StructuredTaskScope` è ancora una API **preview**; i dettagli e le firme possono cambiare. Per questo non la usiamo nei programmi principali né la presentiamo come prerequisito per risolvere il produttore-consumatore. È utile però conoscere l'idea: un ambito lessicale raccoglie i lavori figli e fa sì che il chiamante ne gestisca conclusione e fallimento prima di abbandonare quell'ambito. Un esempio eseguibile basato su questa API richiederebbe `--enable-preview` sia a `javac` sia a `java`, oltre alle altre opzioni di versione richieste dalla compilazione.

> **Perché lo stato conta.** Un'API preview fa parte di un JDK rilasciato, ma viene offerta per raccogliere esperienza prima di una forma definitiva. Una API incubator, come la Vector API del capitolo 23, vive in un modulo sperimentale. Il lettore può provare entrambe, ma una libreria destinata a molti ambienti non dovrebbe nascondere queste dipendenze dietro un esempio che sembra stabile. Nel corpo principale del capitolo 20 usiamo soltanto API stabili del JDK 25.

`ScopedValue` è invece definitiva in Java 25. Descrive un valore associato a un ambito di esecuzione, utile per passare informazioni di contesto immutabili lungo una catena di chiamate senza aggiungere lo stesso parametro a ogni metodo. Non è una variabile globale da modificare liberamente: il suo valore è legato a un ambito e segue regole precise di visibilità. Se una richiesta porta un identificatore per il log, un valore con ambito può essere più adatto di un campo statico condiviso da tutte le richieste. Non serve però a coordinare l'incremento del contatore o la disponibilità di un prodotto; quei problemi riguardano modifiche condivise e attese.

Il programma `ContestoRichiesta.java` rende osservabile la durata del legame. `ID_RICHIESTA` è la chiave condivisa dal codice, non il valore della richiesta: `ScopedValue.newInstance()` la crea inizialmente senza valore associato. `where(...).run(...)` associa `P-17` soltanto durante quella chiamata. `preparaRisposta` invoca un altro metodo che legge il valore con `get()` pur senza riceverlo come parametro. Dopo il ritorno da `run`, `isBound()` torna a `false`. Usare `get()` fuori dall'ambito lancerebbe `NoSuchElementException`; l'esempio controlla invece `isBound()` per mostrare il confine senza trasformare l'errore in un risultato normale.

```
public class ContestoRichiesta {
    private static final ScopedValue<String> ID_RICHIESTA = ScopedValue.newInstance();

    private static void registra(String azione) {
        System.out.println(ID_RICHIESTA.get() + ": " + azione);
    }

    private static void preparaRisposta() {
        registra("catalogo consultato");
    }

    public static void main(String[] args) {
        System.out.println("Prima: " + ID_RICHIESTA.isBound());
        ScopedValue.where(ID_RICHIESTA, "P-17").run(ContestoRichiesta::preparaRisposta);
        System.out.println("Dopo: " + ID_RICHIESTA.isBound());
    }

}
```

Compila con `javac --release 25 -Xlint:all -d build ContestoRichiesta.java` ed esegui `java -cp build ContestoRichiesta`: l'output è `Prima: false`, `P-17: catalogo consultato`, `Dopo: false`. Cambia `P-17` in `P-18` e prevedi quale riga varia. Poi chiediti perché sostituire il valore con un contatore mutabile richiederebbe regole ulteriori: `ScopedValue` trasmette un contesto, ma non protegge le modifiche a un oggetto condiviso. Il programma rimane sul solo percorso stabile di Java 25; la trasmissione ai thread figli creati con `StructuredTaskScope` richiederebbe invece la API preview e viene lasciata all'approfondimento precedente.

Questi due strumenti illuminano una distinzione importante. Il contesto di una richiesta può essere *trasmesso* a funzioni figlie senza essere aggiornato in concorrenza; lo stato di una coda o di un saldo viene invece *modificato* da più attività e richiede un protocollo. Se confondiamo i due casi, rischiamo di usare un lock per ogni dato di contesto oppure di usare un valore contestuale dove servirebbe una transazione. Le novità del linguaggio e della libreria non cancellano le domande fondamentali: chi possiede il dato, chi può cambiarlo e quando l'operazione è conclusa.

Dare un nome al lavoro e capire chi mantiene viva la JVM

I nomi dei thread non determinano il loro comportamento, ma aiutano a leggere un log o una traccia. Nell'esempio `AvvioEJoin`, il nome `lavoratore` rende visibile il passaggio dalla chiamata diretta di `run()` al thread realmente avviato. In un server conviene usare nomi che descrivano il ruolo o il tipo di attività, non numeri senza contesto. Il nome può essere duplicato e può cambiare: è un aiuto diagnostico, non un identificatore globale di identità. Per sapere quale thread sta eseguendo una riga si può chiamare `Thread.currentThread()`; stampare questo riferimento in un esercizio è più utile che supporre da quale thread arrivi una chiamata.

Un thread di piattaforma può essere *daemon* o *non daemon*. La JVM può terminare quando non restano thread non daemon, senza aspettare che i daemon finiscano il proprio lavoro. Per questo un daemon è adatto a certe attività di supporto che non devono mantenere vivo il processo; non è il posto giusto per salvare l'ultimo dato

essenziale alla chiusura se non esiste un protocollo esplicito. `setDaemon(true)` va chiamato prima di `start()` sul thread a cui si applica. Un thread virtuale è sempre daemon secondo il contratto dell'API: questo rende ancora più importante attendere esplicitamente le attività che contano, per esempio tramite `join`, `Future.get()` o la chiusura dell'esecutore. La fine del metodo `main` non deve essere usata come sostituto della gestione della durata delle attività.

> **Approfondimento – Durata del thread e durata dell'operazione.** Un thread può esistere più a lungo dell'oggetto che lo ha creato, se nessuno lo attende o lo ferma. Un'operazione applicativa, invece, ha un momento in cui il chiamante deve poter dire se è conclusa, fallita o ancora in corso. Collegare le due durate è responsabilità del programma. La forma `try` con un `ExecutorService` è un modo per rendere visibile quel confine; creare thread anonimi senza conservarne i riferimenti può renderlo opaco.

La distinzione fra `Runnable` e sottoclasse di `Thread` merita una spiegazione più ampia di una preferenza di stile. `Runnable` descrive un compito mediante il metodo `run()`. La stessa istanza di `Runnable` può essere passata a un `Thread`, a un esecutore o richiamata direttamente in un test, pur con semantiche di esecuzione diverse. Una sottoclasse di `Thread` combina in un unico oggetto compito e politica di esecuzione; può essere utile in casi particolari, ma occupa anche l'unica classe da cui Java permette di ereditare. Una lambda è spesso la forma più breve di un `Runnable`, purché il corpo resti leggibile e la cattura di variabili non nasconda stato condiviso pericoloso.

```
Runnable lavoro = () -> System.out.println(Thread.currentThread().getName());
Thread thread = new Thread(lavoro, "lettore");
thread.start();
thread.join();
```

Il frammento presuppone un metodo che dichiari o gestisca `InterruptedException` per `join`. Il `Runnable` non riceve automaticamente il nome `lettore`: il nome appartiene al `Thread` che lo esegue. Se chiamassimo `lavoro.run()` direttamente, il nome sarebbe quello del chiamante. È la stessa distinzione osservata nel programma completo, ora espressa nella forma che useremo più spesso.

`Thread` offre diversi costruttori che ricevono un `Runnable`. Tutti creano un **thread di piattaforma non ancora avviato**: dopo la costruzione occorre chiamare `start()`. Il parametro `target` indica il lavoro; il nome aiuta la diagnosi, mentre il gruppo e la dimensione dello stack rispondono a esigenze più specifiche.

Costruttore	Che cosa aggiunge
`Thread(Runnable target)`	Un lavoro con un nome generato dalla piattaforma.
`Thread(Runnable target, String name)`	Un nome scelto dal programma.
`Thread(ThreadGroup group, Runnable target)`	Un gruppo di thread esplicito.

Costruttore	Che cosa aggiunge
`Thread(ThreadGroup group, Runnable target, String name)`	Gruppo e nome espliciti.
`Thread(ThreadGroup group, Runnable target, String name, long stackSize)`	Anche una richiesta approssimativa per la dimensione dello stack.

Il gruppo non è necessario per i piccoli esempi di questo capitolo: serve soprattutto quando un'applicazione esistente organizza i thread di piattaforma in gruppi. La dimensione `stackSize` è espressa in byte, ma la JVM e il sistema operativo possono interpretarla in modo diverso o non applicarla; non va usata come garanzia di una certa profondità di ricorsione. In Java 25 si possono anche usare i builder `Thread.ofPlatform()` e `Thread.ofVirtual()`: rendono esplicito il tipo di thread e permettono di comporre alcune opzioni prima dell'avvio. La scelta del costruttore non cambia la semantica di `Runnable.run()` né rende sicuro l'accesso a dati condivisi.

La sincronizzazione è un accordo fra tutti i partecipanti

Per capire perché il contatore protetto funziona, seguiamo una possibile intercalazione. Il primo thread entra in `incrementa`, acquisisce il monitor di `contatore`, legge 10, calcola 11 e lo scrive. Il secondo thread può essere pronto nello stesso momento, ma non può entrare in quello stesso metodo protetto sullo stesso oggetto prima che il primo rilasci il monitor. Quando entra, legge il nuovo valore secondo le garanzie di memoria del lock. Non è necessario indovinare quale thread sarà il primo: entrambe le sequenze ammesse producono due incrementi. La correttezza nasce dal fatto che ogni incremento è racchiuso nello stesso protocollo.

Supponiamo invece che un'altra classe riceva il riferimento al contatore e modifichi direttamente un campo pubblico `valore`. Anche se `incrementa` è `synchronized`, quella scrittura non partecipa al protocollo e può interferire. L'incapsulamento del campo è quindi parte della soluzione: rendere `valore` privato impedisce al codice esterno di aggirare accidentalmente il lock. Un metodo `leggi` protetto dallo stesso monitor rende la regola facile da applicare. Quando lo stato condiviso è distribuito fra più oggetti, serve decidere quale componente possiede l'invariante; spargere `synchronized` su oggetti diversi non crea per magia un unico confine.

L'uso di un blocco sincronizzato richiede inoltre di scegliere il lock senza esporlo. `synchronized(this)` può essere corretto quando l'intero oggetto ha un protocollo noto, ma consente a un chiamante che possiede il riferimento di acquisire lo stesso monitor e trattenere l'oggetto. Un lock privato rende il protocollo interno meno soggetto a interferenze esterne. La scelta dipende dal contratto della classe: un componente che vuole permettere esplicitamente ai chiamanti di sincronizzarsi su di esso ha esigenze diverse. Nella maggior parte delle classi applicative, un lock privato riduce le sorprese.

Il monitor è *rientrante*: un thread che lo possiede può entrare di nuovo in un metodo sincronizzato sullo stesso oggetto, per esempio quando `incrementa()` chiama un altro metodo `synchronized` della stessa istanza. Non va confuso con l'assenza di deadlock: la rientranza riguarda lo stesso thread e lo stesso monitor. Il

deadlock descritto in precedenza coinvolge più thread e più risorse, con un ciclo di attese. Una classe può quindi essere corretta rispetto alla rientranza e ancora pericolosa se prende lock differenti in ordini diversi.

> **Nota di lettura – `synchronized` e output.** Se due thread eseguono metodi sincronizzati dello stesso oggetto e ciascuno stampa una riga, il lock può serializzare quelle sezioni, ma non decide quale thread entrerà per primo. Un output A poi B e un output B poi A possono essere entrambi corretti. Se l'ordine è requisito del problema, bisogna codificarlo con una condizione o con un canale di passaggio, non sperare che il lock favorisca sempre lo stesso lavoratore.

Cosa accade davvero quando un thread aspetta

Il metodo `Object.wait()` appartiene all'oggetto usato come monitor e richiede che il thread lo possieda. La chiamata inserisce il thread nell'insieme di attesa dell'oggetto, rilascia il monitor e sospende l'avanzamento finché un evento previsto dalla specifica lo risveglia. Dopo il risveglio il thread deve riacquisire il monitor prima di ritornare dalla chiamata. Per questo un produttore che chiama `notifyAll()` mentre possiede ancora il monitor non consegna immediatamente il controllo al consumatore: prima deve uscire dalla sezione sincronizzata.

`notify()` seleziona un thread in attesa senza promettere quale; `notifyAll()` rende tutti i thread in attesa candidati a proseguire, uno alla volta per l'acquisizione del monitor. Nel deposito a un solo posto, produttore e consumatore aspettano condizioni opposte sullo stesso oggetto. Se si usasse `notify()` in una variante con più produttori e più consumatori, si potrebbe risvegliare un thread che trova ancora falsa la propria condizione, lasciando fermo quello che avrebbe potuto avanzare. `notifyAll()` non elimina la necessità del `while`, ma evita di basare il protocollo sulla scelta non garantita del thread risvegliato.

La variante temporizzata `wait(millis)` non trasforma l'attesa in un appuntamento esatto. Può ritornare per notifica, interruzione, risveglio spurio o trascorso del tempo. Il codice deve ricontrollare la condizione e, se esiste una scadenza complessiva, ricalcolare quanto tempo resta. Ripetere `wait(1000)` in un ciclo senza aggiornare il tempo potrebbe attendere molto più del limite che il chiamante immaginava. Per molte applicazioni le astrazioni di `java.util.concurrent`, come code e lock con condizioni, riducono il rischio di sbagliare questi dettagli; capire `wait` rimane comunque utile per leggere codice esistente e la semantica su cui si basano vari strumenti.

> **Approfondimento – Interruzione durante `wait`.** Se un thread viene interrotto mentre aspetta, `wait` lancia `InterruptedException` dopo aver riacquisito il monitor secondo le regole della specifica. Il metodo chiamante deve decidere se propagare l'eccezione, ripristinare il segnale e terminare, oppure gestire esplicitamente una chiusura. Catturare l'eccezione e ripetere subito il `while` senza politica può trasformare una richiesta di arresto in un'attesa infinita.

Una coda è anche una decisione sul carico

Fin qui abbiamo visto `BlockingQueue` come rimedio al problema di leggere un elemento non ancora prodotto. C'è un secondo problema: quanto lavoro permettiamo al produttore di accumulare? Se il produttore genera cento elementi al secondo e il consumatore ne usa cinquanta, una coda che cresce senza un limite definito accumula il ritardo. All'inizio tutto sembra funzionare; dopo un'ora la memoria occupata e il tempo necessario a recuperare il lavoro possono essere molto grandi. Una coda limitata rende visibile la differenza di velocità: quando è piena, `put` sospende il produttore finché il consumatore libera spazio. Questo meccanismo si chiama spesso *backpressure*, cioè pressione di ritorno dal lato che consuma verso quello che produce.

Il numero due dell'esempio è scelto per vedere facilmente il blocco, non è una capacità da usare in ogni applicazione. Per scegliere una capacità reale bisogna conoscere dimensione degli elementi, frequenza di produzione, tempi di consumo e latenza accettabile. Una coda molto piccola può frenare inutilmente un produttore che lavora a raffiche brevi; una molto grande può nascondere un consumatore cronicamente in ritardo. La capacità non sostituisce i limiti sull'intero sistema: più code, più richieste e più istanze possono sommare memoria anche quando ciascuna coda è formalmente limitata.

Una `LinkedBlockingQueue` può essere costruita con una capacità esplicita; senza una capacità applicativa scelta dal progettista ha un limite tecnico molto grande. Una `ArrayBlockingQueue` ha capacità fissata alla costruzione. Queste differenze contano più della preferenza per un nome di classe: il contratto di pieno e vuoto deve essere chiaro a chi chiama. Anche la scelta fra `offer` e `put` riflette una politica. Con `offer` immediato il produttore può rifiutare lavoro quando la coda è piena; con `put` aspetta; con `offer` temporizzato concede una finestra e poi restituisce `false`. Quale dei tre sia corretto dipende dalla promessa fatta all'utente o al componente che invia il lavoro.

> **Caso concreto – Una richiesta web.** Un servizio che riceve un ordine non dovrebbe rispondere «accettato» se ha soltanto tentato `offer` e ha ricevuto `false`. Deve riconoscere il rifiuto e comunicarlo, riprovare secondo una politica dichiarata o usare una consegna affidabile diversa dalla memoria del processo. La coda di questo capitolo è volatile: se il processo termina, il suo contenuto non diventa automaticamente persistente. Il buon uso della concorrenza comincia dal significato delle risposte, non dal numero di thread.

Quando due consumatori condividono una coda, ogni elemento prelevato va a uno solo dei due. La coda non duplica il messaggio. Se entrambi devono vedere tutti gli elementi, serve un modello di pubblicazione differente, con una copia o un canale per ciascuno. Due consumatori di una stessa coda sono quindi lavoratori che si dividono il carico, non osservatori che ricevono entrambi lo stesso flusso. Il loro ordine di stampa cambia fra esecuzioni, ma il requisito utile è che nessun elemento venga perso o consumato due volte in quel modello.

La fine del lavoro è un messaggio del protocollo. La sentinella `-1` nell'esempio non appartiene ai dati normali, quindi comunica «non arriveranno altri numeri». Con due consumatori e una sola sentinella, uno può terminare e l'altro rimanere bloccato in `take()`. Occorre inserire due sentinelle, una per ciascun consumatore,

oppure disegnare un altro meccanismo di chiusura. Se il numero dei consumatori varia durante l'esecuzione, le sentinelle richiedono particolare attenzione: bisogna sapere chi è attivo al momento della chiusura. Un'astrazione più alta può rendere più semplice questo contratto, ma non può decidere al posto dell'applicazione che cosa significhi «fine».

L'interruzione è un altro possibile evento di chiusura. Un consumatore bloccato in `take()` può essere interrotto; il metodo lancia `InterruptedException`. Se il ciclo cattura l'eccezione e continua senza criterio, la richiesta di arresto non viene rispettata. Se esce, il programma deve decidere cosa fare degli elementi già prelevati o rimasti nella coda. In una prova didattica stampare i valori è sufficiente; in un sistema di ordini servono regole su conferma, ripetizione ed eventuale perdita. Il capitolo conserva il modello semplice per insegnare la sincronizzazione, ma esplicita il limite della soluzione.

Dalla stampa casuale alla verifica corretta

La concorrenza invita a eseguire il programma una volta e a trarre conclusioni dall'ordine osservato. Molte righe prodotte da thread con priorità diverse o da più consumatori mostrano una sola esecuzione: non descrivono tutte le intercalazioni possibili. Un test di concorrenza deve formulare proprietà che restano vere sotto intercalazioni diverse. Per il contatore protetto la proprietà è «dopo che entrambi i thread sono terminati, il valore è ventimila». Per la coda con un produttore e un consumatore è «i valori zero-quattro sono ricevuti nell'ordine d'inserimento e il consumatore termina». Per due consumatori è «ogni valore è ricevuto una sola volta nel totale», senza vincolare il nome del lavoratore che lo riceve.

L'assenza di un errore in cento esecuzioni non dimostra l'assenza di una race condition. Un programma difettoso può stampare sempre il valore atteso sulla nostra macchina e fallire solo con un carico diverso. Viceversa, un output in ordine diverso non è per forza un errore: può essere una delle sequenze consentite dal contratto. Separare proprietà del programma e dettagli della pianificazione è un passo fondamentale per testare la concorrenza senza inventare garanzie.

Un modo utile di leggere un programma concorrente consiste nel tracciare gli eventi che devono essere ordinati. Nel contatore: avvio dei due lavoratori, ciascun incremento sotto il monitor, loro conclusione, ritorno di entrambe le `join`, lettura del risultato. Gli incrementi del primo e del secondo non hanno un ordine globale prefissato, ma sono serializzati quando entrano nello stesso metodo sincronizzato. La lettura finale avviene dopo le due conclusioni. Se togliamo `join`, la lettura potrebbe avvenire prima che tutti gli incrementi siano stati tentati, anche se ogni incremento fosse atomico. Sono due errori diversi: mancanza di attesa del completamento e mancanza di protezione degli aggiornamenti.

Per il deposito manuale, la traccia parte dallo stato «vuoto». `preleva` che arriva per primo aspetta e rilascia il monitor. `inserisci(0)` entra, trova il posto libero, scrive zero e notifica. Il consumatore torna candidato, riacquisisce il monitor, ricontrolla `prodotto != null`, prende zero, svuota il posto e notifica. Se il produttore arriva prima, inserisce zero; il consumatore lo trova senza dover attendere. Entrambe le storie rispettano lo stesso contratto. La spiegazione del protocollo deve funzionare per entrambe, altrimenti dipende da un ordine favorevole e non dal codice.

> **Metodo di lettura – Tre colonne.** Per analizzare un caso difficile, si può disegnare una colonna per ciascun thread e una per lo stato condiviso. Si scrivono letture, scritture, acquisizioni dei lock e attese. Non è necessario elencare ogni istruzione locale: interessano gli eventi che possono influire sull'altro thread. Questo schema rende visibile la lettura doppia del vecchio saldo, il `while` che protegge il deposito e il ciclo di attesa di un deadlock.

La verifica automatica può ripetere gli esperimenti, ma deve avere un limite di tempo per non restare bloccata su un errore di terminazione. Per un esempio che usa interruzioni o code, un test che non termina è già un difetto rilevante. Invece di dormire un tempo arbitrario e sperare che tutto sia finito, il test attende le attività con `join` o `Future.get()` e controlla la proprietà finale. Se deve evitare attese infinite, usa una variante con timeout e poi gestisce esplicitamente l'attività ancora in corso. Anche in un test, un timeout non è una cancellazione implicita.

Laboratorio: ricostruire gli esempi senza nascondere i passaggi

I programmi usati nei paragrafi precedenti sono piccoli perché ogni prova isola una proprietà. Metterli uno accanto all'altro permette di vedere quali righe hanno un ruolo di coordinamento e quali appartengono al lavoro applicativo. I file completi sono collegati ai rispettivi paragrafi; qui seguiamo una lettura progressiva. Il lettore può compilare ogni sorgente con `javac --release 25 -Xlint:all` e confrontare le righe osservate con le proprietà previste. Nessuno richiede opzioni preview.

Primo passaggio: avviare e attendere

Il programma `AvvioEJoin` chiama il medesimo `Runnable` in due modi. Nel primo caso l'esecuzione rimane nel thread `main`. Nel secondo `start` crea una nuova attività e `join` permette di osservare lo stato concluso. La riga che stampa `dopo run diretto: NEW` è decisiva: l'oggetto `Thread` è ancora nuovo, benché il metodo `run` sia stato eseguito come normale chiamata. Quando si rilegge il codice, conviene indicare con un segno il punto in cui l'attività concorrente nasce davvero.

```
public class AvvioEJoin {
    public static void main(String[] args) throws InterruptedException {
        Runnable lavoro = () -> System.out.println(
                "esegue: " + Thread.currentThread().getName());
        Thread lavoratore = new Thread(lavoro, "lavoratore");
        System.out.println("prima: " + lavoratore.getState());
        lavoratore.run();
        System.out.println("dopo run diretto: " + lavoratore.getState());
        lavoratore.start();
        lavoratore.join();
        System.out.println("dopo join: " + lavoratore.getState());
    }
}
```

La sequenza stampata è `NEW`, `main`, ancora `NEW`, `lavoratore`, `TERMINATED`, con le etichette del programma. Non è una coincidenza favorevole dello scheduler: la

prima chiamata precede `start`, e l'ultima stampa segue `join`. Se togliamo `join`, il programma può osservare `RUNNABLE` o `TERMINATED` a seconda del momento, e non può dichiarare completato il lavoro soltanto perché ha chiamato `start`.

Secondo passaggio: proteggere un aggiornamento

Nel contatore seguente il lavoro dei due thread è identico. Il metodo `incrementa` è breve, ma è proprio il punto in cui una lettura, un calcolo e una scrittura devono apparire come una sezione non interferita dall'altro thread. Anche la lettura finale passa dal metodo sincronizzato `leggi`. Il programma aspetta la conclusione di entrambi i thread prima di chiedere il valore. Se si cambiasse il numero di iterazioni a 100, il risultato atteso diventerebbe 200: il ragionamento non dipende dalla cifra particolare diecimila.

```
public class ContatoreProtetto {
    private int valore;

    public synchronized void incrementa() {
        valore++;
    }

    public synchronized int leggi() {
        return valore;
    }

    public static void main(String[] args) throws InterruptedException {
        ContatoreProtetto contatore = new ContatoreProtetto();
        Runnable lavoro = () -> {
            for (int i = 0; i < 10_000; i++) {
                contatore.incrementa();


            }
        };
        Thread primo = Thread.ofPlatform().start(lavoro);
        Thread secondo = Thread.ofPlatform().start(lavoro);
        primo.join();
        secondo.join();
        System.out.println(contatore.leggi());
    }
}
```

Un esperimento utile consiste nel togliere `synchronized` solo da `incrementa` in una copia del file. Il codice continua a compilare e può persino stampare `20000`; questo non dimostra che sia corretto. Il lettore deve individuare il punto in cui due intercalazioni possono leggere lo stesso vecchio valore. Ripristinata la versione protetta, si può cambiare il numero dei lavoratori, conservando un riferimento a ciascuno e chiamando `join` su tutti prima della lettura.

Terzo passaggio: separare visibilità e incremento

Il segnale di arresto usa `volatile` perché una scrittura di `true` deve essere osservata dal lavoratore. Il programma non incrementa un contatore nello stesso campo e non pretende che la parola chiave renda indivisibili più istruzioni. `onSpinWait()` è incluso per mostrare che il ciclo è un'attesa attiva; il programma lo termina subito. Per una vera attesa di durata incerta sarebbe più appropriato un meccanismo bloccante o una richiesta di interruzione gestita esplicitamente.

```
public class SegnaleVolatile {
    private volatile boolean fermati;

    public static void main(String[] args) throws InterruptedException {
        SegnaleVolatile segnale = new SegnaleVolatile();
        Thread lavoratore = Thread.ofPlatform().start(() -> {
            while (!segnale.fermati) {
                Thread.onSpinWait();
            }
            System.out.println("fermato");
        });
        segnale.fermati = true;
        lavoratore.join();
    }
}
```

Qui `main` potrebbe scrivere `true` prima che il nuovo thread cominci davvero a eseguire il ciclo. Anche quel caso è corretto: il lavoratore leggerà il valore e uscirà. La prova non richiede che i due thread si alternino in un ordine speciale. Se si togliessero sia `volatile` sia ogni altra forma di sincronizzazione sul campo, il programma non avrebbe il contratto di visibilità su cui questa spiegazione si basa; non è un esperimento da lanciare senza un limite di tempo perché potrebbe non terminare.

Quarto passaggio: un deposito con un posto

Il deposito manuale raccoglie ciò che abbiamo imparato su monitor, condizione e risveglio. Il campo `prodotto` è l'unico posto disponibile. `null` significa «vuoto», mentre ogni intero inserito, compresa la sentinella `-1`, significa «pieno». Un metodo non può leggere il valore quando il posto è vuoto e l'altro non può sovrascriverlo quando è pieno. Il `while` custodisce queste due invarianti anche dopo un risveglio non utile. Per osservare la coda di eventi, possiamo compilare il file e verificare che le cinque righe `prelevato` contengano, nell'ordine, zero, uno, due, tre e quattro.

```
public class DepositoConAttesa {
    private Integer prodotto;

    public synchronized void inserisci(int numero) throws InterruptedException {
        while (prodotto != null) {
            wait();
        }
        prodotto = numero;
        notifyAll();
    }

    public synchronized int preleva() throws InterruptedException {
        while (prodotto == null) {
            wait();
        }
        int numero = prodotto;


        prodotto = null;
        notifyAll();
        return numero;
    }

    public static void main(String[] args) throws InterruptedException {
        DepositoConAttesa deposito = new DepositoConAttesa();
        Thread produttore = Thread.ofPlatform().start(() -> {
            try {
                for (int numero = 0; numero < 5; numero++) {
                    deposito.inserisci(numero);
                }
                deposito.inserisci(-1);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }


        });
        Thread consumatore = Thread.ofPlatform().start(() -> {
            try {
                while (true) {
                    int numero = deposito.preleva();
                    if (numero == -1) {
                        break;
                    }
                    System.out.println("prelevato: " + numero);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        produttore.join();
        consumatore.join();


    }
}
```

Il percorso va dal deposito senza sincronizzazione a uno con attesa, poi a più consumatori. La progressione resta utile se ogni passo rende visibile l'invariante che mancava. Con un solo consumatore, la forma corretta evita letture premature.

Con due consumatori, il lettore deve riconoscere che entrambi possono essere risvegliati ma uno solo potrà trovare un valore: chi arriva secondo torna ad aspettare grazie al `while`. Una versione con più consumatori richiede inoltre un protocollo di chiusura per tutti; copiare esattamente il `main` qui sopra e aggiungere un secondo consumatore senza cambiare la sentinella produrrebbe un programma che può non terminare.

Quinto passaggio: usare l'astrazione della libreria

Il deposito manuale insegna il meccanismo, ma ogni nuovo requisito richiede altro codice di protocollo. Una `BlockingQueue` dichiara già le operazioni di attesa e una capacità. Il programma seguente sostituisce il campo facoltativo e i due metodi sincronizzati con una coda limitata. Il lavoro applicativo non cambia: produrre cinque numeri, consumarli nell'ordine e terminare alla sentinella. Si può quindi confrontare il numero di decisioni che il programmatore deve prendere nelle due forme, senza sostenere che una coda risolva automaticamente tutti i problemi di chiusura.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class ProduttoreConsumatore {
    private static final int FINE = -1;

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> coda = new ArrayBlockingQueue<>(2);
        Thread produttore = Thread.ofVirtual().name("produttore").start(() -> {
            try {
                for (int numero = 0; numero < 5; numero++) {
                    coda.put(numero);
                }
                coda.put(FINE);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();

            }
        });
        Thread consumatore = Thread.ofVirtual().name("consumatore").start(() -> {
            try {
                while (true) {
                    int numero = coda.take();
                    if (numero == FINE) {
                        break;
                    }
                    System.out.println("consumato: " + numero);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        produttore.join();

        consumatore.join();
    }
}
```

I due thread qui sono virtuali, ma la coda applica le stesse condizioni di pieno e vuoto che avrebbe con thread di piattaforma. `main` aspetta entrambi. L'ordine delle stampe del solo consumatore è il contenuto ordinato del flusso; se aggiungessimo stampe nel produttore, le due serie di righe potrebbero intrecciarsi diversamente. Questa differenza fra ordine dei valori e ordine degli eventi di stampa va conservata quando si spiegano esempi concorrenti.

Sesto passaggio: un incremento atomico autonomo

Quando lo stato condiviso è un solo numero, `AtomicInteger` offre operazioni atomiche già definite. Il programma che segue non usa `volatile int`: l'operazione `incrementAndGet` comprende l'aggiornamento in modo atomico. `get` legge il valore finale dopo la conclusione dei due lavoratori. La somiglianza dell'output con `ContatoreProtetto` non significa che le due classi siano intercambiabili in ogni situazione. Se dovessimo mantenere coerenti più campi, un contatore atomico isolato non proteggerebbe l'intera invariante.

```
import java.util.concurrent.atomic.AtomicInteger;

public class ContatoreAtomico {
    public static void main(String[] args) throws InterruptedException {
        AtomicInteger valore = new AtomicInteger();
        Runnable lavoro = () -> {
            for (int i = 0; i < 10_000; i++) {
                valore.incrementAndGet();
            }
        };
        Thread primo = Thread.ofPlatform().start(lavoro);
        Thread secondo = Thread.ofPlatform().start(lavoro);
        primo.join();
        secondo.join();
        System.out.println(valore.get());
    }

}
```

Il nome `incrementAndGet` dice anche quale valore restituirebbe a chi lo chiama: il nuovo valore dopo l'incremento. `getAndIncrement`, invece, restituisce il valore precedente. Entrambi aggiornano atomicamente il numero, ma la differenza conta quando quel valore viene usato per assegnare un identificatore o una posizione. Nel programma non usiamo il valore di ritorno perché vogliamo soltanto contare; il valore finale viene letto una volta, dopo le `join`. Se si stampasse a ogni incremento, l'ordine delle ventimila righe non avrebbe un significato utile e la stampa stessa cambierebbe molto il profilo temporale dell'esperimento.

Settimo passaggio: attività con un risultato

L'esecutore virtuale separa la creazione dei lavori dalla gestione diretta dei riferimenti ai thread. Le due lambda passate a `submit` sono `Callable<Integer>` perché producono numeri. I `Future` rendono esplicito il momento in cui `main` vuole i risultati. `get()` può attendere; quando entrambe le chiamate sono terminate, la somma è deterministica. Il `try` con risorse chiude l'esecutore alla fine del blocco anche se un'operazione esce con un'eccezione gestita dal chiamante secondo il contratto dell'API.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class EsecutoreVirtuale {
    public static void main(String[] args)
            throws InterruptedException, ExecutionException {
        try (ExecutorService esecutore =
                Executors.newVirtualThreadPerTaskExecutor()) {
            Future<Integer> prima = esecutore.submit(() -> 20 + 2);
            Future<Integer> seconda = esecutore.submit(() -> 30 + 3);
            System.out.println(prima.get() + seconda.get());
        }
    }
}
```

In questo esempio il primo `get` potrebbe terminare prima del secondo, dopo il secondo o trovarlo già completato. La somma stampata resta 55 perché i due calcoli non condividono stato mutabile e `main` usa i risultati soltanto dopo averli ottenuti. Se sostituissimo i calcoli con due chiamate di rete, dovremmo decidere cosa fare quando una fallisce: aspettare l'altra, cancellarla, mostrare un risultato parziale o propagare l'errore. Il `Future` rappresenta lo stato del singolo lavoro, ma la politica complessiva appartiene all'operazione chiamante.

Piccola guida ai metodi di `Thread`

Raccogliamo in una tabella i metodi di `java.lang.Thread` che abbiamo incontrato. Alcuni sono facili da interpretare perché sembrano descrivere un'azione, ma il loro contratto richiede una precisazione. `start()` programma l'avvio di un nuovo thread; non attende che `run()` finisca. `run()` è il corpo del lavoro, ma una sua chiamata diretta rimane nel thread chiamante. `currentThread()` restituisce il thread che sta eseguendo in quel punto, non necessariamente quello da cui abbiamo ottenuto un riferimento in precedenza. `join()` aspetta la terminazione del thread su cui è invocato, mentre la variante con timeout può tornare anche se il thread è ancora vivo.

`interrupt()` segnala una richiesta di interruzione. `isInterrupted()` legge il segnale su un oggetto `Thread` senza cancellarlo; `interrupted()` è un metodo statico che opera sul thread corrente e cancella lo stato. `sleep()` sospende il thread corrente e può essere interrotto; non rilascia i monitor che il thread possiede. `yield()` offre un suggerimento allo scheduler. `getState()` fornisce uno degli stati osservabili al momento della chiamata: subito dopo, lo stato può essere già

cambiato. Una misura che interroga ripetutamente `getState()` non sostituisce un protocollo di coordinamento.

`setName` e `getName` riguardano la diagnostica; `setPriority` e `getPriority` riguardano una proprietà la cui efficacia sulla pianificazione è dipendente dalla piattaforma e dal tipo di thread. `setDaemon` va deciso prima dell'avvio per i thread di piattaforma; `isDaemon` permette di osservarlo. Il metodo `isAlive()` risponde se il thread è stato avviato e non è ancora terminato, ma una risposta `true` non significa che sia su un core in quell'istante. Tutte queste domande riguardano la *vita* o la *descrizione* del thread; nessuna di esse rende atomico un aggiornamento a un oggetto condiviso.

Metodo o gruppo	Domanda utile	Errore frequente
`start`, `run`	Nasce un thread nuovo oppure eseguo un metodo qui?	Chiamare `run` credendo di creare concorrenza
`join`, `isAlive`	Il lavoro è terminato?	Interpretare un timeout come completamento
`interrupt`, `isInterrupted`, `interrupted`	Come viene richiesta e osservata la chiusura?	Ingoiare l'interruzione o azzerarla senza saperlo
`sleep`, `yield`	Il thread corrente sospende o cede volontariamente?	Usarli come strumenti di sincronizzazione
`getState`	Quale stato è osservabile ora?	Supporre che lo stato resti uguale dopo la lettura
`getName`, `setName`	Come riconoscere il ruolo nei log?	Usare il nome come identità globale garantita
`getPriority`, `setPriority`	Quale priorità è configurata?	Pretendere un ordine di esecuzione fisso
`isDaemon`, `setDaemon`	Il thread tiene in vita la JVM?	Lasciare lavoro essenziale a un daemon non atteso

Questa tabella è un indice per leggere le prove, non sostituisce le firme dell'API. Per esempio, `join()` può lanciare `InterruptedException`; `setDaemon` e `setPriority` hanno vincoli propri. Quando il comportamento è necessario alla correttezza, si consulta la documentazione della versione del JDK usata. La panoramica serve a collegare ogni metodo alla decisione pratica che il lettore deve prendere.

Un esperimento controllato sulle priorità

Si possono avviare due lavori finiti, ciascuno con un conteggio locale e una sola stampa finale, assegnando priorità differenti ai thread di piattaforma. Se ripetiamo la prova, può cambiare quale riga finale compare per prima. Il risultato non permette di inferire una legge come «la priorità alta termina sempre prima». Il sistema operativo, la JVM e il carico della macchina partecipano alla pianificazione; la specifica non trasforma la priorità in una precedenza funzionale fra risultati. Per rendere deterministico un ordine, si può far attendere il secondo lavoro con un coordinamento esplicito, ma a quel punto è il coordinamento a fornire la garanzia, non la priorità.

Una stampa continua in un ciclo senza fine amplificherebbe visivamente le differenze osservate su una macchina, ma renderebbe difficile fermare il programma e mescolerebbe il costo della console con la pianificazione dei thread. Limitiamo il numero di iterazioni e dichiariamo che l'ordine di arrivo non è un'asserzione di test. L'osservazione critica dello scheduler insegna che quando non definiamo una relazione fra le attività, non possiamo usare l'intercalazione delle loro stampe come parte del contratto del programma.

Scegliere il modello partendo dal bisogno

Ricapitoliamo tre richieste diverse. Se dobbiamo contare accessi indipendenti, un `AtomicInteger` può essere sufficiente. Se dobbiamo cambiare insieme saldo e storico di un conto, serve una sezione critica che protegga l'intera invariante, oppure una transazione al livello che possiede i dati. Se dobbiamo trasferire lavoro da un produttore a un consumatore, una `BlockingQueue` esprime il passaggio e la disponibilità. Usare il medesimo strumento per tutti e tre i casi rende il programma meno chiaro e può lasciarlo scorretto.

La scelta fra thread di piattaforma e virtuali arriva dopo aver identificato la natura del lavoro. Per poche attività lunghe che impegnano la CPU, un numero controllato di lavoratori può essere ragionevole. Per molte attività che aspettano I/O, i thread virtuali offrono uno stile semplice. In entrambi i casi, la condivisione dei dati mantiene le stesse regole. Il numero di thread non può essere il primo parametro da ottimizzare se non abbiamo ancora stabilito che cosa significhi «risultato corretto» e quando l'operazione deve terminare.

> **Per verificare la comprensione.** Si prenda il programma del deposito e si descrivano due esecuzioni: una in cui il consumatore entra per primo, una in cui entra per primo il produttore. In entrambe si deve poter giustificare la stampa dei cinque valori. Poi si immagini un secondo consumatore e si spieghi perché il `while` resta indispensabile e perché occorre una seconda sentinella. Infine si scelga se un contatore di ordini completati richiede soltanto un numero atomico o una modifica coordinata con lo stato degli ordini. La risposta deve citare l'invariante che si vuole mantenere.

Errori che una stampa ordinata può nascondere

È possibile che un programma con una race condition produca righe ordinate e rassicuranti. Nel caso del conto, per esempio, i due prelievi possono apparire uno dopo l'altro nella console pur avendo entrambi letto il saldo prima della stampa. La console è una risorsa con proprie regole di sincronizzazione; il fatto che due messaggi non si sovrappongano carattere per carattere non dimostra che le operazioni di dominio siano state coordinate. Per valutare il programma occorre identificare le letture e le scritture del saldo, il lock che le protegge e il punto in cui l'esito viene deciso.

Un altro errore frequente consiste nell'usare una collezione ordinaria da più thread perché ogni thread «tocca elementi diversi». Se i thread modificano la struttura della collezione, per esempio aggiungendo a una `ArrayList`, possono interferire sulla dimensione, sull'array interno e sugli indici anche quando gli oggetti aggiunti sono distinti. La scelta corretta può essere una collezione concorrente, un lock intorno alla modifica o la raccolta di risultati locali da unire dopo `join`. Il punto è capire che cosa è condiviso: non basta che i dati logici sembrino diversi se il contenitore fisico è lo stesso.

Il caso opposto è condividere soltanto oggetti immutabili. Se un thread costruisce un record con dati finali, lo consegna attraverso una `BlockingQueue` e un altro lo legge, il consumatore non deve coordinare modifiche interne che non esistono. Deve comunque ricevere l'oggetto tramite un canale con le necessarie garanzie di pubblicazione. La coda offre questo passaggio secondo il proprio contratto. L'immutabilità riduce il numero di invarianti da proteggere, ma non cancella la necessità di definire come e quando il riferimento passa da un thread all'altro.

> **Approfondimento – Pubblicazione sicura.** Un oggetto può essere costruito da un thread e reso visibile a un altro. Le regole del Java Memory Model stabiliscono quando il secondo thread vede correttamente gli effetti della costruzione e delle scritture precedenti. Una coda concorrente, un monitor usato correttamente o la conclusione osservata con `join` forniscono relazioni utili. Passare il riferimento attraverso un campo ordinario scritto e letto senza protocollo può essere una data race. Non si risolve perché l'oggetto «è stato creato prima» secondo l'orologio della parete.

Dal ciclo infinito alla chiusura esplicita

Molti esempi introduttivi usano `while (true)` perché rende evidente il lavoro ripetuto. In un servizio vero occorre rispondere a una domanda ulteriore: chi decide quando fermarsi? Un thread che legge da una coda può ricevere una sentinella; uno che aspetta I/O può essere interrotto; un esecutore può essere chiuso e attendere i lavori già accettati. La scelta deve essere compatibile con le risorse in uso. Se il thread possiede uno stream, una connessione o un file, la terminazione deve condurre anche alla chiusura di quella risorsa.

La sentinella dell'esempio ha una proprietà comoda: viaggia nello stesso canale dei dati. Arriva dopo i cinque numeri inseriti dal singolo produttore, quindi il consumatore non può incontrarla prima dei valori precedenti. Con più produttori, però, uno potrebbe inserire la sentinella mentre un altro ha ancora dati da

produrre. Serve un coordinatore che sappia quando tutti i produttori hanno finito, oppure un protocollo più ricco. Ancora una volta, il meccanismo `put/take` è corretto, ma il significato dell'ultimo elemento dipende dall'organizzazione del lavoro.

L'interruzione è una forma di richiesta che non viaggia nella coda. Può svegliare un thread da `take`, ma non descrive di per sé se i dati rimasti vadano completati, restituiti a un altro lavoratore o scartati. Una strategia può essere «termina appena possibile»; un'altra «completa l'elemento già preso e poi termina». Entrambe possono essere legittime, ma vanno rese esplicite. Per questo `catch (InterruptedException e) {}` non è soltanto una cattiva abitudine stilistica: elimina l'informazione che il sistema usa per gestire la durata dell'attività.

Nel codice di esempio il `catch` ripristina il segnale e lascia terminare la lambda. La forma è adatta alla prova normale, in cui nessun thread viene interrotto prima della sentinella. Se volessimo provare la cancellazione a metà produzione, dovremmo aggiungere un percorso che svegli e concluda anche il consumatore. Un test che interrompe il produttore e poi aspetta indefinitamente il consumatore rivelerebbe un difetto del protocollo di chiusura, non della coda. Tenere separati trasferimento dei dati e politica di terminazione rende questo difetto più facile da vedere.

Il costo nascosto dell'attesa attiva

Nel segnale `volatile` il lavoratore esegue un ciclo che controlla ripetutamente il campo `fermati`. Questo è *busy waiting*, o attesa attiva: mentre attende, il thread continua a occupare tempo di elaborazione. `Thread.onSpinWait()` può aiutare la piattaforma a ottimizzare un ciclo di spin molto breve, ma non trasforma l'attesa in un blocco senza costo. Per una richiesta che può arrivare dopo secondi o minuti, conviene scegliere un'attesa bloccante o un meccanismo di notifica. L'esempio resta utile perché isola la visibilità senza confonderla con altri strumenti.

Aggiungere `Thread.sleep(1)` al ciclo ridurrebbe il numero di controlli, ma introdurrebbe una latenza arbitraria e non renderebbe corretta una versione non `volatile`. La specifica chiarisce che `sleep` non è una barriera di sincronizzazione. Il thread potrebbe ancora non avere una relazione che obblighi a osservare la scrittura dell'altro. Se il requisito è «svegliati quando arriva un evento», la forma appropriata è una primitive di coordinamento, non un sondaggio temporizzato scelto per tentativi.

Questo esempio permette di separare due tipi di efficienza. Un thread bloccato non consuma continuamente CPU per chiedere se la condizione è cambiata; un thread in spin può reagire con poca latenza se l'attesa è davvero brevissima e il sistema ha risorse disponibili. La scelta richiede misure e conoscenza del carico. In un manuale di base non serve trasformare l'eccezione di micro-ottimizzazione in consiglio generale: prima si scrive un protocollo corretto, poi si misura se il costo d'attesa è un problema reale.

Mettere alla prova una spiegazione di concorrenza

Una buona spiegazione deve poter rispondere a quattro domande. Quale stato è condiviso? Quali thread lo leggono o lo scrivono? Quale regola deve rimanere vera? Quale evento autorizza ciascun thread a proseguire? Nel deposito, lo stato è `prodotto`; produttore e consumatore lo modificano sotto lo stesso monitor; il posto non può essere contemporaneamente vuoto e pieno; i `while` attendono la condizione necessaria. Nel contatore, lo stato è `valore`; i due lavoratori lo incrementano; ogni incremento deve contare una volta; il monitor serializza la sezione critica e `join` permette di leggere il totale finale.

Se una di queste risposte manca, una prova che «funziona sul mio computer» non colma il vuoto. Lo stesso vale per il codice di un libro. Un esempio che stampa due righe in un certo ordine deve spiegare quale istruzione o quale garanzia impone quell'ordine. Se non esiste, la didascalia deve chiamarlo output possibile, non output necessario. Questa disciplina è particolarmente importante quando il testo vuole insegnare a diagnosticare: il lettore deve imparare a distinguere una prova ripetibile da una fotografia di una singola esecuzione.

Una tecnica utile per l'autoverifica è cambiare deliberatamente la pianificazione senza cambiare il contratto: aggiungere lavoro locale a uno dei due thread, invertire l'ordine di `start`, usare thread virtuali al posto di quelli di piattaforma dove il programma lo consente. La proprietà finale deve restare vera se la sincronizzazione è corretta. Non è un test formale di tutti gli interleaving, ma costringe a formulare un'affermazione precisa. Se il programma passa solo quando un thread «arriva prima», la soluzione dipende dal caso e deve essere ripensata.

Collegare il capitolo al resto del libro

I thread eseguono metodi di classi ordinarie: valgono ancora incapsulamento, generics, eccezioni e contratti delle collezioni. Una `BlockingQueue<Integer>` usa i generics per dichiarare il tipo degli elementi; `InterruptedException` impone al chiamante una decisione di gestione; la lambda `Runnable` cattura riferimenti secondo le regole viste nella programmazione funzionale. La concorrenza non è un mondo separato in cui le regole precedenti scompaiono. Aggiunge una dimensione: più percorsi di esecuzione possono osservare e modificare lo stesso stato.

Questo collegamento spiega anche la preferenza per oggetti con responsabilità chiare. Se la classe `DepositoConAttesa` possiede il valore e il protocollo di attesa, chi la usa non deve sapere quando chiamare `wait` o su quale monitor sincronizzarsi. Se il protocollo fosse distribuito fra produttore e consumatore, ciascuno dovrebbe conoscere dettagli interni dell'altro e una modifica rischierebbe di rompere l'accordo. Un confine di classe ben scelto aiuta la correttezza concorrente oltre che la leggibilità.

Infine, il passaggio al capitolo sui classloader ricorda che la JVM gestisce sia attività sia definizioni di tipi durante l'esecuzione. Un thread può richiedere per la prima volta una classe mentre un altro thread sta lavorando; i meccanismi di caricamento hanno proprie garanzie di coordinamento. Non dobbiamo implementarle nei nostri esempi di contatore, ma dobbiamo evitare una conclusione sbagliata: la presenza di più thread non rende la JVM priva di regole. Il nostro compito è rispettare i contratti che le API e il linguaggio definiscono, non assumere un ordine che nessuno ha promesso.

Una traccia finale da saper raccontare

Per controllare di aver capito il capitolo, immaginiamo che `main` crei una coda con due posti, avvii un produttore e un consumatore virtuali e attenda entrambi. Il produttore inserisce 0 e 1; a quel punto la coda potrebbe essere piena oppure il consumatore potrebbe aver già preso un valore. Se è piena, `put(2)` attende. Quando il consumatore esegue `take()`, libera un posto e permette al produttore di avanzare. Nessuna di queste possibilità cambia l'ordine dei valori che il singolo consumatore riceve dalla coda. Dopo 4 arriva la sentinella, che fa uscire il consumatore. Le due `join` assicurano che `main` non dichiari finito il programma dimostrativo prima dei lavoratori.

La stessa traccia, letta con il deposito manuale, richiede di citare il monitor e i due `while`. Se il consumatore arriva prima, `wait` rilascia il monitor; il produttore può inserire 0; `notifyAll` permette al consumatore di tornare candidato; il consumatore riacquisisce il monitor e controlla la condizione. Se il produttore arriva prima, il consumatore trova il valore senza attendere. Una spiegazione che funziona solo per il primo caso non ha ancora descritto il protocollo: ha descritto una sola possibile sequenza di eventi.

Infine, se qualcuno propone di sostituire tutto con `Thread.sleep(100)` dopo `start`, chiediamo quale contratto garantisca che cento millisecondi bastino, che il valore sia visibile e che l'operazione sia completa. Non c'è una risposta basata soltanto su `sleep`. Il percorso del capitolo conduce proprio qui: definire la proprietà, scegliere lo strumento che la garantisce, verificare l'esito senza trasformare lo scheduler in un oracolo.

> **Domanda di controllo – Perché non basta una prova riuscita?** Un programma con `valore++` non protetto può stampare il totale atteso anche molte volte di seguito. Quel risultato significa soltanto che le intercalazioni osservate non hanno reso visibile la perdita, oppure che il difetto non è emerso in quelle condizioni. Il contratto del linguaggio non promette l'atomicità dell'espressione. La prova di correttezza passa dal protocollo usato da tutti i thread: monitor comune o operazione atomica appropriata. Dopo averlo stabilito, i test verificano che l'implementazione e la gestione della durata rispettino il protocollo. Le due forme di evidenza si completano: una spiegazione teorica senza esecuzione può nascondere un errore nel codice, mentre un'esecuzione riuscita senza modello può nascondere una race condition.

Nel capitolo 20A sulle operazioni atomiche partiamo dal contatore di queste pagine e cambiamo domanda: che cosa accade quando l'incremento o il decremento deve anche decidere se una richiesta può essere accettata? Il caso delle ultime copie disponibili porta a `compareAndSet`; una statistica molto aggiornata porta invece a valutare `LongAdder`. Il criterio resta quello costruito qui: prima si identifica l'invariante, poi si sceglie il confine che la protegge.

Un secondo controllo riguarda il significato di «terminato». Un thread può aver concluso il proprio `run`, ma il programma può ancora dover leggere un risultato, chiudere una risorsa o comunicare un errore al chiamante. `join` attende il thread; `Future.get` attende e riporta anche il risultato o il fallimento dell'attività. La scelta dipende dall'informazione che il chiamante deve ricevere. Questa è la ragione per

cui i due esempi finali del capitolo usano strumenti diversi pur potendo entrambi eseguire un calcolo in un altro thread.

20A. Operazioni atomiche: una decisione condivisa alla volta

L'ultima copia del libro

Nel capitolo 20 abbiamo affidato a due thread lo stesso contatore. Abbiamo visto che `valore++` non è una sola azione: legge un numero, ne calcola un altro e lo scrive. Quando due thread percorrono quei passaggi insieme, uno degli incrementi può andare perduto. Un monitor comune, usato tramite `synchronized`, ha risolto il problema. Abbiamo anche incontrato `AtomicInteger`, che sa incrementare un valore condiviso senza farci scrivere quel monitor. Perché allora dedicare un altro capitolo agli atomici? Perché l'incremento è il caso facile. Nelle applicazioni dobbiamo spesso prendere una **decisione** in base al valore letto.

Immaginiamo una biblioteca con cinque copie prenotabili di un libro. Arrivano venti richieste quasi contemporaneamente. Una prenotazione può riuscire soltanto se rimane almeno una copia; dopo cinque successi, tutte le altre devono essere respinte. La condizione da conservare è semplice da dire: il numero di copie disponibili non deve mai diventare negativo e le richieste accettate non devono superare cinque. Questa condizione è un'*invariante*: deve rimanere vera in ogni stato che gli altri thread possono osservare.

Il codice `if (copieDisponibili > 0) copieDisponibili--;` sembra già raccontare la regola. In realtà un thread può leggere `1`, fermarsi, lasciare che un altro legga anch'esso `1` e solo dopo proseguire. Entrambi hanno deciso di accettare l'ultima copia. Anche dichiarando il campo `volatile` non trasformiamo controllo e decremento in una decisione indivisibile: `volatile` riguarda visibilità e ordine degli accessi, non la composizione automatica di più istruzioni. Un blocco `synchronized` che racchiude **sia** il controllo **sia** la modifica sarebbe corretto; questa volta vogliamo capire se possiamo esprimere la stessa decisione per una sola variabile con un'operazione atomica.

> **Definizione – Operazione atomica.** Per gli accessi che seguono lo stesso protocollo, un’operazione atomica appare come un passaggio indivisibile sullo stato coinvolto. Un incremento atomico impedisce che un altro incremento si inserisca fra lettura e scrittura. Il **CAS** aggiunge una condizione: sostituisce il valore soltanto se coincide con quello atteso; altrimenti segnala il mancato aggiornamento. Non ogni operazione atomica richiede un valore atteso dal chiamante. La garanzia copre soltanto lo stato dell’operazione: non rende indivisibili anche la richiesta applicativa, la stampa o la scrittura su disco. Occorre quindi dichiarare quali valori devono cambiare insieme.

La libreria è già nel JDK

Java offre `java.util.concurrent.atomic` nel modulo standard `java.base`. `AtomicInteger` custodisce un `int`; `AtomicLong` custodisce un `long`; `AtomicBoolean` un booleano; `AtomicReference<T>` un riferimento a un oggetto. Le classi non sono sostituti generici dei normali `Integer`, `Long` o `Boolean`: sono oggetti mutabili progettati per far comunicare thread che condividono uno stato. Nel nucleo del capitolo usiamo solo API stabili di Java 25, senza dipendenze esterne e senza opzioni preview.

Riprendiamo il contatore atomico del capitolo 20. Il metodo `incrementAndGet()` aggiorna il contatore e restituisce il **nuovo** valore; `getAndIncrement()` aggiorna lo stesso contatore ma restituisce il **precedente**. Se partiamo da 7, la prima chiamata a `getAndIncrement()` restituisce 7 e lascia 8; una successiva `incrementAndGet()` restituisce 9 e lascia 9. Questa distinzione è importante quando il risultato diventa un numero di biglietto o una posizione: un errore di un'unità può assegnare due volte lo stesso numero o saltarne uno.

Un contatore indipendente è un buon problema per `AtomicInteger`. Se però il numero governa una decisione, leggere con `get()` e modificare in una chiamata separata non è sufficiente. La figura 20A.1 mette a confronto le due sequenze quando resta una sola copia.

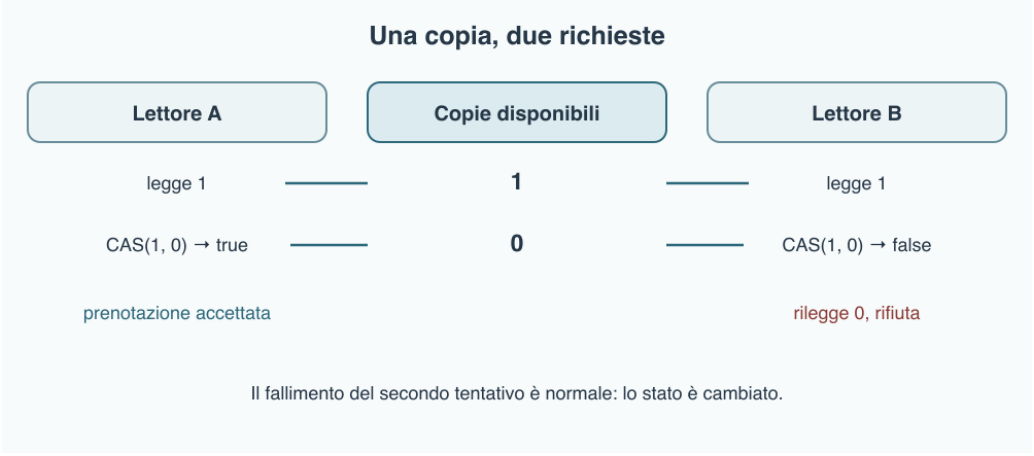


Figura 20A.1 – Due lettori possono osservare «1», ma uno solo può sostituirlo con «0» quando il confronto e la sostituzione costituiscono un'unica operazione.

Confrontare e sostituire

`compareAndSet` viene spesso abbreviato in *CAS*, da *compare and set*. Riceve il valore che ci aspettiamo e quello che vorremmo installare. Se il valore corrente coincide con quello atteso, lo sostituisce e restituisce `true`. In caso contrario lascia lo stato invariato e restituisce `false`. Il confronto del nostro `AtomicInteger` riguarda due `int`; più avanti vedremo che per `AtomicReference` riguarda l'identità dei riferimenti.

Supponiamo che A e B leggano entrambi `1`. A tenta `compareAndSet(1, 0)` e riesce. B tenta la stessa sostituzione, ma trova `0`: riceve `false`. B non deve concludere che si è verificato un errore della libreria; deve rileggere lo stato. Alla nuova lettura

trova zero e rifiuta la prenotazione. Il fallimento di un CAS è una normale possibilità del programma concorrente. Questa breve ripetizione forma un *ciclo CAS*: osservare, proporre un nuovo valore, tentare, riprovare se qualcuno lo ha cambiato.

Il programma completo `PrenotazioniAtomiche` applica la regola. Lo si compila dalla directory `esempi/C20A` con `javac --release 25 PrenotazioniAtomiche.java` e lo si esegue con `java PrenotazioniAtomiche`. Usa thread virtuali, già spiegati nel capitolo 20, per far partire venti richieste. La scelta del tipo di thread non rende atomico il contatore: la correttezza viene dal protocollo di aggiornamento. Due `CountDownLatch` organizzano la prova: ogni lavoratore segnala su `pronti` di essere arrivato al cancello, poi attende `via.await()`. Il thread principale aspetta i venti segnali e apre il cancello con `countDown()`. Non garantisce un ordine di arrivo, ma rende plausibile la sovrapposizione delle richieste senza ricorrere a `sleep`.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicInteger;

public class PrenotazioniAtomiche {
    private final AtomicInteger copieDisponibili;

    public PrenotazioniAtomiche(int copieIniziali) {
        if (copieIniziali < 0) {
            throw new IllegalArgumentException("Le copie non possono essere negative");
        }
        copieDisponibili = new AtomicInteger(copieIniziali);
    }

    public boolean prenota() {

        while (true) {
            int osservate = copieDisponibili.get();
            if (osservate == 0) {
                return false;
            }
            if (copieDisponibili.compareAndSet(osservate, osservate - 1)) {
                return true;
            }
        }
    }

    public int disponibili() {
        return copieDisponibili.get();
    }

    public static void main(String[] args) throws InterruptedException {
```

```
        PrenotazioniAtomiche catalogo = new PrenotazioniAtomiche(5);
        AtomicInteger accettate = new AtomicInteger();
        CountDownLatch pronti = new CountDownLatch(20);
        CountDownLatch via = new CountDownLatch(1);
        List<Thread> lettori = new ArrayList<>();

        for (int i = 0; i < 20; i++) {
            Thread lettore = Thread.ofVirtual().start(() -> {
                try {
                    pronti.countDown();
                    via.await();
                    if (catalogo.prenota()) {
                        accettate.incrementAndGet();
                    }
                } catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();


                }
            });
            lettori.add(lettore);
        }

        pronti.await();
        via.countDown();
        for (Thread lettore : lettori) {
            lettore.join();
        }
        if (accettate.get() != 5 || catalogo.disponibili() != 0) {
            throw new AssertionError("Invariante delle copie violata");
        }
        System.out.println("Prenotazioni accettate: " + accettate.get());
        System.out.println("Copie disponibili: " + catalogo.disponibili());
    }

}
```

Fermiamoci sul metodo `prenota()`. `get()` legge il numero corrente. Se è zero, il metodo risponde `false` e non tenta una sottrazione. Altrimenti calcola `osservate - 1`, ma quel calcolo è solo una proposta. Il punto in cui la prenotazione riesce è il `compareAndSet`: la proposta viene installata soltanto se nessun altro ha cambiato il numero letto. Quando il CAS fallisce, il `while` rientra dall'inizio; non riusa il vecchio numero, perché il dato che giustificava la decisione non è più attuale. Il costruttore rifiuta un numero iniziale negativo, così l'invariante nasce vera. Un'applicazione reale definirebbe anche come aggiungere nuove copie, ma ogni metodo che modifica questo stesso numero dovrebbe rispettare il medesimo protocollo.

Nel `main`, venti lavoratori chiamano `prenota()`. Un secondo `AtomicInteger` conta quanti hanno ricevuto `true`; è soltanto un contatore della prova, non parte della prenotazione. Dopo ogni `join()`, il thread principale controlla `accettate == 5` e `disponibili == 0`. L'output finale è quindi stabile: `Prenotazioni accettate: 5` e `Copie disponibili: 0`. Non sappiamo quali cinque lettori abbiano ottenuto una copia, né possiamo dedurlo dal programma. Ripetere l'esecuzione aiuta a trovare errori d'implementazione, mentre il motivo per cui il limite è rispettato sta nel CAS e nel fatto che ogni successo sottrae esattamente uno da un valore positivo.

> **Concetto chiave – Dove cade la decisione.** Un `get()` isolato non prenota nulla. Un `compareAndSet` riuscito è il punto in cui il cambiamento diventa effettivo per quello stato atomico. Identificare questo punto rende possibile ragionare anche quando le righe di codice dei diversi thread si intrecciano in modo imprevedibile.

Il ciclo ha un limite di applicazione

Un ciclo CAS non mette il thread in attesa di un evento futuro. Se tutte le copie sono finite, il nostro metodo esce; non gira aspettando che qualcuno ne restituisca una. Se invece più thread modificano continuamente il contatore, un chiamante può dover ritentare molte volte. Non promettiamo che ogni tentativo individuale finisca entro un tempo fisso né che il CAS sia più veloce di un monitor su qualunque macchina. Quando il problema è *attendere* una copia restituita, servono un protocollo di attesa e una notifica, oppure una struttura come `BlockingQueue` o un `Semaphore`, secondo ciò che rappresentiamo. Sostituire l'attesa con un `while` che ripete `get()` spreca CPU e non definisce la politica di assegnazione.

Per un unico valore, `AtomicInteger` rende il confine dell'aggiornamento molto preciso. Questo non significa che ogni `if` seguito da un incremento possa essere trasformato meccanicamente in CAS. Dobbiamo controllare che il nuovo valore dipenda soltanto dallo stato letto e che il tentativo possa essere ripetuto senza produrre effetti esterni duplicati. Una prenotazione che addebita una carta o invia una conferma richiede un confine transazionale e una politica sui fallimenti; il CAS del contatore non li fornisce.

Le funzioni di aggiornamento possono essere richiamate più volte

La libreria offre `getAndUpdate` e `updateAndGet` quando vogliamo trasformare un valore senza scrivere esplicitamente il ciclo. Il primo restituisce il valore precedente, il secondo quello nuovo. A entrambi passiamo una funzione che riceve il valore corrente e calcola il candidato successivo. La funzione può essere applicata più volte se un altro thread interviene fra calcolo e aggiornamento: la documentazione richiede perciò che non produca effetti collaterali.

Nel programma `AggiornamentoPuro` aumentiamo di due il numero di posti, senza superare il limite dieci. `Math.min(10, valore + 2)` dipende solo dal parametro ricevuto; se l'API la ricalcola, non invia messaggi e non modifica altri oggetti. I valori piccoli dell'esempio evitano overflow aritmetico; un contatore senza limite deve invece decidere che cosa fare al raggiungimento di `Integer.MAX_VALUE`.

```
import java.util.concurrent.atomic.AtomicInteger;

public class AggiornamentoPuro {
    public static void main(String[] args) {
        AtomicInteger posti = new AtomicInteger(7);
        int precedente = posti.getAndUpdate(valore -> Math.min(10, valore + 2));
        System.out.println("Prima: " + precedente + ", dopo: " + posti.get());

        int nuovo = posti.updateAndGet(valore -> Math.min(10, valore + 2));
        System.out.println("Valore restituito: " + nuovo);
    }
}
```

Il programma stampa `Prima: 7, dopo: 9`, poi `Valore restituito: 10`. Il nome del metodo ci dice quale fotografia riceviamo come risultato, mentre il valore custodito cambia in modo atomico. Consideriamo ora una funzione che chiama `inviaEmail()` prima di restituire il nuovo numero. Se il tentativo di aggiornamento deve ripartire, l'email potrebbe essere inviata due volte per una sola modifica riuscita. Una funzione che registra un pagamento presenta lo stesso errore, con conseguenze più serie. Per separare il calcolo dall'effetto, prima completiamo la decisione sullo stato e poi gestiamo la comunicazione con un protocollo adatto, capace di trattare anche il fallimento fra i due passaggi.

> **Buona pratica – Purezza nel tentativo.** Nel ciclo CAS e nelle funzioni passate a `getAndUpdate`, `updateAndGet`, `getAndAccumulate` o metodi analoghi, il calcolo del candidato deve poter essere ripetuto. Una funzione *pura* restituisce lo stesso risultato a parità di argomenti e non modifica stato osservabile fuori da quel calcolo. Non basta che il risultato numerico finale sia giusto: anche gli effetti del percorso contano.

Un numero progressivo è un'altra domanda

Se dobbiamo assegnare un identificatore progressivo in memoria, `AtomicLong.getAndIncrement()` può restituire a ogni chiamante un valore distinto finché restiamo nell'intervallo numerico concordato. Qui non serve leggere prima, controllare una copia disponibile e poi decrementare: la singola operazione esprime già l'assegnazione. Ma se l'identificatore deve rimanere unico fra più processi o sopravvivere ai riavvii, un `AtomicLong` in una JVM non basta. L'archivio che conserva i dati deve partecipare alla garanzia. Lo stesso nome «contatore» nasconde due problemi diversi: assegnare un numero locale e coordinare un'identità persistente.

Anche `AtomicBoolean` va letto in questo modo. Può rappresentare un passaggio `false → true` tentato da molti thread, per esempio quando un'azione di inizializzazione deve essere reclamata una sola volta. `compareAndSet(false, true)` dice chi ha ottenuto il diritto di provare. Non dice che l'inizializzazione sia già terminata, né che sia riuscita: se gli altri thread hanno bisogno del risultato devono poter attendere e osservare successo o errore con un protocollo più ricco, come un `Future`. Usare un booleano chiamato `pronto` immediatamente dopo averlo posto a `true` potrebbe pubblicare una promessa prematura.

Cambiamo domanda: quante visualizzazioni abbiamo avuto?

La biblioteca vuole conoscere quante volte una pagina è stata visitata. Ogni richiesta aggiunge uno. Il totale verrà mostrato in un rapporto dopo la conclusione delle attività, oppure letto durante il lavoro come indicazione approssimativa. Qui non stiamo decidendo se assegnare l'ultima copia in base a una fotografia esatta. Sappiamo già usare `AtomicLong` per un conteggio corretto, ma se molti thread aggiornano lo stesso punto della memoria possono contendersi quell'unico valore. Il problema cambia ancora: la domanda è se una struttura specializzata possa distribuire meglio gli aggiornamenti.

`LongAdder` mantiene un totale tramite più componenti interni e li combina quando chiediamo `sum()`. Sotto forte contesa può offrire un throughput migliore rispetto a `AtomicLong`, al prezzo di più memoria. Il verbo *può* conta: con pochi thread, poche modifiche o una particolare piattaforma il guadagno può essere nullo o negativo. La figura 20A.2 aiuta a vedere il compromesso: un solo valore facilita una lettura puntuale; più contributi riducono la concentrazione degli aggiornamenti ma richiedono una somma.

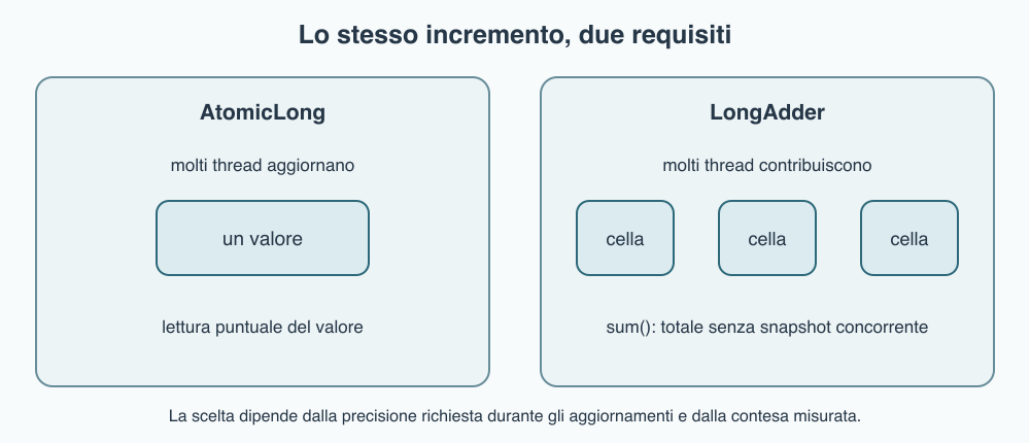


Figura 20A.2 – Un adder raccoglie contributi distribuiti: `sum()` *durante gli aggiornamenti non è una fotografia atomica del totale.*

Nel programma `StatisticheAccessi` otto lavoratori incrementano mille volte ciascuno. Il thread principale chiama `sum()` **dopo** le otto `join`, quando gli aggiornamenti sono conclusi. Per questo il totale atteso è esattamente 8.000. Se invece un altro thread chiamasse `sum()` mentre i lavoratori stanno ancora aggiornando, il numero sarebbe utile per monitorare un andamento, ma non sarebbe uno snapshot atomico da usare per assegnare l'ultima copia.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.LongAdder;

public class StatisticheAccessi {
    public static void main(String[] args) throws InterruptedException {
        LongAdder visualizzazioni = new LongAdder();
        List<Thread> lavoratori = new ArrayList<>();
        for (int i = 0; i < 8; i++) {
            Thread lavoratore = Thread.ofVirtual().start(() -> {
                for (int j = 0; j < 1_000; j++) {
                    visualizzazioni.increment();
                }
            });
            lavoratori.add(lavoratore);
        }

        for (Thread lavoratore : lavoratori) {
            lavoratore.join();
        }
        long totale = visualizzazioni.sum();
        if (totale != 8_000) {
            throw new AssertionError("Visualizzazioni perse: " + totale);
        }
        System.out.println("Visualizzazioni concluse: " + totale);
    }
}
```

`DoubleAdder` offre un'idea analoga per valori `double`; bisogna ricordare che i calcoli in virgola mobile hanno già proprie regole di arrotondamento e che un ordine diverso delle somme può produrre piccole differenze. `LongAccumulator` e `DoubleAccumulator` permettono di definire una funzione di accumulo e un elemento identità: per esempio possiamo voler osservare il massimo di una serie di durate. La funzione deve essere adatta a combinare contributi indipendenti; una sottrazione arbitraria non lo è, perché cambiare ordine e raggruppamento cambia il risultato. Queste API meritano spazio quando il requisito è davvero statistico e un confronto sul carico reale mostra un beneficio. Non usiamo `LongAdder` per un numero di serie che ogni chiamante deve ricevere in modo univoco: `sum()` non offre quella prenotazione puntuale.

> **Approfondimento – Come misurare senza raccontarsi una storia.** Un singolo `System.nanoTime()` attorno a un ciclo in `main` non dimostra che una classe sia generalmente più veloce. Il carico deve dichiarare quanti thread scrivono, quanto spesso si legge, quale rapporto esiste fra lavoro utile e aggiornamento e su quale JDK e macchina si misura. Occorre considerare riscaldamento della JVM e variazione fra esecuzioni. Prima verifichiamo l'invariante; poi misuriamo il throughput, cioè il numero di operazioni completate in un intervallo, e il tempo delle operazioni quando questo interessa all'utente.

Due campi che devono restare d'accordo

Finora lo stato importante stava in un solo `int`. Supponiamo ora che una schermata mostri insieme «copie disponibili» e «copie prenotate». In ogni fotografia coerente la loro somma deve essere cinque. Due `AtomicInteger` separati impedirebbero gli aggiornamenti persi sui singoli campi, ma non renderebbero atomica la coppia: un lettore potrebbe osservare il primo dopo il decremento e il secondo prima dell'incremento. Vedrebbe una somma di quattro, che non rappresenta nessuno stato completo della prenotazione.

Una prima soluzione è usare un lock comune per aggiornare e leggere entrambi i campi insieme. Quando lo stato è piccolo, immutabile e interamente in memoria, possiamo anche creare una nuova fotografia con i due valori e pubblicarla sostituendo **un solo riferimento**. `AtomicReference<Stato>` non rende magicamente atomici i campi dell'oggetto: rende atomica la sostituzione del riferimento. Funziona qui perché ogni `Stato` non cambia dopo la costruzione e tutti i lettori osservano la coppia prendendo un unico riferimento. Il record del programma controlla l'invariante nel costruttore, così una fotografia incompleta non può essere costruita con quei due valori.

La figura 20A.3 mostra la differenza. Con due contenitori indipendenti il lettore può attraversare una fase intermedia; con un riferimento alla fotografia immutabile vede «5, 0» oppure «4, 1». La sostituzione è un solo punto di pubblicazione.



Figura 20A.3 – Un riferimento atomico a uno stato immutabile pubblica insieme i valori collegati, a condizione che tutti i lettori usino la stessa fotografia.

Il programma `StatoImmutabileAtomico` usa venti richieste per le cinque copie. Non memorizza i nomi dei lettori: sviluppa soltanto l'invariante che lega i due numeri. Anche qui il metodo crea un nuovo stato a partire da quello osservato e tenta `compareAndSet`. Se il riferimento è cambiato, rilegge *entrambi* i numeri dalla nuova fotografia e ricostruisce il candidato. Dopo le `join`, il programma stampa `Disponibili: 0, prenotati: 5`.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicReference;

public class StatoImmutabileAtomico {
    record Stato(int disponibili, int prenotati) {
        Stato {
            if (disponibili < 0 || prenotati < 0 || disponibili + prenotati != 5) {
                throw new IllegalArgumentException("Stato non valido");
            }
        }
    }

    private final AtomicReference<Stato> stato =
            new AtomicReference<>(new Stato(5, 0));


    public boolean prenota() {
        while (true) {
            Stato prima = stato.get();
            if (prima.disponibili() == 0) {
                return false;
            }
            Stato dopo = new Stato(prima.disponibili() - 1, prima.prenotati() + 1);
            if (stato.compareAndSet(prima, dopo)) {
                return true;
            }
        }
    }

    public Stato fotografia() {
        return stato.get();
    }


    public static void main(String[] args) throws InterruptedException {
        StatoImmutabileAtomico catalogo = new StatoImmutabileAtomico();
        List<Thread> lettori = new ArrayList<>();
        for (int i = 0; i < 20; i++) {
            lettori.add(Thread.ofVirtual().start(catalogo::prenota));
        }
        for (Thread lettore : lettori) {
            lettore.join();
        }
        Stato risultato = catalogo.fotografia();
        if (risultato.disponibili() != 0 || risultato.prenotati() != 5) {
            throw new AssertionError("Invariante dello stato violata");
        }
        System.out.println("Disponibili: " + risultato.disponibili()
                + ", prenotati: " + risultato.prenotati());


    }
}
```

Per i riferimenti, `compareAndSet(prima, dopo)` confronta il riferimento attuale con `prima` per **identità**, non chiama `equals()` per confrontare il contenuto. Il nostro codice riprova sul riferimento appena letto, quindi la distinzione non rompe l'esempio; è però importante quando oggetti distinti rappresentano valori uguali.

L'immutabilità deve essere reale: un record che contiene una `List` modificabile non diventa automaticamente immutabile. Dovremmo fare una copia difensiva e impedire modifiche successive agli elementi rilevanti, oppure usare un modello di stato differente.

> **Attenzione – Pubblicare uno stato non completa una transazione.** Se dopo il CAS dobbiamo scrivere un file, aggiornare un database o inviare una conferma, può verificarsi un errore fra i due passi. La fotografia in memoria resta coerente, ma non rappresenta necessariamente ciò che è stato salvato. Un database condiviso da più processi richiede una transazione o un'altra forma di coordinamento nell'archivio. Il lock di una sola JVM e `AtomicReference` non estendono la loro garanzia oltre la memoria di quel processo.

La storia che il valore finale non racconta: ABA

Immaginiamo che un thread legga il riferimento A e prepari una modifica. Intanto un altro thread sostituisce A con B e poi rimette **lo stesso oggetto A**. Il primo thread trova ancora A e il suo CAS può riuscire. Per una semplice regola «il valore corrente è A?» ciò può andare bene. Per una struttura dati in cui conta che nessuno abbia toccato il nodo nel frattempo, è un problema: la storia A → B → A non è equivalente a «non è successo nulla». Lo si chiama problema *ABA*.

`AtomicStampedReference` mantiene un riferimento e un numero di versione aggiornabili insieme. Il thread può allora controllare non solo che il riferimento sia A, ma anche che la versione sia ancora quella osservata. Questo non trasforma il numero di versione in una soluzione universale: bisogna progettarne incremento e possibile ritorno dopo overflow. Nel percorso principale non ci serve una struttura dati senza lock; il box serve a non promettere che CAS distingua da solo tutte le storie possibili.

Quale strumento scelgo?

Una scelta affidabile comincia dalla proprietà richiesta, non dall'elenco delle classi disponibili. Se una biblioteca deve impedire la sesta prenotazione, occorre una decisione esatta su uno stato condiviso. Se deve solo mostrare un contatore di visualizzazioni aggiornato spesso, una lettura approssimativa durante il lavoro può essere accettabile. Se deve confermare una prenotazione persistente insieme al nome del lettore, la regola coinvolge più dati e forse più processi. La stessa istruzione `increment()` non rende equivalenti questi tre problemi.

Proprietà richiesta	Strumento da valutare	Confine da verificare
Aggiornare un solo valore con decisione esatta	`AtomicInteger`, `AtomicLong` o `AtomicBoolean`	Tutta la decisione sta nell'operazione atomica o nel ciclo CAS.
Pubblicare una fotografia immutabile di più valori in memoria	`AtomicReference<Stato>`	Tutti i lettori usano la stessa fotografia; gli oggetti non cambiano dopo la pubblicazione.

Proprietà richiesta	Strumento da valutare	Confine da verificare
Contare molti eventi per una statistica	`LongAdder` o `DoubleAdder`	Una lettura concorrente non deve servire come snapshot atomico; il beneficio va misurato.
Proteggere una regola su più oggetti mutabili	`synchronized` o un lock esplicito	Ogni accesso rilevante usa lo stesso protocollo e la sezione critica ha il confine giusto.
Attendere disponibilità o limitare accessi	`BlockingQueue`, `Semaphore` o un coordinamento equivalente	È definito quando attendere, svegliarsi e gestire interruzione o chiusura.
Salvare insieme dati che devono sopravvivere o coordinare più processi	Transazione dell'archivio	La garanzia comprende errore, riavvio e altri processi secondo il contratto dell'archivio.

Questa tabella non è una gara in cui vince sempre lo strumento con il nome più recente. Un lock breve può essere la soluzione più chiara per una regola che attraversa più campi. Un `AtomicInteger` può esprimere con precisione una sola variabile. Un adder può aiutare un contatore molto conteso, ma proprio la sua struttura distribuita cambia ciò che possiamo chiedere a `sum()` mentre arrivano aggiornamenti. Sotto forte contesa anche un ciclo CAS può riprovare molte volte e consumare CPU: misurare e capire il carico fa parte della scelta.

Nel capitolo 25 incontriamo `Files.move` con `ATOMIC_MOVE`. L'aggettivo è lo stesso, ma riguarda un altro confine: la pubblicazione di un file agli osservatori del file system, nei limiti del provider. Non significa che un `AtomicInteger` renda atomico il file, né che uno spostamento atomico garantisca da solo la durata dei dati dopo uno spegnimento. Nel capitolo 27, un client HTTP può riprovare una richiesta dopo un timeout; il CAS locale non rende automaticamente idempotente quella richiesta remota. Per riconoscere il problema giusto, bisogna sempre chiedere: *chi osserva lo stato, dove vive e quale insieme di cambiamenti deve apparire indivisibile?*

Strumenti più vicini all'hardware: un approfondimento

Il package contiene anche array atomici, utili quando ogni posizione ha un proprio aggiornamento atomico, e updater per particolari campi `volatile`. `AtomicMarkableReference` associa a un riferimento un segno booleano; `AtomicStampedReference`, già incontrato, vi associa una versione intera. `VarHandle`, nel package `java.lang.invoke`, permette operazioni su variabili con diverse modalità di accesso e ordine della memoria. Questi strumenti sono utili quando si progettano strutture concorrenti o interoperabilità specializzata; aumentano però il numero di contratti da capire. Il percorso di base resta sulle operazioni ordinarie delle classi atomiche e sui lock, perché già consentono di risolvere i problemi che abbiamo costruito. Ottimizzare l'ordine della memoria prima di aver dimostrato il bisogno non rende un programma più comprensibile.

> **Versione Java – API stabili.** Il toolkit atomico è presente nella piattaforma da molto prima di Java 25; le classi usate nei programmi del capitolo fanno parte di `java.base` e non richiedono `--enable-preview`. `VarHandle` è anch'esso un'API stabile, ma lo presentiamo come approfondimento perché il suo uso corretto richiede un modello più dettagliato degli effetti di memoria.

Proviamo a trasferire il mattone

Partiamo dalle cinque copie e cambiamo una condizione alla volta. Se ogni lettore deve ricevere il numero della propria prenotazione, il booleano restituito da `prenota()` non basta: occorre decidere se il numero identifica una posizione fra le cinque copie, un ordine temporale o un record persistente. Nel primo caso possiamo far restituire dal CAS riuscito il numero associato allo stato precedente, documentando se conta da zero o da uno. Nel secondo caso dobbiamo anche definire l'ordine desiderato e che cosa succede dopo il riavvio. Il lettore ha capito il modello quando riesce a formulare queste domande prima di cambiare il tipo di ritorno del metodo.

Se invece trasformiamo le prenotazioni in *visualizzazioni della scheda del libro*, non esiste più il limite di cinque e non dobbiamo respingere il sesto lettore. Diventa ragionevole valutare `LongAdder`: molte richieste aggiungono uno e un rapporto periodico legge il totale. La risposta cambia perché cambia l'invariante, non perché abbiamo imparato una sintassi nuova. Infine, se vogliamo salvare nome del lettore, copia assegnata e conferma in un archivio, torniamo a chiederci quale sistema possa rendere coerente l'insieme persistente. Nessun metodo di `AtomicInteger` da solo attraversa quel confine.

Autoverifica e attività

Prima di guardare i sorgenti, proviamo a rispondere. Due richieste leggono entrambe 1: quali risultati possono ottenere dai rispettivi `compareAndSet(1, 0)` e perché non possono riuscire entrambe? Se un metodo usa `getAndUpdate` con una lambda che invia una notifica, quale evento può essere duplicato? Dopo la fine di tutti i lavoratori, che cosa cambia nella lettura di `LongAdder.sum()` rispetto a una lettura mentre essi stanno ancora scrivendo? Le risposte ragionate e le prove pratiche si trovano nelle attività del capitolo 20A; la verifica non richiede di indovinare l'ordine in cui lo scheduler eseguirà i thread.

La lezione del capitolo è una domanda da portare nel resto del libro: **qual è la più piccola decisione che deve rimanere indivisibile, e quali osservatori devono vederla così?** Quando la risposta riguarda una sola variabile in memoria, una classe atomica può essere uno strumento chiaro. Quando la risposta comprende attesa, più stati mutabili o persistenza, allarghiamo il modello invece di chiedere a un contatore di fare un lavoro che non gli appartiene.

Riferimenti essenziali

Riferimento essenziale – Specifiche Java SE 25. La documentazione Oracle del package `java.util.concurrent.atomic` definisce il toolkit e il suo perimetro. Le schede di `AtomicInteger`, `AtomicReference`, `LongAdder` e `AtomicStampedReference` precisano i contratti usati nel capitolo. La scheda di `VarHandle` documenta le modalità di accesso richiamate nell'approfondimento. Sono specifiche delle API, non misure delle prestazioni della nostra applicazione: i vantaggi di throughput dipendono dal carico e vanno verificati separatamente.

21. Classloader Java: trovare una classe mentre il programma gira

Perché una classe deve essere caricata

Quando compiliamo un sorgente Java, `javac` produce un file `.class` che contiene bytecode. Questo file non è già una classe in esecuzione nella JVM: l'applicazione deve ancora trovare i byte, controllarli e inserirli nel proprio spazio di esecuzione. Il *classloader*, o caricatore di classi, partecipa a questo lavoro. È il componente a cui la JVM può chiedere una definizione quando serve. Il suo compito non si esaurisce nel leggere un file: una definizione può provenire da un modulo, da un JAR o, in applicazioni specializzate, da una fonte gestita da un caricatore personalizzato.

Perché caricare le classi a richiesta invece di prepararle tutte all'avvio? Un programma può contenere funzionalità che non verranno mai usate in una certa esecuzione; caricarle tutte avrebbe un costo inutile. Il caricamento appartiene al tempo di esecuzione. Se nel sorgente compare `new Servizio()`, il compilatore deve conoscere il tipo per verificare il programma, ma non crea l'oggetto e non carica la classe nella JVM durante la compilazione. Sono fasi diverse. Analogamente, `Class.forName(nome)` esprime una richiesta in cui il nome può arrivare da una stringa a tempo di esecuzione; non è l'unico modo in cui Java carica dinamicamente una classe.

> **Definizione – Caricamento, collegamento, inizializzazione.** Caricare significa ottenere una definizione di classe. Il *linking* comprende verifiche e preparazione necessarie alla sua integrazione nella JVM; la risoluzione dei riferimenti può avvenire secondo le regole della VM. Inizializzare significa eseguire l'inizializzazione statica secondo le condizioni previste dal linguaggio. Dire semplicemente «la classe viene caricata» non implica che il suo blocco `static` sia già stato eseguito. Questa distinzione aiuta a interpretare gli esempi di `Class.forName`, i messaggi di errore e i tempi osservati.

I caricatori forniti dalla piattaforma

Il programma completo osserva il caricatore di una classe dell'applicazione, di `java.sql.Driver` e di `java.util.ArrayList`. Con la distribuzione JDK 25 usata per la verifica, la prima è associata al caricatore dell'applicazione, la seconda a quello della piattaforma e la terza al caricatore bootstrap. Nel terzo caso `Class.getClassLoader()` restituisce `null`, che l'esempio stampa come `bootstrap`. Non dobbiamo confondere quel `null` con l'assenza della classe: `ArrayList` esiste, ed è proprio il suo caricatore bootstrap a non essere rappresentato come un normale oggetto `ClassLoader` restituito dal metodo.

```
Applicazione -> jdk.internal.loader.ClassLoaders$AppClassLoader
java.sql.Driver -> jdk.internal.loader.ClassLoaders$PlatformClassLoader
java.util.ArrayList -> bootstrap
```

Queste righe documentano l'osservazione su una JVM concreta, non un formato di output da promettere a tutte le implementazioni. I nomi `jdk.internal.loader...` sono dettagli di implementazione: il codice del lettore non deve dipendere da quelle stringhe. Il contratto pubblico offre `ClassLoader.getSystemClassLoader()` e `ClassLoader.getPlatformClassLoader()`, oltre a `Class.getClassLoader()`. Il caricatore dell'applicazione cerca normalmente classi e risorse del percorso dell'applicazione; quello della piattaforma serve classi della piattaforma; il bootstrap è alla base della gerarchia. Il preciso insieme di classi visibili a ciascuno è influenzato anche dal sistema di moduli introdotto nelle edizioni moderne di Java.

Nei JDK precedenti a Java 9 comparivano un *extension classloader*, cartelle come `jre/lib/ext` e l'archivio `rt.jar`. Quel modello non descrive Java 25: ora incontriamo il *platform classloader* e l'immagine runtime modulare. Se leggiamo documentazione o progetti più vecchi, possiamo riconoscere quei nomi storici, ma non dobbiamo cercare `rt.jar` né configurare un meccanismo di estensioni rimosso. La relazione fra genitori e figli rimane utile; cambiano i componenti concreti che occupano alcuni posti nella gerarchia.

```
            bootstrap
                ↑
            piattaforma
                ↑
            applicazione
                ↑
      eventuale caricatore figlio
```

Figura 21.1 – Relazione di delega tipica. La freccia punta verso il genitore consultato prima dal figlio. Non rappresenta ereditarietà fra le classi caricate. Un caricatore personalizzato può scegliere comportamenti speciali; la figura descrive il modello predefinito che studiamo qui.

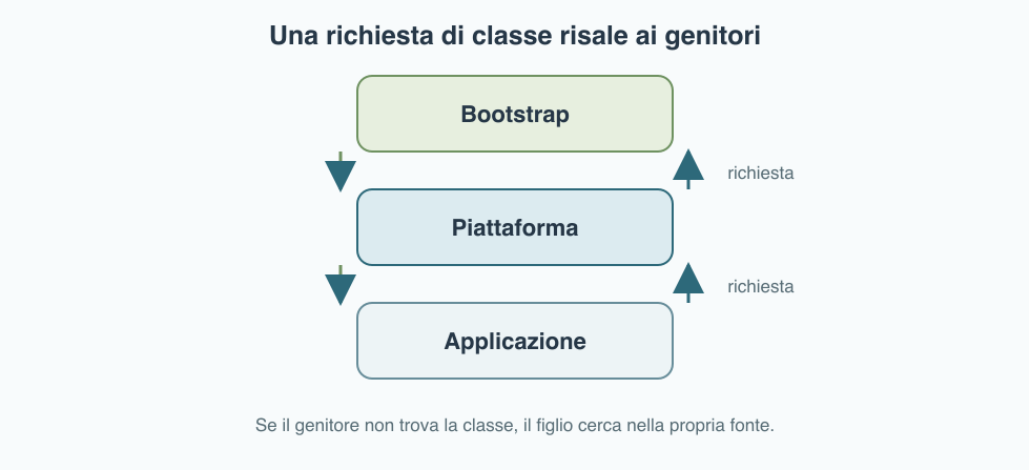


Figura 21.2 – Il percorso della richiesta e della risposta nei caricatori integrati.

La gerarchia di delega non coincide con la gerarchia delle **classi Java** dei caricatori. Per esempio, nell'API pubblica `URLClassLoader` estende `SecureClassLoader`, che estende `ClassLoader`: questa è una relazione fra tipi. Dire invece che il caricatore dell'applicazione consulta come genitore quello della

piattaforma descrive una relazione fra **oggetti caricatori** durante una ricerca. Non possiamo dedurre dal nome *platform* che la sua classe Java estenda la classe concreta del caricatore bootstrap; quest'ultimo non compare neppure come un normale oggetto restituito da `Class.getClassLoader()` per le classi che definisce. La distinzione evita di leggere come un diagramma di ereditarietà una freccia che indica una richiesta di delega.

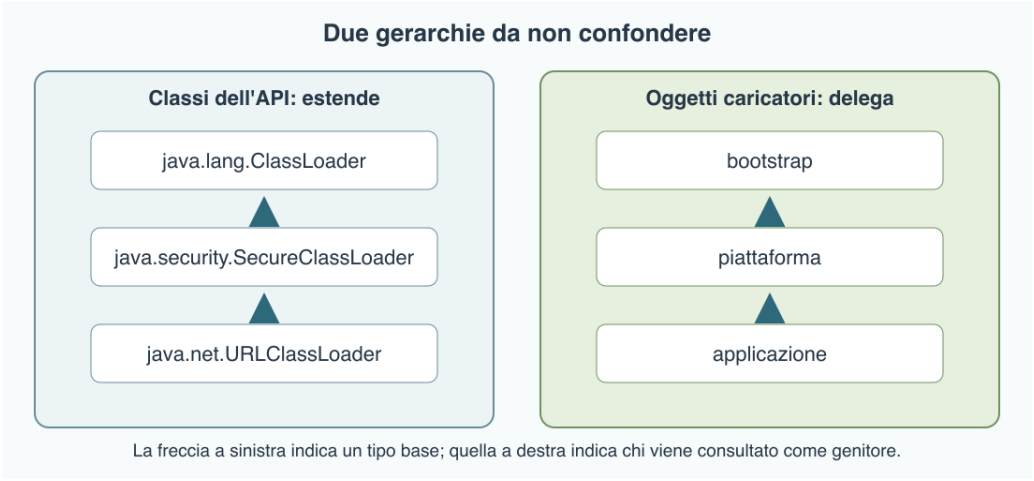


Figura 21.3 – A sinistra un esempio di ereditarietà pubblica verificabile nell'API; a destra la relazione di delega tipica dei caricatori integrati.

La delega, passo dopo passo

Supponiamo che il caricatore dell'applicazione riceva il nome binario `java.util.ArrayList`. Prima controlla se ha già caricato quella definizione. Se non l'ha, la normale implementazione di `loadClass` chiede al genitore; la ricerca sale fino al bootstrap, che può trovare la classe. Il risultato torna lungo la catena. Il caricatore dell'applicazione non definisce una seconda copia di `ArrayList`. Se invece il nome appartiene a una classe dell'applicazione che i genitori non conoscono, il controllo torna al caricatore dell'applicazione, che prova `findClass`. È la logica detta *parent first*.

L'algoritmo pubblico di `ClassLoader.loadClass(String, boolean)` segue tre passaggi essenziali: `findLoadedClass` controlla una definizione già caricata dal caricatore corrente; il genitore o il bootstrap riceve la richiesta; infine `findClass` è il punto in cui il caricatore corrente cerca una definizione propria. L'argomento `resolve` chiede anche la risoluzione tramite `resolveClass` quando è `true`. Non è corretto interpretare `false` come «la classe non sarà caricata»: il metodo può caricarla e definirla ugualmente, rimandando soltanto quell'operazione di risoluzione. Inoltre, codice concorrente può chiedere classi allo stesso caricatore: l'implementazione sincronizza il lavoro con un lock di caricamento appropriato. Il capitolo 20A ha mostrato perché un singolo CAS protegge una sola decisione sullo stato; il caricamento comprende invece ricerca, delega, definizione e risoluzione. Non va sostituito meccanicamente con un contatore atomico.

> **Schema – Una richiesta che risale e ridiscende.** Il figlio controlla ciò che ha già definito; il genitore prova a trovare la classe; se il genitore fallisce, il figlio cerca nei propri byte. La risposta positiva o l'eccezione torna al chiamante. La delega protegge le classi fondamentali e facilita la condivisione di definizioni, ma non garantisce che nel processo esista una sola definizione per ogni nome.

La frase finale è importante. L'identità di una classe nella JVM dipende dal nome binario **e** dal caricatore che l'ha definita. Due caricatori distinti possono definire due classi con lo stesso nome; per la JVM sono tipi diversi. Questa possibilità è utile, per esempio, quando un contenitore isola componenti o versioni di librerie, ma richiede grande attenzione nel passare oggetti da un contesto all'altro. Una `ClassCastException` fra due nomi apparentemente uguali può nascere proprio da questa differenza di caricatori. La delega evita duplicazioni nel caso normale della stessa catena; non rende però unico un nome di classe in tutta la JVM.

Vedere la gerarchia senza dipendere dagli indirizzi

Il programma chiama `getParent()` sul caricatore di sistema e stampa il risultato. Una JVM può rappresentare l'oggetto con il nome di una classe interna e un suffisso simile a un indirizzo; quel suffisso cambia fra esecuzioni e non ha valore didattico. Il controllo significativo è che, nella configurazione comune, il genitore del caricatore di sistema sia quello della piattaforma. `getParent()` può inoltre essere `null` quando si arriva al livello bootstrap. Un test robusto confronterebbe gli oggetti restituiti dalle API pubbliche, non le loro rappresentazioni testuali.

La ricerca delle *risorse* segue un problema vicino ma distinto. `getResource("configurazione.txt")` cerca un elemento sul percorso visibile al caricatore e restituisce un `URL` o `null`; `getResourceAsStream` può offrire direttamente uno stream da chiudere. La risorsa non è per forza un file ordinario del filesystem, soprattutto se si trova in un JAR o in un'immagine modulare. Per questo il percorso va scritto con separatori `/` e interpretato rispetto alla radice di ricerca prevista dall'API, invece di concatenare a mano un percorso assoluto dipendente dal sistema operativo. Se la risorsa è obbligatoria, un risultato `null` deve diventare un errore gestito con un messaggio utile, non una `NullPointerException` nella lettura successiva.

Classe non trovata: eccezione o errore?

`ClassNotFoundException` e `NoClassDefFoundError` segnalano problemi diversi, anche se entrambi possono avere a che fare con una classe non disponibile. `ClassNotFoundException` è un'**eccezione controllata**: un metodo come `Class.forName("esempio.Mancante")` o `loadClass("esempio.Mancante")` riceve un nome e può non trovare la definizione richiesta. Il chiamante deve gestire o dichiarare l'eccezione. Nel programma precedente usiamo `Class.forName("java.util.ArrayList")`, che trova una classe reale e stampa il suo nome; sostituendo una stringa inesistente si può osservare `ClassNotFoundException` in modo controllato.

`NoClassDefFoundError` è invece un `Error` che può comparire quando la JVM ha bisogno di una definizione necessaria all'esecuzione ma non riesce a renderla disponibile. Per vederlo, compiliamo `A` e `B`, poi togliamo `A.class` prima di eseguire

`B`, che usa direttamente `new A()`. Il bytecode di `B` contiene un riferimento ad `A`; all'esecuzione la definizione manca. La diagnostica può includere una `ClassNotFoundException` come causa, ma il fallimento osservato dal codice che usa `B` è `NoClassDefFoundError`. Può esserci anche un errore di inizializzazione precedente che lasci la classe inutilizzabile; quindi non riduciamo il significato del tipo al solo «file cancellato».

Possiamo ripetere il primo caso con un programma breve. La stringa indica una classe che non abbiamo compilato; perciò l'eccezione viene catturata e il programma resta in controllo della situazione:

```
public class ClasseMancante {
    public static void main(String[] args) {
        try {
            Class.forName("esempio.plugin.Assente");
        } catch (ClassNotFoundException e) {
            System.out.println(e.getClass().getSimpleName() + ": " + e.getMessage());
        }
    }
}
```

L'output è `ClassNotFoundException: esempio.plugin.Assente`. Anche `ClassLoader.loadClass` può segnalare la stessa eccezione. Se vogliamo provare separatamente il caricatore di sistema e quello della piattaforma, dobbiamo dare a ciascuna chiamata il proprio `try` o continuare dopo il primo errore: tre richieste nello stesso blocco `try` si fermerebbero alla prima eccezione. Il messaggio mostra il nome richiesto, mentre le righe della traccia dello stack dipendono da JDK e ambiente e non sono un output da memorizzare.

Per il secondo caso usiamo due sorgenti separati, con `B` che usa `A` direttamente:

```
public class A {
    public String messaggio() {
        return "A disponibile";
    }
}
```

```
public class B {
    public static void main(String[] args) {
        System.out.println(new A().messaggio());
    }
}
```

Compilando entrambi, `java -cp build/errori B` stampa `A disponibile`. Dopo avere spostato `A.class` fuori dal percorso di esecuzione, lo stesso comando fallisce con `NoClassDefFoundError: A`. Il sorgente di `A` può rimanere dov'è: durante l'esecuzione la JVM cerca la definizione compilata, non ricompila automaticamente quel sorgente. Per tenere l'esperimento isolato, dalla directory `esempi/C21` possiamo usare questi comandi e ripristinare il file alla fine:

```
mkdir -p build/errori
javac --release 25 -d build/errori errori/A.java errori/B.java
java -cp build/errori B
mv build/errori/A.class build/A.class.spostata
java -cp build/errori B
mv build/A.class.spostata build/errori/A.class
```

Il secondo avvio deve fallire: in questo solo esperimento il suo codice d'uscita diverso da zero è il risultato cercato. La traccia completa può cambiare fra versioni della JVM, mentre la distinzione fra una richiesta per nome non soddisfatta e una definizione necessaria che manca resta il punto da capire.

> **Esercizio guidato – Due tempi diversi.** Si compilino due classi nello stesso percorso di output. Si esegua `B` una prima volta con entrambe presenti, poi si sposti temporaneamente `A.class` fuori da quel percorso e si ripeta l'esecuzione. Si ripristini il file al termine. Il risultato dipende dalla fase di esecuzione, non dal fatto che il sorgente `B.java` sia ancora disponibile. È un esperimento da svolgere in una cartella di prova, senza rimuovere librerie del progetto.

Un'altra differenza utile riguarda l'inizializzazione. La forma a un argomento di `Class.forName` carica e inizializza la classe richiesta secondo il suo contratto. La forma con tre argomenti permette di scegliere il booleano di inizializzazione e il caricatore. `ClassLoader.loadClass` normalmente carica senza richiedere direttamente l'inizializzazione. In seguito, un'operazione che richiede l'uso attivo della classe può inizializzarla. Non è dunque corretto etichettare `Class.forName` come «caricamento dinamico» e `new` come «caricamento statico» se con quest'ultimo si suggerisce un caricamento a tempo di compilazione: entrambi arrivano a una JVM in esecuzione, ma esprimono richieste diverse.

Personalizzare un caricatore senza perdere la delega

Esistono casi in cui le definizioni arrivano da una fonte che il caricatore standard non conosce: un sistema di plugin, un archivio speciale o una generazione controllata di bytecode. La libreria offre la classe astratta `ClassLoader`. Nel caso comune si estende il caricatore e si ridefinisce `findClass(String name)`, lasciando a `loadClass` il comportamento di delega. `findClass` deve trovare i byte corrispondenti al nome richiesto e chiamare `defineClass` con un array valido. Se non trova la definizione, lancia `ClassNotFoundException`. Il risultato della ricerca va controllato, le risorse aperte vanno chiuse e occorre definire quali byte considerare affidabili.

> **Approfondimento – Che cos'è il weaving.** Il *weaving* è la trasformazione del bytecode di una classe per aggiungere o modificare un comportamento, per esempio una misura dei tempi o un controllo all'ingresso di un metodo. Può avvenire dopo la compilazione, modificando i file `.class`, oppure durante il caricamento tramite strumenti adatti. Un caricatore personalizzato può partecipare al secondo percorso, ma il semplice esempio che segue **non modifica** il bytecode: legge una definizione già compilata da una directory diversa. Separare i due meccanismi evita di confondere «trovare byte» e «trasformare byte».

Un caricatore personalizzato che possiamo provare

Facciamo ora un esperimento completo. La classe `CaricatoreDaDirectory` cerca un file `.class` sotto la directory indicata all'avvio. L'altro sorgente, `Messaggio`, viene compilato in una directory separata: non deve trovarsi nel class path dell'applicazione, altrimenti il caricatore padre potrebbe trovarlo prima del nostro `findClass`. Questa separazione è la condizione che permette di osservare davvero il nuovo caricatore.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class CaricatoreDaDirectory extends ClassLoader {
    private final Path directory;

    public CaricatoreDaDirectory(Path directory, ClassLoader parent) {
        super(parent);
        this.directory = directory.toAbsolutePath().normalize();
    }

    @Override
    protected Class<?> findClass(String nome) throws ClassNotFoundException {
        Path file = directory.resolve(nome.replace('.', '/') + ".class").normalize();
        if (!file.startsWith(directory)) {

            throw new ClassNotFoundException(nome);
        }
        try {
            byte[] bytecode = Files.readAllBytes(file);
            return defineClass(nome, bytecode, 0, bytecode.length);
        } catch (IOException e) {
            throw new ClassNotFoundException(nome, e);
        }
    }

    public static void main(String[] args) throws ReflectiveOperationException {
        if (args.length != 1) {
            throw new IllegalArgumentException("Indicare la directory del plugin");
        }
        ClassLoader caricatore = new CaricatoreDaDirectory(
                Path.of(args[0]), CaricatoreDaDirectory.class.getClassLoader());
```

```
        Class<?> tipo = Class.forName("esempio.plugin.Messaggio", true, caricatore);
        Object istanza = tipo.getDeclaredConstructor().newInstance();
        System.out.println(tipo.getMethod("testo").invoke(istanza));
        System.out.println(tipo.getClassLoader().getClass().getSimpleName());
    }
}
```

Il nome binario `esempio.plugin.Messaggio` diventa il percorso `esempio/plugin/Messaggio.class` sotto la directory ricevuta dal costruttore. `Files.readAllBytes` chiude internamente il file dopo la lettura; se il file non esiste o la lettura fallisce, `findClass` conserva la causa dentro `ClassNotFoundException`. Il controllo con `normalize` evita che il percorso calcolato esca banalmente dalla directory scelta, ma non trasforma questa classe in una barriera di sicurezza per input ostili. `defineClass` verifica che i byte rappresentino una classe compatibile con il nome richiesto; gli errori di formato non vengono finti come «file mancante».

Il piccolo plugin contiene un metodo pubblico e non ha dipendenze applicative:

```
package esempio.plugin;

public class Messaggio {
    public String testo() {
        return "ciao dal plugin";
    }
}
```

Dalla directory `esempi/C21` possiamo compilare e avviare così, con due percorsi di output distinti:

```
mkdir -p build/app build/plugin
javac --release 25 -d build/app CaricatoreDaDirectory.java
javac --release 25 -d build/plugin plugin/esempio/plugin/Messaggio.java
java -cp build/app CaricatoreDaDirectory build/plugin
```

Il risultato atteso è `ciao dal plugin`, seguito da `CaricatoreDaDirectory`. La prima riga viene dal metodo del plugin; la seconda mostra il tipo del caricatore che ha definito quella classe. Se compilassimo `Messaggio` anche in `build/app`, il modello *parent first* potrebbe farla definire dal padre, e la seconda riga cambierebbe. L'esempio rende quindi osservabili sia il caricamento da byte sia l'effetto della delega.

`defineClass` non è un metodo di ricerca. Trasforma byte di una classe valida in un oggetto `Class<?>` associato al caricatore che lo chiama; può fallire se il formato è invalido o se sono violati vincoli del runtime. `findLoadedClass` serve a evitare di definire nuovamente, nello stesso caricatore, una classe già definita; `getParent` rende osservabile la catena di delega. Per spiegare la relazione fra questi metodi, non serve copiare l'implementazione interna del JDK riga per riga: basta seguire la richiesta dal nome ai byte e poi all'oggetto `Class`, distinguendo il contratto pubblico dai dettagli privati dell'implementazione.

Metodo	Domanda a cui risponde	Risultato rilevante
`loadClass(nome)`	Quale classe visibile corrisponde al nome?	`Class<?>` oppure `ClassNotFoundException`
`findLoadedClass(nome)`	Questo caricatore l'ha già definita?	`Class<?>` oppure `null`
`findClass(nome)`	Questo caricatore sa reperire i byte?	`Class<?>` oppure `ClassNotFoundException`
`defineClass(nome, byte[], ...)`	Questi byte definiscono una classe valida qui?	un oggetto `Class<?>` oppure un errore di definizione
`getParent()`	A chi delega normalmente questo caricatore?	un `ClassLoader` oppure `null` al livello bootstrap
`getResource(nome)`	Quale risorsa visibile ha quel nome?	un `URL` oppure `null`

La tabella distingue l'azione di ciascun metodo dalla risposta possibile. `findLoadedClass` non cerca sul filesystem: esamina ciò che il caricatore ha già definito. `findClass`, nell'implementazione di base, non inventa una sorgente di byte: se vogliamo leggere da un archivio particolare, dobbiamo fornire il comportamento. `defineClass` è protetto e finale nelle forme di uso comune: il caricatore personalizzato lo invoca, non ne riscrive il significato. Anche `getResource` va trattato come una ricerca facoltativa, perché un `URL` nullo è una risposta prevista.

Un'applicazione modulare aggiunge un altro livello al ragionamento. Il sistema di moduli stabilisce quali pacchetti un modulo esporta e quali moduli sono leggibili da un altro; il caricatore stabilisce da quale definizione proviene una classe. Le due domande interagiscono ma non si sostituiscono. Trovare i byte di una classe non implica poter usare qualsiasi suo tipo pubblico da qualsiasi modulo, né poter accedere tramite reflection a ogni membro privato. Questa distinzione sarà utile quando, nel capitolo sulla reflection, incontreremo `opens` e `InaccessibleObjectException`.

Pensiamo a un plugin che implementa un'interfaccia `Servizio` condivisa. Se l'applicazione e il plugin vedono la *stessa* definizione dell'interfaccia grazie a un caricatore comune, l'applicazione può ricevere l'istanza come `Servizio`. Se invece il plugin carica una propria copia della classe dell'interfaccia, con lo stesso nome ma un caricatore differente, il cast può fallire. Prima di modificare la politica di delega conviene quindi decidere quali API sono comuni e quali classi devono restare isolate. La figura a frecce non è solo teoria: spiega una regola concreta per progettare confini fra componenti.

> **Approfondimento – Isolamento e fiducia.** Caricare byte ricevuti da una fonte esterna non rende quel codice automaticamente sicuro. Un caricatore personalizzato è uno strumento di risoluzione e di identità dei tipi, non una sandbox generale. Anche una scelta di caricamento *child first*, adottata talvolta per isolare plugin, richiede una progettazione accurata: non deve permettere di sostituire classi della piattaforma o rompere le API condivise. Nel nostro esempio manteniamo la delega al genitore; i byte del plugin vanno comunque considerati codice eseguibile con i permessi del processo.

Dalla classe caricata alla reflection

Una volta ottenuto un oggetto `Class<?>`, il programma può chiedere informazioni sul tipo e, quando il contratto di accesso lo consente, usare la Reflection API. Il caricatore risponde alla domanda «quale definizione di classe sto usando?»; la reflection risponde a domande come «quali costruttori, metodi o annotazioni espone?». La differenza fra identità dei tipi e semplice uguaglianza dei nomi rimane essenziale. Se un plugin definisce un'interfaccia comune con un caricatore incompatibile con quello dell'applicazione, i due lati potrebbero vedere nomi uguali ma tipi diversi. Prima di tornare all'oggetto `Class` nel capitolo 24, il capitolo 23 propone un approfondimento facoltativo sulla Vector API: affronta un problema indipendente e può essere letto anche dopo la reflection.

Per verificare

Spiega perché `ArrayList.class.getClassLoader()` restituisce `null` nell'esempio, quale passaggio del modello predefinito permette a una classe dell'applicazione di essere trovata dopo il fallimento del genitore, e perché due caricatori possono produrre tipi distinti con lo stesso nome. Verifica la seconda risposta nel programma con caricatore da directory, mantenendo il plugin fuori dal class path dell'applicazione: se lo inserisci nel percorso sbagliato, quale caricatore lo trova? Se sai prevedere e spiegare la differenza, la figura della delega è diventata uno strumento di ragionamento anziché un disegno da ricordare a memoria.

23. Approfondimento: Vector API, dati elaborati per gruppi

L'API vettoriale di questo capitolo affronta il calcolo su più elementi alla volta. Il nome «vettore» e l'idea di parallelismo possono ricordare i thread, ma il problema è diverso da quello del capitolo 20A: una somma per corsie non rende atomico un aggiornamento condiviso fra thread. Prima di scegliere questo strumento chiediamo se il lavoro riguarda dati numerici indipendenti, non una decisione concorrente su un unico stato.

Questo è un approfondimento autonomo fra il capitolo sui classloader e quello sulla reflection. Se vuoi seguire subito il filo «caricamento di un tipo → ispezione del tipo», puoi passare al capitolo 24 e tornare qui in seguito: i vettori non sono un prerequisito della reflection.

> **Versione Java – API incubator.** La Vector API appartiene al modulo `jdk.incubator.vector` in Java 25: richiede `--add-modules jdk.incubator.vector` e può cambiare o essere rimossa in una versione futura. Non è una normale API stabile di Java SE 25. Gli esempi di questo approfondimento seguono la documentazione del modulo nel JDK 25.

Perché parlare di vettori

In molti problemi scientifici, grafici o di elaborazione dei segnali si esegue la stessa operazione su molti numeri indipendenti. Per sommare due array `float[]`, un ciclo ordinario legge una coppia di elementi alla volta, somma e scrive il risultato. Una CPU può disporre di istruzioni SIMD, capaci di applicare una stessa operazione a più *corsie* di dati in una istruzione. L'ottimizzazione automatica del compilatore può talvolta sfruttarle, ma non tutti i cicli hanno una forma facilmente riconoscibile. La Vector API offre un modo esplicito per descrivere operazioni su gruppi di valori, lasciando alla JVM il compito di adattarle all'hardware disponibile.

> **Definizione – Vettore e corsia.** Un vettore in questa API è un gruppo di valori dello stesso tipo primitivo, per esempio più `float`. Ogni posizione del gruppo è una *corsia*. La somma di due vettori somma le corsie corrispondenti: non calcola la somma di tutti gli elementi di un solo vettore. Questa distinzione evita di confondere un'operazione elemento per elemento con una riduzione.

Nel programma completo, `FloatVector.SPECIES_PREFERRED` sceglie una specie adatta alla piattaforma per valori `float`. La specie descrive, fra l'altro, quante corsie contiene il vettore. Il numero stampato da `SPECIE.length()` può cambiare fra macchine e JVM; perciò l'esempio non promette una larghezza fissa. Il risultato numerico, invece, è prevedibile per i piccoli valori usati: `[11.0, 22.0, 33.0, 44.0, 55.0]`.

```
FloatVector a = FloatVector.fromArray(SPECIE, sinistra, i);
FloatVector b = FloatVector.fromArray(SPECIE, destra, i);
a.add(b).intoArray(risultato, i);
```

Le prime due righe caricano gruppi di numeri dagli array, a partire dall'indice `i`. La terza somma corsia per corsia e scrive il gruppo nel risultato. Il ciclo avanza di `SPECIE.length()` elementi. Se la lunghezza dell'array non è multipla delle corsie, resta una *coda*: il programma la elabora con un normale ciclo scalare. `SPECIE.loopBound(sinistra.length)` calcola il limite fino al quale i gruppi completi possono essere letti senza superare l'array. Per questo l'esempio funziona anche quando la macchina usa più corsie dei cinque elementi disponibili: in quel caso il ciclo vettoriale può non eseguire alcuna iterazione e la coda scalare fa tutto il lavoro.

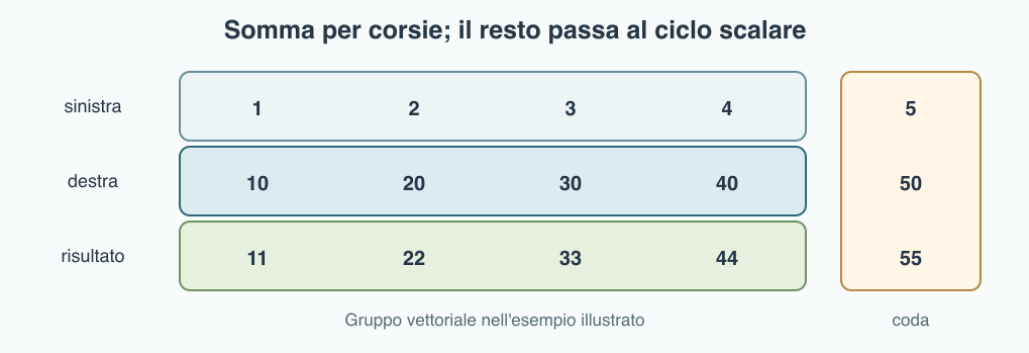


Figura 23.1 – Esempio illustrato con quattro corsie. La larghezza effettiva di `SPECIES_PREFERRED` dipende dalla piattaforma.

Prima di iniziare, il metodo controlla che i due array abbiano la stessa lunghezza. Senza quel controllo, un accesso oltre i limiti potrebbe produrre un errore a metà elaborazione e lasciare un risultato parziale. La validazione stabilisce chiaramente il contratto: una somma elemento per elemento richiede due sequenze confrontabili. Non occorre introdurre un comportamento implicito per array di lunghezze diverse.

Compilare e leggere la prova

Il file si compila con `javac --release 25 --add-modules jdk.incubator.vector SommaVector.java` e si esegue con `java --add-modules jdk.incubator.vector SommaVector`. Il JDK emette un avviso sull'uso del modulo incubator: è atteso e conferma la natura sperimentale dell'API. Se si omette l'opzione, il modulo potrebbe non essere disponibile alla compilazione o all'esecuzione. Gli esempi dei capitoli stabili continuano a compilare senza questa opzione.

La presenza di un ciclo vettoriale non prova da sola un miglioramento di prestazioni. La misura dipende da dimensione dei dati, hardware, configurazione della JVM, riscaldamento del compilatore JIT e costo del trasferimento dei dati. Per cinque numeri il programma è una dimostrazione di semantica, non un benchmark. Anche un confronto con un ciclo scalare richiede prove ripetute e un metodo di misura adatto; cronometrare una singola esecuzione da `main` confonderebbe avvio della JVM, compilazione e lavoro vero.

Un esercizio utile è cambiare la lunghezza degli array in modo che restino zero, uno o molti elementi dopo l'ultimo vettore completo. Il lettore può verificare che il risultato non perda mai la coda. Solo dopo aver capito questo confine ha senso esplorare le maschere della Vector API, che permettono di esprimere operazioni su corsie selezionate. Restano però parte della stessa API incubator: ogni esempio aggiuntivo dovrà essere verificato sulla versione di JDK usata dal volume.

Quando la lunghezza non coincide con le corsie

Il primo programma usa due cicli: uno per i gruppi completi, uno per il resto. È una soluzione leggibile e utile per iniziare, perché il ciclo scalare rende evidente che nessun elemento deve essere perso. La Vector API offre anche le *maschere*. Una maschera ha una condizione booleana per ciascuna corsia: indica quali corsie sono valide in una certa operazione. Nel secondo programma, `SPECIE.indexInRange(i, sinistra.length)` costruisce la maschera delle posizioni che ricadono ancora nell'array. Le forme mascherate di `fromArray` e `intoArray` leggono e scrivono soltanto le corsie selezionate.

```
VectorMask<Float> valide = SPECIE.indexInRange(i, sinistra.length);
FloatVector a = FloatVector.fromArray(SPECIE, sinistra, i, valide);
FloatVector b = FloatVector.fromArray(SPECIE, destra, i, valide);
a.add(b).intoArray(risultato, i, valide);
```

Per capire queste quattro righe, prendiamo la figura con quattro corsie e cinque numeri. Il primo giro usa gli indici da zero a tre: tutte le corsie sono valide. Il secondo giro parte dall'indice quattro: solo la prima corsia corrisponde al quinto elemento; le altre tre cadrebbero oltre la fine dell'array e la maschera le esclude. Il risultato scritto contiene comunque cinque valori, non otto. Se la piattaforma preferisce una larghezza diversa, lo stesso metodo calcola la maschera in base alla lunghezza effettiva della specie. È questo il motivo per cui il programma non codifica il numero quattro come limite del ciclo.

La versione mascherata non è automaticamente più veloce di quella con il ciclo scalare finale. La forma scelta può influire sulla generazione del codice macchina e il risultato dipende dai dati e dall'hardware. Qui la maschera insegna un meccanismo dell'API: permette di esprimere «opera solo su queste corsie» e di gestire in modo uniforme il gruppo parziale. Il primo programma, con la coda scalare, può rimanere preferibile come spiegazione iniziale. Entrambi vanno verificati con array vuoti, più corti di una specie, lunghi esattamente una specie e lunghi una specie più un elemento.

> **Approfondimento – Maschera e condizione Java.** Una `VectorMask<Float>` non è un singolo `boolean` che decide se eseguire tutto il ciclo. Contiene una scelta per ciascuna corsia. Se il gruppo ha quattro corsie e soltanto la prima è ancora nell'array, la maschera rappresenta concettualmente `vero, falso, falso, falso`. La forma esatta dell'oggetto e il numero di corsie dipendono dalla specie. Questa distinzione evita di pensare alla maschera come a un semplice `if` attorno al caricamento dell'intero vettore.

Specie, tipo degli elementi e portabilità

`VectorSpecies<Float>` descrive la combinazione di tipo degli elementi e forma del vettore. `Float` nel parametro generico identifica il tipo delle corsie nell'API, mentre i dati degli array restano `float` primitivi. Il metodo `length()` restituisce il numero delle corsie; `loopBound(length)` calcola il maggiore multiplo di quel numero che non supera la lunghezza passata. Se una specie ha quattro corsie e l'array ne contiene undici, il limite è otto: i gruppi completi partono dagli indici zero e quattro; tre elementi rimangono per il ciclo scalare. Se la specie ne ha otto, il limite resta otto, ma c'è un solo gruppo completo. La formula è semplice, ma va collegata all'indice reale del ciclo per evitare accessi fuori limite.

La specie *preferred* lascia alla piattaforma una scelta. Questo favorisce la portabilità del sorgente, non l'identità del comportamento prestazionale. Il programma non deve promettere che `SPECIE.length()` valga un numero specifico. Nei test si controlla il risultato dell'operazione e, se si vuole, si stampa la larghezza come informazione diagnostica. Una figura didattica può disegnare quattro corsie per rendere visibile il ragionamento, purché la didascalia dica che è un esempio illustrato e non il vincolo dell'API.

Un algoritmo vettoriale deve anche considerare il tipo di dato. La somma di `float` segue le regole dei numeri in virgola mobile: valori molto grandi, valori molto piccoli, infiniti e `NaN` non si comportano come interi matematici esatti. Nel nostro esempio usiamo numeri piccoli che hanno un risultato intuitivo. Se si trasformasse il programma in una riduzione, per esempio la somma di tutti gli elementi, l'ordine delle addizioni potrebbe cambiare e con esso alcuni risultati arrotondati. Prima di promettere equivalenza bit per bit con un ciclo scalare, occorrerebbe stabilire il contratto numerico richiesto e controllare l'operazione concreta.

Dall'esempio al problema reale

La somma elemento per elemento è un primo mattone. Un'applicazione reale potrebbe elaborare campioni audio, trasformare canali di un'immagine o applicare una funzione a grandi blocchi di dati numerici. Il vantaggio potenziale nasce quando lo stesso calcolo semplice si ripete su abbastanza elementi da rendere importante il lavoro sulle corsie. Se ogni elemento richiede una decisione complessa o l'accesso irregolare a oggetti sparsi, il codice vettoriale può perdere chiarezza senza offrire un beneficio. Prima si individua un ciclo che pesa davvero nel tempo totale, poi si prova una forma vettoriale e la si misura.

La misura deve confrontare risultati corretti e condizioni comparabili. Un benchmark che include nel tempo la creazione degli array per una variante e la esclude dall'altra non confronta lo stesso lavoro. Un'esecuzione singola della JVM include avvio e attività del compilatore JIT. Dimensione degli array, riscaldamento, memoria, architettura e versione del JDK vanno registrati. Se i dati sono già in cache in una prova e non nell'altra, la differenza può dipendere dalla memoria anziché dalle istruzioni SIMD. La Vector API dà un linguaggio per esprimere il calcolo; la prova di vantaggio appartiene a un esperimento misurato.

Questo approfondimento permette di sperimentare SIMD e maschere sapendo quale dipendenza si introduce. Se il modulo cambierà in una versione successiva, occorrerà verificare di nuovo esempi e comandi prima di riusarli. Per ora il punto da portare con sé è il metodo: partire da un calcolo concreto, capire che cosa fa

ogni corsia, controllare anche la coda dell'array e misurare soltanto dopo avere verificato la correttezza del risultato.

Per verificare

Esegui i due programmi con array di lunghezza zero, uno, cinque e dieci. Per ogni lunghezza indica quanti gruppi completi elabora il primo programma sulla macchina usata e quale maschera vale nell'ultimo giro del secondo. Aggiungi un test con array di lunghezze diverse e verifica che entrambi lancino `IllegalArgumentException` prima di produrre un risultato. Distingui l'output numerico, che deve essere uguale nei due casi di prova, dal numero di corsie e dal tempo di esecuzione, che non sono risultati universali del libro. Se trasferisci il metodo a campioni audio o pixel, stabilisci prima quale risultato numerico deve restare invariato.

24. Reflection API: osservare i tipi a tempo di esecuzione

La domanda prima dello strumento

Un programma conosce normalmente i tipi che usa mentre viene compilato: chiama un metodo perché il compilatore può verificarne nome, parametri e accessibilità. In alcuni sistemi, però, il tipo concreto viene scelto più tardi. Un framework può cercare classi annotate, un serializzatore può esaminare la struttura di un record, uno strumento diagnostico può mostrare i metodi disponibili. La Reflection API permette di osservare informazioni sui tipi e, entro i confini di accesso previsti, di costruire oggetti o invocare membri a tempo di esecuzione. Non serve a rendere «dinamico» ogni programma Java; serve quando la struttura stessa è un dato del problema.

Nel capitolo sui classloader abbiamo visto come una definizione arrivi alla JVM. La reflection parte da un oggetto `Class<?>` già disponibile e chiede che cosa quella definizione esponga. Si può ottenere l'oggetto con `Libro.class`, con `oggetto.getClass()` o con una ricerca per nome come `Class.forName(nome)`. Le tre forme non sono sempre intercambiabili: la prima ha un tipo noto al compilatore, la seconda parte da un'istanza già presente, la terza può ricevere un nome scelto a tempo di esecuzione e deve gestire `ClassNotFoundException`.

Nel capitolo 20A abbiamo incontrato `VarHandle` come strumento avanzato per accedere a variabili con operazioni e ordini di memoria specifici. Un `VarHandle` non è la stessa cosa di un `Field` riflessivo: può essere ottenuto anche da una ricerca con `MethodHandles.Lookup`, ma il suo compito principale è eseguire accessi tipizzati alla variabile secondo il contratto del handle. La reflection di questo capitolo risponde anzitutto alla domanda «quali membri possiede questo tipo?». Non usiamo la sola scoperta di un campo per dichiarare sicura una modifica concorrente.

> **Definizione – Metadato.** Un metadato descrive un altro elemento. Il nome di una classe, i suoi metodi pubblici, il tipo di un parametro e un'annotazione disponibile a runtime sono metadati. La reflection legge o usa queste descrizioni; non cambia retroattivamente il sorgente compilato. Il valore di un campo di una specifica istanza è invece stato dell'oggetto, a cui si può accedere tramite un membro riflessivo quando le regole lo permettono.

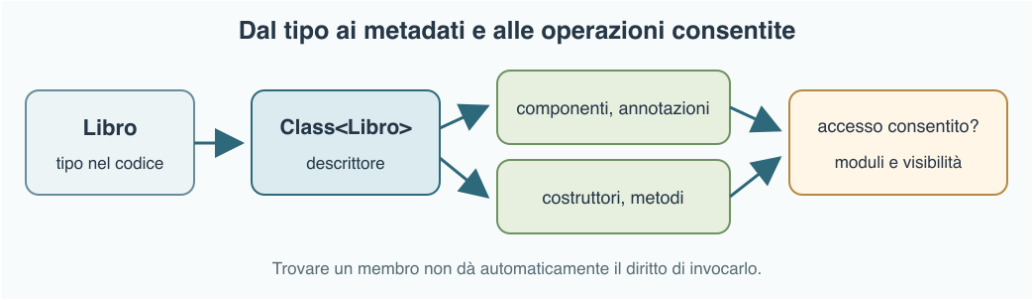


Figura 24.1 – Ispezione e invocazione sono passi distinti; l'accesso segue le regole del linguaggio e dei moduli.

Un programma che si lascia ispezionare

Nel programma completo definiamo un record `Libro` con due componenti, `titolo` e `pagine`, e un metodo `descrizione()`. La classe del record porta l'annotazione `@Etichetta("catalogo")`. Il programma ottiene `Libro.class`, chiama `isRecord()`, percorre `getRecordComponents()` e legge l'annotazione con `getAnnotation(Etichetta.class)`. La sequenza stampata inizia con il nome binario `EsploraReflection$Libro`, poi `record: true`, i due nomi delle componenti e l'etichetta.

Il simbolo `$` nel nome binario deriva dal fatto che, per mantenere l'esempio in un solo file, `Libro` è dichiarato come record membro di `EsploraReflection`. Un record dichiarato come tipo di primo livello avrebbe un nome diverso. Questa osservazione è utile quando si confronta il nome usato dal sorgente con quello richiesto da `Class.forName` o mostrato da un classloader.

L'annotazione definisce `@Retention(RetentionPolicy.RUNTIME)`. Senza questa politica, `getAnnotation` non potrebbe recuperarla durante l'esecuzione. `getAnnotation` restituisce l'annotazione cercata oppure `null`: il programma sa che il proprio tipo è annotato e usa direttamente `value()`, ma un lettore generico dovrebbe controllare il risultato. Le annotazioni da sole non eseguono alcuna logica; è il codice che le legge a decidere che cosa significano. Qui la stringa «catalogo» è soltanto un dato descrittivo.

Costruire e invocare

`getConstructor(String.class, int.class)` cerca un costruttore pubblico con esattamente quei tipi di parametro e restituisce un `Constructor<Libro>`. La chiamata `newInstance("Java", 320)` crea un `Libro`; il compilatore non ha scritto direttamente `new Libro("Java", 320)` nel punto d'uso, ma la verifica dei parametri e l'invocazione avvengono comunque secondo il contratto del costruttore riflessivo. Poi `getMethod("descrizione")` cerca un metodo pubblico senza parametri e `invoke(libro)` lo esegue su quell'istanza. Il risultato viene stampato come `Java (320 pagine)`.

```
Constructor<Libro> costruttore = tipo.getConstructor(String.class, int.class);
Libro libro = costruttore.newInstance("Java", 320);
Method metodo = tipo.getMethod("descrizione");
System.out.println(metodo.invoke(libro));
```

La forma riflessiva richiede più controlli della chiamata ordinaria. Il metodo potrebbe non esistere, la firma potrebbe essere diversa, il chiamante potrebbe non avere accesso, il costruttore potrebbe lanciare un'eccezione. Nell'esempio `main` dichiara `ReflectiveOperationException` perché è un programma dimostrativo; un'applicazione dovrebbe tradurre o propagare gli errori con un messaggio che spieghi quale componente non ha rispettato il contratto. Se il metodo invocato lancia un'eccezione, `Method.invoke` la incapsula in `InvocationTargetException`, la cui causa va esaminata per capire il problema originale.

> **Schema – Dove si sposta il controllo.** `libro.descrizione()` viene controllato dal compilatore rispetto al tipo statico di `libro`. `metodo.invoke(libro)` controlla a tempo di esecuzione che `metodo` sia invocabile su quell'oggetto. La flessibilità costa controlli più tardivi e rende il codice meno facile da leggere, refattorizzare e verificare automaticamente. Per una chiamata il cui tipo è già noto, la forma diretta resta più chiara.

Metodi pubblici, metodi dichiarati e accessibilità

`getMethod` cerca un metodo pubblico tenendo conto dell'ereditarietà. `getDeclaredMethod` cerca un metodo dichiarato direttamente dal tipo, anche se non è pubblico, ma il semplice fatto di ottenerne un oggetto `Method` non concede il diritto di invocarlo. La stessa distinzione vale per `getFields` e `getDeclaredFields`, per i costruttori pubblici e quelli dichiarati. Quando si costruisce uno strumento di ispezione, occorre scegliere se interessa l'API pubblica vista dai chiamanti oppure la struttura interna della classe esaminata. Mischiare le due viste può generare un rapporto fuorviante.

In Java 25 i moduli contano anche per la reflection. Esportare un pacchetto rende disponibili i tipi pubblici secondo le regole modulari; aprire un pacchetto con `opens` abilita l'accesso riflessivo profondo nei limiti dichiarati. `setAccessible(true)` non è un passepartout: se l'accesso non può essere abilitato, può essere lanciata `InaccessibleObjectException`. Un framework che richiede reflection sui membri non pubblici deve documentare la configurazione del modulo. Non suggeriamo di aprire indiscriminatamente tutti i pacchetti o di usare opzioni JVM di aggiramento come soluzione generale: il confine di incapsulamento è parte dell'architettura del programma.

> **Approfondimento – Perché `trySetAccessible` esiste.** Il metodo tenta di abilitare l'accesso e restituisce un booleano. Questo consente di trattare esplicitamente un rifiuto, quando l'applicazione prevede una strada alternativa. Se l'operazione è indispensabile, la soluzione corretta è definire un contratto di modulo e di API compatibile con quel bisogno. Anche la reflection su una classe presente non giustifica automaticamente l'accesso a ogni suo dettaglio privato.

Tipi generici e identità delle classi

Un oggetto `Class<List>` non conserva come parte della sua identità ogni argomento di tipo usato in una variabile `List<String>` o `List<Integer>`. L'erasure dei generics, già studiata, non scompare con la reflection. Alcune dichiarazioni conservano però metadati generici: `getGenericReturnType()` può descrivere la firma dichiarata di un metodo, e le API di `java.lang.reflect.Type` rappresentano forme come tipi parametrizzati e variabili di tipo. Questo non significa che, osservando un'istanza qualunque di `ArrayList`, possiamo recuperare sempre l'argomento `String` con cui una variabile fu dichiarata. Per un framework che deve conoscere il tipo degli elementi, il contratto deve fornire informazioni sufficienti.

Vale anche l'identità dei tipi spiegata nel capitolo sui classloader. Due oggetti `Class<?>` con lo stesso `getName()` non rappresentano necessariamente lo stesso tipo se provengono da caricatori definitori diversi. Un registro di plugin che usa soltanto il nome testuale come chiave può confondere definizioni isolate. Quando il programma passa un oggetto a `Method.invoke` o tenta un cast, la JVM verifica la vera identità del tipo, non soltanto le lettere del nome.

Prima di invocare: costruire una descrizione affidabile

Immaginiamo di estendere l'ispettore a tipi che non abbiamo scritto noi. Stampare soltanto `getName()` non basta: un parametro `int`, una classe `Integer` e un array `int[]` richiedono trattamenti diversi. `Class<?>` rappresenta anche primitive, array e `void`; non significa sempre «classe da istanziare». Usiamo `isPrimitive()`, `isArray()` e `getComponentType()` per riconoscere queste forme. Per una gerarchia chiusa, `isSealed()` e `getPermittedSubclasses()` descrivono le sottoclassi direttamente ammesse; non enumerano tutti gli oggetti che potranno esistere. Queste informazioni aiutano a validare un contratto prima di tentare operazioni che non hanno senso.

Per il nome distinguiamo il nome binario, adatto alla ricerca di classi ordinarie, dal nome canonico, vicino alla forma del sorgente. `getCanonicalName()` può restituire `null` per classi locali o anonime. Per mostrare un tipo a un lettore possiamo usare `getTypeName()`, ma una stringa leggibile non sostituisce l'identità dell'oggetto `Class`. Anche gli array hanno nomi binari particolari: non costruiamo un protocollo di caricamento concatenando a mano parentesi quadre al nome di una classe.

L'ispettore deve inoltre dichiarare quale vista dei membri sta mostrando. Un metodo ereditato è utile a chi vuole conoscere ciò che può chiamare; un campo privato dichiarato nella superclasse appartiene invece a un'altra vista. Per esaminare quest'ultimo dobbiamo risalire con `getSuperclass()` e interrogare ciascun tipo. I costruttori non sono ereditati: cercare il costruttore della sottoclasse non cerca quello del padre. L'ordine restituito per campi e metodi non è una presentazione stabile da usare in un file di confronto. Ordiniamo esplicitamente per nome e firma; per le componenti di un record conserviamo invece l'ordine dichiarato, significativo anche per il costruttore canonico.

> **Approfondimento – Il compilatore lascia tracce.** Un metodo *bridge* è un ponte generato per mantenere il polimorfismo dopo la cancellazione dei tipi generici. Un metodo *synthetic* è una struttura introdotta nella traduzione, anziché un metodo scritto ordinariamente nel sorgente. `isBridge()` e `isSynthetic()` aiutano a distinguerli. Un browser del sorgente può filtrarli; un diagnostico della JVM può aver bisogno di mostrarli. Filtrare tutto ciò che non riconosciamo produce un rapporto incompleto, mentre mostrare tutto senza spiegarlo fa credere al lettore di avere scritto metodi che non vede nel sorgente.

`Modifier.isStatic(membro.getModifiers())` permette di scegliere se occorre un destinatario. Un `Field` identifica una dichiarazione, non un valore già letto: `get(oggetto)` osserva il valore in quell'istanza, mentre per un campo statico possiamo passare `null`. Un primitivo letto con `get` viene restituito attraverso il relativo wrapper. Scrivere con `set` richiede accesso e conversioni ammesse; non esegue automaticamente la validazione di un setter. In un ispettore di catalogo scegliamo quindi la sola lettura. Se il requisito cambia in «modificare il numero di pagine», preferiamo un metodo applicativo che protegga l'invariante invece di trasformare ogni campo scoperto in una porta di scrittura.

La firma cercata e gli argomenti passati sono due problemi diversi

Supponiamo di aggiungere `descrizione(int limite)` al record. La ricerca deve usare `int.class`, anche se nell'invocazione passiamo un `Integer`: `getMethod` non applica la risoluzione del sovraccarico che il compilatore applica a una chiamata Java. Una firma con `Number` va cercata con `Number.class`, non con la classe concreta dell'argomento. Perciò un sistema configurabile deve conoscere i tipi della firma o definire una propria regola di selezione, con una risposta esplicita alle ambiguità.

Il frammento seguente usa il `Servizio` del programma `ContrattiReflection`. Il metodo `pagine` riceve un `int`; quello `conta` riceve un array di stringhe. La prima ricerca fissa una firma primitiva, la seconda una firma array.

```
var pagine = Servizio.class.getMethod("pagine", int.class);
Object risultato = pagine.invoke(servizio, Integer.valueOf(7));
var conta = Servizio.class.getMethod("conta", String[].class);
Object totale = conta.invoke(servizio, (Object) new String[]{"Java", "I/O"});
```

Otteniamo rispettivamente i wrapper del numero sette e del numero due. Cercare `pagine` con `Integer.class` produce invece `NoSuchMethodException`, prima di qualsiasi esecuzione del corpo.

Dopo la ricerca, `invoke` ammette unboxing e alcune conversioni di ampliamento verso primitive. Non effettua conversioni arbitrarie da stringhe né restringimenti numerici. `null` non può diventare un `int`. Un metodo statico può essere invocato con destinatario `null`; un metodo di istanza richiede un oggetto compatibile. Il risultato di un metodo `void` è `null`, quello di un risultato primitivo è un wrapper. Una chiamata a un metodo di istanza conserva il dispatch dinamico: trovare un metodo sul padre non impone di eseguire il corpo del padre quando il figlio lo ridefinisce.

> **Nota bene – Due livelli di varargs.** `Method.invoke` riceve un numero variabile di argomenti, ma il metodo cercato può avere a sua volta un parametro array. Per chiamare un metodo dichiarato con `String... valori`, cerchiamo `String[].class` e passiamo l'array come un unico argomento, per esempio `metodo.invoke(oggetto, (Object) valori)`. Il cast evita che quell'array venga interpretato come l'elenco degli argomenti di `invoke`. La reflection non confeziona automaticamente gli argomenti del metodo bersaglio come una chiamata varargs scritta nel sorgente.

Per provare la distinzione, aggiungiamo due firme con `int` e `Integer`, poi cerchiamole separatamente. Il successo della chiamata non dimostra di avere selezionato la firma voluta: controlliamo `getParameterTypes()` e il risultato. Se il contratto richiede la versione primitiva, un fallback silenzioso sul wrapper nasconde un errore di configurazione.

Parametri e annotazioni: leggere ciò che è stato conservato

Un mapper potrebbe associare i nomi dei parametri ai campi di una configurazione. `Method` e `Constructor` condividono la base `Executable`, da cui otteniamo `Parameter[]`. Tuttavia i nomi del sorgente non sono automaticamente disponibili per ogni metodo: per conservarli con `javac` si usa `-parameters`. `Parameter.isNamePresent()` permette di controllare il contratto. Se manca il nome, una stringa come `arg0` non è una chiave di dominio affidabile. Una variante robusta richiede un'annotazione esplicita oppure usa una posizione documentata.

Anche per le annotazioni bisogna scegliere quale domanda stiamo facendo. `getDeclaredAnnotations()` osserva quelle direttamente presenti. `@Inherited` riguarda l'eredità di annotazioni sulla classe attraverso le superclassi; non propaga automaticamente annotazioni dai metodi ridefiniti né dalle interfacce. Per annotazioni ripetibili, `getAnnotationsByType` considera il contenitore previsto dal contratto: cercare soltanto una singola annotazione può perdere occorrenze. Infine un'annotazione sul tipo usato da un parametro non coincide con un'annotazione sulla dichiarazione del parametro: il modello `AnnotatedType` serve al primo caso.

Il criterio riusabile è stabilire dove vive il significato. Se un validatore vuole un vincolo su ciascun elemento di `List<String>`, leggere soltanto le annotazioni della lista come parametro non descrive necessariamente il suo argomento di tipo. Torniamo al capitolo 14 e dichiariamo retention e target prima di scrivere il lettore. Verifichiamo poi una dichiarazione annotata e una priva di annotazione: l'assenza deve essere una situazione prevista, non un `NullPointerException` accidentale.

Attraversare un tipo generico senza fingere che sia una classe

Consideriamo un campo dichiarato `List<? extends Number> valori`. `getType()` dà la classe raw `List`, utile per alcune operazioni runtime. `getGenericType()` descrive invece la dichiarazione. Il risultato è un `Type`, non necessariamente un `Class<?>`: un cast diretto a `Class` fallirebbe proprio sul caso che volevamo capire.

Forma osservata	Informazione da esaminare	Domanda nel catalogo
`Class<?>`	Identità del tipo; eventuale componente array	È `String`, `int` o un array concreto?
`ParameterizedType`	Tipo raw, argomenti e tipo proprietario	Quali argomenti dichiara questa `List`?
`TypeVariable<?>`	Dichiarazione di origine e limiti	Che cosa ammette il parametro `T`?
`WildcardType`	Limiti superiori e inferiori	Che cosa possiamo assumere su `? extends Number`?
`GenericArrayType`	Tipo generico della componente	Come è dichiarato un array `T[]`?

Per il campo `valori` dello `Schema` nel programma di prova, possiamo svolgere il primo livello così:

```
var lista = (ParameterizedType) Schema.class
        .getDeclaredField("valori").getGenericType();
var limite = (WildcardType) lista.getActualTypeArguments()[0];
Type confine = limite.getUpperBounds()[0];
```

Qui `Type` va importato da `java.lang.reflect`. I cast sono leciti perché conosciamo la dichiarazione di questo esempio: `confine` è `Number.class`. Un ispettore generale deve invece controllare la forma con `instanceof` prima di convertirla. Se il campo diventasse semplicemente `Number`, il primo cast non sarebbe più valido.

La tabella è una mappa per costruire un visitatore: riconosciamo la forma, leggiamo le sue parti, ripetiamo il ragionamento sulle parti. `List<List<String>>` richiede due livelli. Un limite come `T extends Comparable<T>` può riportarci alla variabile iniziale: un descrittore ricorsivo deve ricordare i tipi già visitati per non procedere senza fine. Questa cautela nasce dal problema concreto, non dalla quantità di classi dell'API.

Nel programma di verifica controlliamo una lista parametrizzata, una wildcard, una variabile e un array generico. Il limite è intenzionale: stiamo descrivendo firme, non verificando ogni elemento di una lista ricevuta. Se un importatore deve accettare soltanto numeri, deve validare i valori. Una firma generica utile per documentare lo schema non annulla l'erasure né impedisce a codice legacy di consegnarci dati incompatibili.

Accesso profondo: una prova deve attraversare davvero il modulo

`canAccess(destinatario)` risponde se il membro è accessibile al chiamante nelle condizioni attuali. Per membri statici e costruttori il destinatario richiesto è `null`; per membri di istanza è un oggetto compatibile. `trySetAccessible()` pone una domanda diversa: possiamo sopprimere i controlli di accesso secondo le regole della piattaforma? Un `false` va gestito; dopo un tentativo riuscito non abbiamo trasformato la dichiarazione privata in una API pubblica stabile.

Una prova in cui ispettore e oggetto sono classi annidate nella medesima classe esterna può avere accessi già consentiti dal rapporto di *nestmate*, cioè appartenenza allo stesso gruppo di classi annidate. Non dimostra che un framework separato possa fare altrettanto. Proviamo quindi due moduli nominati: il modulo del catalogo esporta il package pubblico, ma inizialmente non lo apre; quello dell'ispettore richiede il catalogo. L'accessore pubblico rimane invocabile, il campo privato non diventa accessibile solo perché il package è esportato. Aggiungendo un'apertura qualificata `opens biblioteca.modello to biblioteca.ispettore` cambiamo precisamente il contratto della seconda operazione.

> **Concetto chiave – Apertura mirata.** `exports` serve al contratto pubblico; `opens` autorizza la reflection profonda verso i destinatari previsti. Un'apertura non equivale a rendere importabili dal sorgente tutti i membri privati. La configurazione appartiene al modulo che possiede il package. Se il framework può lavorare con accessori pubblici, quella alternativa riduce la dipendenza dalla rappresentazione interna.

La prova modulare completa usa il modulo del modello. Lo script di verifica compila prima il modulo chiuso, poi ne prepara una copia temporanea con apertura qualificata. Le due esecuzioni stampano `accesso profondo: false` e `accesso profondo: true`, mantenendo in entrambe il successo dell'accessore pubblico. La copia temporanea evita di confondere i due contratti.

Non usiamo il successo di `trySetAccessible` per promettere che qualunque campo `final` sia modificabile. In particolare i campi finali dei record e i campi `static final` non diventano scrivibili in questo modo. Per ricostruire un record leggiamo nomi e tipi delle componenti, cerchiamo il costruttore canonico e passiamo valori validati nel medesimo ordine. Il costruttore conserva il punto in cui far rispettare gli invarianti. La variante è un record con una componente aggiuntiva: il mapper deve adattare la firma o rifiutare lo schema, anziché assumere per sempre due parametri.

Array costruiti quando il tipo arriva più tardi

Se un importatore scopre soltanto a runtime il tipo della componente, non può scrivere `new T[n]` per un generico arbitrario. `java.lang.reflect.Array.newInstance(tipoComponente, lunghezza)` costruisce l'array concreto; `Array.getLength`, `Array.get` e `Array.set` permettono di usarlo attraverso un riferimento `Object`. Con componente `int.class` otteniamo un vero `int[]`, non un `Integer[]`. Per un array di primitive non è valido il cast a `Object[]`.

La prova è creare tre interi, assegnarne uno, leggere valore e lunghezza. Poi tentiamo di inserire una stringa: il rifiuto dimostra che l'array mantiene il proprio tipo. Se il programma deve soltanto raccogliere oggetti eterogenei, una `List<Object>` può essere più chiara. L'array riflessivo risolve il requisito «serve proprio un array di questo tipo runtime», non il requisito generico «mi serve una raccolta».

Proxy dinamici: intercettare un contratto di interfaccia

Ora la richiesta è contare le chiamate al servizio del catalogo senza duplicare il suo lavoro. Un *proxy* è un oggetto che si presenta al chiamante attraverso lo stesso contratto e decide come inoltrare la richiesta. `Proxy.newProxyInstance` crea un'implementazione runtime delle interfacce indicate; un `InvocationHandler` riceve il metodo e gli argomenti. Non crea una sottoclasse arbitraria della classe concreta, e le interfacce devono rispettare i vincoli dell'API, inclusa l'esclusione delle interfacce sealed.

Nel programma `ContrattiReflection` il servizio ha una sola operazione: il gestore incrementa un contatore e chiama il metodo sull'oggetto reale. La parte decisiva è il destinatario. Invocare lo stesso metodo sul proxy dall'interno del suo gestore produrrebbe ricorsione; lo invochiamo sul servizio. Se il servizio lancia un'eccezione, estraiamo la causa di `InvocationTargetException`. Lasciare il wrapper aggiungerebbe un livello artificiale e potrebbe alterare il contratto delle eccezioni checked dell'interfaccia.

Anche `equals`, `hashCode` e `toString` passano attraverso il gestore. Il nostro esempio assegna al proxy uguaglianza per identità e un testo descrittivo, senza contarli come richieste al catalogo. Una delega ingenua di `equals` al servizio può perfino rendere falsa l'uguaglianza del proxy con sé stesso. I metodi default richiedono una politica esplicita; `InvocationHandler.invokeDefault` consente di eseguire il default sul proxy quando questa è la scelta prevista.

Per trasferire l'idea, sostituiamo il contatore con un controllo di autorizzazione, mantenendo l'interfaccia e la gestione degli errori. Se il servizio è unico e noto, un decoratore scritto a mano può offrire lo stesso comportamento con più controlli del compilatore. Il proxy diventa utile quando le interfacce sono scoperte o trattate uniformemente da un'infrastruttura. La comodità di intercettare non elimina il bisogno di definire quali chiamate contare e quale identità assegnare all'oggetto.

Reflection, handle e class file: scegliere il livello

Un MethodHandle rappresenta un'operazione invocabile con un tipo esplicito e controlli di accesso legati alla ricerca tramite `Lookup`. È adatto a infrastrutture che preparano e compongono chiamate; non è una promessa automatica di maggiore velocità. `invokeExact` richiede corrispondenza esatta del tipo della chiamata, mentre gli adattamenti vanno progettati. Un `VarHandle` riguarda accessi a variabili e relativi modi di ordinamento, come abbiamo visto nel capitolo 20A. La Core Reflection rimane più diretta per scoprire e descrivere membri.

La Class-File API di `java.lang.classfile`, stabile da Java 24, lavora invece sulla rappresentazione dei file di classe: serve per analizzarli, generarli o trasformarli. Non è la chiamata a un metodo di un oggetto già presente. Se vogliamo inventariare un archivio senza caricare le classi, questo livello può essere

pertinente; se vogliamo leggere una componente di un record vivo, il livello è diverso.

Un'ultima scelta riguarda il riuso delle descrizioni. Scoprire la stessa firma a ogni chiamata può essere inutile: possiamo conservarla per il tipo già verificato. Una mappa globale con riferimenti forti a classi di plugin può però trattenere anche i loro classloader. `ClassValue` offre un'associazione per classe utile per questo genere di metadati; il valore deve essere progettato senza collegamenti indesiderati ad altre generazioni di plugin. Prima di ottimizzare misuriamo il carico reale: il costo di accesso al file o di parsing può dominare quello della reflection. Il capitolo 29 fornisce gli strumenti per rendere ripetibile la misura.

> **Riferimenti essenziali – Dal contratto al controllo.** La specifica Core Reflection chiarisce il rapporto con il modello JVM. Le specifiche di Class, Method, AccessibleObject e Proxy sono i riferimenti per ricerca, invocazione, accesso e intercettazione. Descrivono le garanzie della piattaforma; la scelta dello schema, la validazione e la politica del proxy restano responsabilità dell'applicazione.

Quando la reflection è una buona scelta

Uno strumento che esplora record e annotazioni a runtime ha una motivazione precisa per usare reflection. Anche un framework di dipendenze, un test runner o un sistema di serializzazione possono averne bisogno. Una normale chiamata applicativa fra due classi note al compilatore, invece, è più semplice senza reflection. Conviene valutare costi di prestazione, gestione degli errori e stabilità dei nomi: se rinominiamo un metodo, una stringa passata a `getMethod` potrebbe non essere rilevata dal compilatore. Test che coprono la scoperta riflessiva diventano quindi parte del contratto.

Come esercizio, si aggiunga a `Libro` un metodo pubblico con un parametro e si modifichi la ricerca con `getMethod` usando il tipo esatto del parametro. Poi si cerchi un nome inesistente e si osservi `NoSuchMethodException`, distinguendola da `ClassNotFoundException` del capitolo sui classloader. Infine si sposti il record in un modulo che non apre il pacchetto e si ragioni su quali operazioni pubbliche restano lecite e quali richieste di accesso profondo falliscono. Lo scopo è imparare a individuare il confine, non a superarlo automaticamente.

Una ricerca riflessiva che fallisce: leggere l'errore

Supponiamo che il nome di un metodo arrivi da una configurazione e che il programma esegua `tipo.getMethod(nomeConfigurato)`. Se la configurazione contiene un refuso, non esiste alcun controllo del compilatore su quella stringa. A tempo di esecuzione la ricerca lancia `NoSuchMethodException`. La risposta utile non è stampare soltanto una lunga traccia: occorre dire quale nome è stato richiesto, su quale classe, con quali tipi di parametri e da quale configurazione proveniva. I tipi dei parametri fanno parte della ricerca. `getMethod("descrizione")` e `getMethod("descrizione", String.class)` pongono domande diverse anche se il nome è uguale.

Trovato il metodo, l'invocazione può ancora fallire. Se l'oggetto passato non è un'istanza compatibile con il metodo di istanza, oppure il numero e il tipo dei

parametri non corrispondono, l'API segnala l'errore al chiamante. Se invece il corpo del metodo viene eseguito e lancia una sua eccezione, `InvocationTargetException` conserva quella causa. Per diagnosticare, distinguiamo l'errore nel *trovare e chiamare* il metodo dall'errore prodotto *dal metodo chiamato*. È la stessa disciplina usata nel capitolo sulle eccezioni: il punto in cui l'errore viene visto non coincide sempre con quello in cui nasce.

Un framework può nascondere questi dettagli dietro messaggi più comprensibili, ma il codice che usa reflection deve conoscerli. Se il contratto del framework dice «un componente annotato deve avere un costruttore pubblico senza argomenti», un `NoSuchMethodException` indica una violazione precisa di quel contratto. Se l'accesso è negato dal modulo, l'errore è diverso: modificare il nome del metodo non aprirà il pacchetto. Se il metodo stesso lancia un'eccezione di dominio, aprire il modulo non la risolverà. Una diagnosi accurata evita correzioni casuali come aggiungere opzioni JVM finché il programma smette di fallire.

La reflection è dunque potente quando il programma deve lavorare con metadati scoperti a tempo di esecuzione, ma richiede contratti espliciti e prove che attraversino anche i casi di errore. Un capitolo sui classloader spiega da dove arriva la definizione; questo capitolo spiega come interrogarla e usarla entro i suoi confini. Il passo successivo, in un'applicazione reale, è decidere se la flessibilità guadagnata giustifica la complessità aggiunta. Spesso un'interfaccia comune, un servizio modulare o una normale chiamata tipizzata esprimono meglio lo stesso bisogno. Quando la struttura non è nota in anticipo, invece, la reflection permette di scoprirla senza fingere che il compilatore potesse verificarla prima.

Per verificare

Nel programma di questo capitolo cambia prima il nome richiesto in `getMethod`, poi il tipo del parametro del costruttore, in due copie separate. Confronta le eccezioni e identifica quale ricerca le produce. Ripristinato il codice, fai lanciare un'eccezione da `descrizione()` e osserva la causa riportata da `Method.invoke`. Racconta i tre punti del percorso: ricerca del membro, controllo dell'accesso e sua esecuzione. Infine chiediti se, per un tipo noto già in compilazione, questa ricerca sia necessaria: la scelta dello strumento deve seguire il problema, come nel caso del plugin del capitolo 21.

25. I/O, file e risorse: portare i dati oltre il programma

Un archivio che sopravvive all'esecuzione

Finora molti esempi hanno creato i dati in memoria e li hanno perduti alla fine di `main`. Una biblioteca, invece, deve poter leggere domani i titoli registrati oggi. Questo cambiamento introduce un confine: il programma controlla i suoi oggetti, ma non controlla completamente il file system. Il file può mancare, essere illeggibile o cambiare mentre lo consultiamo; una scrittura può finire lo spazio disponibile. La prima domanda non è «quale metodo di `Files` devo ricordare?», ma «che cosa sto trasferendo, fra quali luoghi, e che cosa deve restare vero se il trasferimento fallisce?».

Chiamiamo *I/O* l'insieme delle operazioni di ingresso e uscita. Una sorgente fornisce dati, una destinazione li riceve. Un file è una possibile sorgente o destinazione; anche una connessione di rete lo è. Una risorsa è un oggetto che mantiene aperto qualcosa al di fuori della normale memoria degli oggetti Java, per esempio un descrittore di file. Chiudere la risorsa non è un dettaglio cosmetico: libera quella capacità e conclude l'operazione secondo il contratto dell'API.

La figura 25.1 segue un testo dall'archivio fino al programma. I byte sul disco vengono interpretati con una codifica; il risultato sono caratteri e stringhe. Nel percorso inverso, i caratteri devono essere codificati in byte. Se l'archivio contiene un'immagine, quella conversione testuale sarebbe un errore: l'immagine va trattata come byte. Questa distinzione collega il capitolo 11 su Unicode al capitolo 12 sulle eccezioni.

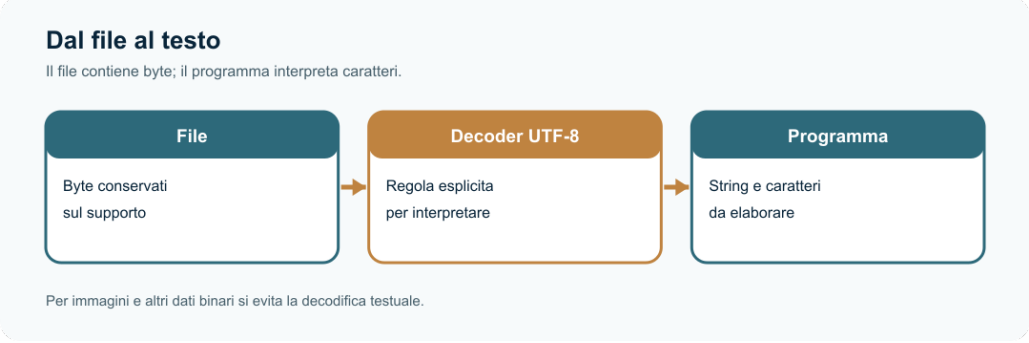


Figura 25.1 – La codifica collega i byte del file ai caratteri del programma.

Byte, caratteri e codifica

`InputStream` legge byte, `OutputStream` li scrive. Il valore restituito da `read()` è un intero da 0 a 255, oppure `-1` quando il flusso è terminato: `-1` non è un byte del file. Una chiamata che legge in un array può ottenere meno byte di quelli richiesti, anche quando altri dati arriveranno. Il codice che ha bisogno di tutto il contenuto deve ripetere la lettura fino a fine flusso; `transferTo` esprime una copia semplice, ma non rende la destinazione automaticamente atomica o recuperabile dopo un guasto.

`Reader` e `Writer` lavorano invece con caratteri Java, rappresentati nei loro metodi elementari da unità UTF-16. Un `Reader` non equivale a un lettore di code point Unicode: un simbolo fuori dal piano multilingue di base usa due unità. Per passare fra byte e caratteri servono un decoder e un encoder. `InputStreamReader` e `OutputStreamWriter` fanno da ponte; `Files.newBufferedReader` e `Files.newBufferedWriter` sono scelte comode quando l'origine è un `Path`. Nel programma completo indichiamo `StandardCharsets.UTF_8` in entrambe le direzioni, così il formato dell'archivio è dichiarato dal codice.

> **Concetto chiave – Il charset è parte del formato.** UTF-8 stabilisce come rappresentare i caratteri in byte. La parola Java `String` non è già «un file UTF-8»: è testo in memoria. Se produttore e lettore scelgono codifiche diverse, il testo può corrompersi o la decodifica può fallire. Quando il formato è noto, non affidiamoci al charset predefinito della macchina; un eventuale byte order mark e il trattamento degli input malformati vanno decisi dal contratto del file.

Un buffer trattiene temporaneamente dati per ridurre il numero di accessi all'origine. Avvolgere un flusso in `BufferedInputStream` o un lettore in `BufferedReader` può aiutare quando leggiamo molti frammenti piccoli. Non risolve però il problema di un algoritmo che carica un archivio enorme tutto in memoria. `Files.readString` è adatto a un file piccolo di cui possiamo accettare il costo in memoria; per un archivio lungo usiamo un lettore e processiamo una riga alla volta. L'analoga scelta in uscita è fra `Files.writeString` e una scrittura progressiva con `BufferedWriter`.

Un caso concreto aiuta a scegliere. Se dobbiamo copiare un'immagine senza modificarla, `Files.copy` esprime già l'operazione fra due percorsi. Se la sorgente è una connessione e la destinazione è un file, `InputStream.transferTo(OutputStream)` trasferisce byte senza introdurre una codifica testuale. Se invece dobbiamo contare titoli che iniziano con una lettera, serve interpretare righe e caratteri, quindi il lettore testuale è parte della soluzione. Il fatto che tutte e tre le operazioni «leggano un file» non le rende intercambiabili: cambia il significato dei dati.

Con una lettura manuale a blocchi, il valore restituito da `read(byte[])` indica quanti elementi dell'array sono stati riempiti; soltanto quelli vanno scritti. Un programma che scrive sempre l'intero array può aggiungere al file di uscita byte vecchi nell'ultimo giro. Anche `0` e `-1` hanno significati diversi quando una particolare API può restituire zero: la fine del flusso è segnalata da `-1`. Per imparare questa disciplina, si può copiare un file di lunghezza non multipla della dimensione del buffer e confrontare byte per byte il risultato.

Decorare un flusso: ogni strato aggiunge un compito

Supponiamo di leggere da una sorgente che fornisce pochi byte alla volta. Non cambia il formato, ma cambia il costo di ogni accesso. Possiamo avvolgere la sorgente con un buffer e poi con un lettore testuale: sorgente di byte → buffer → decoder → lettore di righe. La decorazione mantiene il contratto di base e aggiunge un comportamento. Un `BufferedReader` non sa da solo quale fosse la codifica dei byte prima del decoder; un `BufferedInputStream` non sa dove finisca una riga. Ogni strato risponde a una domanda diversa.

Quando siamo noi a creare tutta la catena, chiudiamo il suo oggetto esterno con `try-with-resources`: i wrapper ordinari propagano la chiusura alla risorsa sottostante. Se un metodo riceve uno stream dal chiamante, invece, il contratto deve dire chi lo possiede. Un metodo di copia non dovrebbe chiudere senza avviso `System.out` o uno stream che il chiamante deve riutilizzare. `transferTo` trasferisce il contenuto, ma non chiude i due estremi. La proprietà della risorsa è una responsabilità progettuale, non qualcosa che il compilatore deduce dal tipo.

> **Nota bene – Fine del flusso e disponibilità.** `available()` non misura la lunghezza complessiva dell'ingresso e non è una prova di fine flusso: stima byte leggibili senza bloccare. `readAllBytes()` legge tutto e consuma memoria proporzionale ai dati; `readNBytes(n)` può restituire meno di `n` byte se l'ingresso termina. Se il formato richiede esattamente una certa quantità, controlliamo la lunghezza ottenuta. `mark` e `reset` consentono di tornare a un punto soltanto negli stream che li supportano e nei limiti dichiarati; non trasformano qualunque sorgente in un file ad accesso casuale.

Nel programma sui contratti I/O usiamo un ingresso in memoria che limita deliberatamente ogni lettura a tre byte. Il ciclo scrive soltanto il numero restituito e il confronto finale controlla l'intero risultato. È un esperimento piccolo, ma toglie l'illusione che «ho chiesto otto byte» significhi «ne ho ricevuti otto». La variante sostituisce l'array in memoria con una sorgente di rete: il principio resta valido anche se ora una lettura può attendere nuovi dati.

Per un formato binario con numeri primitivi, `DataInputStream` e `DataOutputStream` aggiungono operazioni come `readInt` e `writeInt`. Il contratto deve fissare ordine dei campi, dimensioni, versione e risposta al troncamento; `readInt` segnala una fine anticipata con `EOFException`. `writeUTF` usa una forma UTF-8 modificata e una lunghezza limitata: non è una scorciatoia per un file di testo UTF-8 generale. Per dati compressi possiamo aggiungere gli stream di `java.util.zip`, ma la dimensione decompressa richiede limiti propri: un ingresso piccolo può produrre molto più contenuto. Il criterio rimane scegliere lo strato che rappresenta il formato, non aggiungerne finché l'esempio funziona.

Una decodifica che rifiuta dati ambigui

Il catalogo arriva da un sistema esterno che dichiara UTF-8. Vogliamo rifiutare una sequenza di byte malformata anziché sostituirla con un carattere e pubblicare un titolo diverso. Un `CharsetDecoder` consente di scegliere `CodingErrorAction.REPORT` per l'ingresso malformato e per i caratteri non rappresentabili. Possiamo passarlo a `InputStreamReader`, poi avvolgere questo in `BufferedReader`. Non affidiamoci a una presunta uniformità dei costruttori: un ponte costruito soltanto con un charset può avere una politica di sostituzione, mentre il contratto di `Files.newBufferedReader` prevede errori per sequenze malformate o non mappabili.

Per costruire questo lettore, il seguente frammento assume un `InputStream ingresso` già disponibile. Il metodo che esegue la lettura propaga o gestisce `IOException`; chiudendo `lettore` chiudiamo anche `ingresso`, secondo il contratto di proprietà appena stabilito.

```
var decoder = StandardCharsets.UTF_8.newDecoder()
        .onMalformedInput(CodingErrorAction.REPORT)
        .onUnmappableCharacter(CodingErrorAction.REPORT);
try (var lettore = new BufferedReader(new InputStreamReader(ingresso, decoder))) {
    String riga;
    while ((riga = lettore.readLine()) != null) {
        System.out.println(riga);
    }
}
```

Importiamo `CodingErrorAction` da `java.nio.charset` e i due lettori da `java.io`. Il decoder viene creato per questo flusso: non lo condividiamo fra letture concorrenti. Stampare è soltanto l'azione dimostrativa; nell'importatore la riga va validata prima di produrre il risultato.

La verifica usa i byte `C3 28`: il primo inizia una sequenza UTF-8 che il secondo non completa. Il lettore configurato per il rifiuto deve lanciare `MalformedInputException`, non produrre un titolo. Poi proviamo un titolo valido con un carattere accentato e uno fuori dal piano multilingue di base. Il successo sui soli caratteri ASCII non avrebbe verificato la parte interessante della codifica.

> **Approfondimento – Una sequenza può attraversare due blocchi.** Un carattere UTF-8 può occupare più byte. Se decodifichiamo ogni blocco isolatamente con `new String(blocco, UTF_8)`, il confine del buffer può tagliare la sequenza e corromperla. Un lettore o un decoder mantenuto lungo il flusso conserva lo stato necessario. L'I/O a blocchi suddivide il trasporto, non definisce i confini dei caratteri o dei record del formato.

Anche leggere «una riga alla volta» ha un limite: `readLine()` costruisce l'intera riga in memoria e rimuove il terminatore. Un file senza a capo può quindi essere una singola riga enorme. Se l'ingresso non è controllato, definiamo un massimo per byte, caratteri e lunghezza delle righe, imponendolo durante la lettura, non dopo aver allocato tutto. `newLine()` usa il separatore della piattaforma; se il protocollo richiede LF, scriviamo esplicitamente `\n`. La trasformazione del nostro catalogo sceglie un archivio locale: per un formato di scambio rendiamo esplicita anche questa scelta.

Il nome di un file non è il suo contenuto

`Path` rappresenta un percorso nel file system; `Files` esegue le operazioni su quel percorso. Con `Path.of("dati", "libri.txt")` il programma costruisce un percorso relativo: il suo significato dipende dalla directory di lavoro del processo. Un percorso assoluto, come quello restituito da `toAbsolutePath()`, indica una posizione rispetto alla radice del file system. Nessuno dei due garantisce che il file esista. `resolve` unisce parti di percorso; `relativize` descrive come raggiungere un percorso da un altro, quando il provider lo consente e le radici sono compatibili.

`normalize()` elimina elementi lessicali come `.` e coppie `nome/..`. Non legge il file system e non è una verifica di sicurezza. In presenza di collegamenti simbolici, il percorso normalizzato e quello realmente visitato possono differire. `toRealPath()` interroga il file system e risolve i collegamenti secondo le opzioni indicate, ma richiede che il percorso sia accessibile. Se un'applicazione accetta nomi di file da

un utente, un semplice controllo testuale sul prefisso non basta a garantire che tutte le aperture future rimangano entro una directory: altri processi possono cambiare il file system fra controllo e uso.

Prendiamo `Path.of("archivi").resolve("2026").resolve("libri.txt")`. L'API conserva i separatori appropriati al provider: non serve inserire manualmente `/` o `\\` nella stringa. Su un percorso relativo, `toAbsolutePath` usa la directory di lavoro; non trasforma il percorso in una risorsa fissa dell'applicazione. Se l'utente avvia il programma da un'altra cartella, quel medesimo nome relativo può designare un altro file. Il programma deve quindi ricevere la directory di lavoro come parametro o ricavarla da una configurazione esplicita. Un percorso usato nei test può essere creato con `Files.createTempDirectory`, come fa il nostro esempio, per evitare dipendenze dal computer dell'autore.

Una chiamata a `Files.exists` può restituire `false` sia quando il file manca sia quando non è possibile determinarne l'esistenza. In ogni caso `if (Files.exists(p)) Files.readString(p)` non elimina l'eccezione: fra le due chiamate il file può sparire. È meglio tentare l'operazione, poi trattare `NoSuchFileException`, `AccessDeniedException` o una più generale `IOException` secondo la decisione che l'applicazione deve prendere. Il capitolo sulle eccezioni ci ha preparato proprio a distinguere il fallimento previsto dall'invariante violato.

Aprire in scrittura significa scegliere che cosa può andare perduto

Prima di modificare un archivio dobbiamo rispondere a una domanda: il file deve essere nuovo, sostituito oppure prolungato? Le opzioni di apertura rendono la risposta concreta. Con le impostazioni ordinarie di `Files.newBufferedWriter` il file viene creato se manca e troncato se esiste. È comodo per un risultato rigenerabile; è pericoloso se credevamo di aggiungere un titolo alla fine. `APPEND` chiede di scrivere in coda; `CREATE_NEW` richiede che il nome non esista e fa fallire l'apertura se è già occupato. `CREATE` consente invece anche un file esistente.

Immaginiamo di creare una ricevuta numerata. Il controllo «se non esiste, creala» lascia un intervallo in cui un altro processo può usare lo stesso nome. Un'apertura con `CREATE_NEW` esprime direttamente il vincolo. Il programma di prova crea la ricevuta e tenta di crearla una seconda volta: attendiamo `FileAlreadyExistsException` e verifichiamo che il contenuto originale sia rimasto intatto. Se il requisito cambia in «aggiungi righe al registro», la scelta diventa `APPEND`, ma dobbiamo ancora definire coordinamento e recupero per scrittori concorrenti. Non assumiamo che una sequenza di più scritture costituisca un record indivisibile su ogni provider.

`flush()` spinge dati dal buffer verso lo strato successivo. Non certifica che siano già su un supporto persistente; `close()` può fallire mentre completa la scrittura. Per questo il nostro importatore pubblica dopo essere uscito dal blocco delle risorse. `FileChannel.force` può richiedere la persistenza di aggiornamenti del file nelle condizioni documentate, ma la durabilità di una sostituzione coinvolge anche il file system e i metadati della directory. Non ricaviamo una promessa universale di recupero da un singolo metodo.

Un'importazione che non pubblica risultati a metà

Supponiamo di trasformare un archivio di titoli. Vogliamo eliminare righe vuote e spazi ai bordi, poi scrivere i titoli in maiuscolo. Se scriviamo direttamente sul file finale e la riga cento contiene un dato non valido, i lettori vedranno un archivio troncato. La soluzione svolta usa un file provvisorio nella stessa directory della destinazione. Solo dopo la lettura, la validazione e la chiusura del writer chiede uno spostamento atomico. La figura 25.2 rende visibile la differenza fra «sto costruendo» e «è pronto per i lettori».



Figura 25.2 – Costruire il provvisorio e pubblicarlo sono operazioni distinte.

Il codice è completo e può essere compilato con `javac --release 25 ImportaArchivio.java` ed eseguito con `java ImportaArchivio`. I percorsi del programma di prova vengono creati in una directory temporanea: l'esempio non dipende dal nome di una cartella sul computer del lettore.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.AtomicMoveNotSupportedException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;

public class ImportaArchivio {
    static int importa(Path sorgente, Path destinazione) throws IOException {
        Path cartella = destinazione.toAbsolutePath().getParent();
        Path provvisorio = Files.createTempFile(cartella, "archivio-", ".tmp");
        boolean pubblicato = false;
        Throwable fallimento = null;
        try {
```

```
            int righe = 0;
            int numeroRiga = 0;
            try (BufferedReader lettore = Files.newBufferedReader(
                    sorgente, StandardCharsets.UTF_8);
                 BufferedWriter scrittore = Files.newBufferedWriter(
                    provvisorio, StandardCharsets.UTF_8)) {
                String riga;
                while ((riga = lettore.readLine()) != null) {
                    numeroRiga++;
                    String valore = riga.strip();
                    if (valore.isEmpty()) {
                        continue;
                    }
                    if (valore.contains(";")) {
                        throw new IOException("Separatore inatteso alla riga " + numeroRiga);
                    }

                    scrittore.write(valore.toUpperCase(java.util.Locale.ROOT));
                    scrittore.newLine();
                    righe++;
                }
            }
            try {
                Files.move(provvisorio, destinazione,
                        StandardCopyOption.ATOMIC_MOVE,
                        StandardCopyOption.REPLACE_EXISTING);
            } catch (AtomicMoveNotSupportedException ex) {
                throw new IOException("Pubblicazione atomica non disponibile", ex);
            }
            pubblicato = true;
            return righe;
        } catch (IOException | RuntimeException | Error ex) {
            fallimento = ex;

            throw ex;
        } finally {
            if (!pubblicato) {
                try {
                    Files.deleteIfExists(provvisorio);
                } catch (IOException | RuntimeException pulizia) {
                    if (fallimento != null) {
                        fallimento.addSuppressed(pulizia);
                    } else {
                        throw pulizia;
                    }
                }
            }
        }
    }
```

```
    public static void main(String[] args) throws IOException {
        Path cartella = Files.createTempDirectory("esempio-c25-");
        Path sorgente = cartella.resolve("ingresso.txt");
        Path destinazione = cartella.resolve("uscita.txt");
        try {
            Files.writeString(sorgente, "  libro  \n\njava\n", StandardCharsets.UTF_8);
            System.out.println("Righe importate: " + importa(sorgente, destinazione));
            System.out.print(Files.readString(destinazione, StandardCharsets.UTF_8));
        } finally {
            Files.deleteIfExists(destinazione);
            Files.deleteIfExists(sorgente);
            Files.deleteIfExists(cartella);
        }
    }
}
```

`createTempFile` crea davvero il file, invece di inventare un nome sperando che sia libero. Il file viene creato accanto alla destinazione perché uno spostamento atomico fra file system diversi in genere non è disponibile. La variabile `pubblicato` distingue il successo dall'uscita per eccezione; nel secondo caso `finally` tenta di eliminare il provvisorio. Se la pulizia fallisce durante la gestione di un errore, `addSuppressed` conserva quel secondo problema senza nascondere il primo. Un provvisorio non eliminato richiederà una pulizia successiva: non possiamo promettere che ogni guasto lasci la directory vuota. Le due risorse del `try` vengono chiuse in ordine inverso. Se la lettura o la scrittura fallisce e anche la chiusura fallisce, l'errore di chiusura viene conservato come *eccezione soppressa*: non cancella la causa che ha interrotto il lavoro.

Nel ciclo, `readLine()` restituisce `null` alla fine. Ogni riga non vuota viene validata prima di essere scritta. La condizione sul punto e virgola è un vincolo inventato per l'esempio, non una regola dei file di testo. Il contatore `numeroRiga` avanza anche sulle righe vuote, mentre `righe` conta soltanto i titoli importati: il messaggio di errore indica così la posizione fisica nell'ingresso. Un vero archivio avrebbe inoltre una grammatica dichiarata e una politica sui duplicati. Eseguendo il programma, osserviamo `Righe importate: 2`, poi `LIBRO` e `JAVA` su righe distinte. L'esito dimostra il percorso positivo; per provare quello negativo inseriamo `titolo;errato` nel sorgente e verifichiamo che il file finale precedente non sia stato sostituito.

La prova del fallimento esegue proprio quest'ultimo esperimento su file temporanei: crea una destinazione con `VERSIONE PRECEDENTE`, incontra una riga malformata, poi controlla che il contenuto precedente sia intatto e che nessun file provvisorio sia rimasto nella cartella. Stampa `Destinazione precedente intatta; provvisorio rimosso`. Il test rende osservabile la proprietà per cui abbiamo introdotto il file provvisorio; una semplice esecuzione positiva non l'avrebbe dimostrata. Non prova ancora la resistenza a un crash improvviso o il comportamento di un provider diverso.

> **Attenzione – Atomico non significa durabile.** `ATOMIC_MOVE` richiede al provider uno spostamento indivisibile per gli osservatori; se non è supportato il programma fallisce esplicitamente. La combinazione delle opzioni e la sostituzione di una destinazione esistente hanno dettagli dipendenti dal provider: un sistema reale deve provarli sul proprio file system. Lo spostamento atomico non promette, da solo, che i byte sopravvivano a uno spegnimento improvviso: quella è una diversa garanzia di persistenza. Se l'applicazione accetta una pubblicazione non atomica, deve definire e documentare un piano di recupero, invece di nascondere il ripiego dentro un `catch`.

Il termine «atomico» era comparso nel capitolo 20A per una decisione su stato condiviso nella JVM. Qui il confine è diverso: gli osservatori del file system devono vedere il vecchio file oppure il nuovo, secondo le garanzie del provider. Un contatore Java atomico non rende indivisibile la pubblicazione del file; `ATOMIC_MOVE` non rende atomici i campi Java aggiornati prima o dopo lo spostamento. Quando un'operazione richiede entrambe le cose, occorre progettare esplicitamente il coordinamento e la gestione del fallimento fra i due mondi.

Esplorare directory senza trattenere risorse

`Files.createDirectories` crea le directory mancanti di un percorso, mentre `createDirectory` richiede che il padre esista già. `Files.list` restituisce uno `Stream<Path>` che mantiene aperta la directory: va chiuso con `try-with-resources`, anche se il tipo somiglia agli stream del capitolo 18. `Files.walk` visita più livelli e richiede la stessa disciplina. `DirectoryStream` è un'altra scelta per iterare una directory senza costruire subito una lista. Un `FileVisitor` rende espliciti i momenti in cui entriamo in una directory, vediamo un file, incontriamo un errore e usciamo: serve soprattutto quando copia, pulizia o ricerca devono reagire in modo diverso a ciascun evento.

Per eliminare un albero non basta chiamare `Files.delete` sulla sua radice non vuota. Occorre una visita in cui i figli vengano trattati prima della directory. Una cancellazione può fallire dopo avere già rimosso alcuni figli: il programma deve decidere se continuare, fermarsi o registrare i risultati parziali. Lo stesso vale per una copia o uno spostamento che attraversa più file. `Files.copy` e `Files.move` operano su percorsi indicati; non sono una transazione generale sull'intero albero. I collegamenti simbolici richiedono una scelta esplicita: copiarli come collegamenti o seguirli cambia sia il contenuto sia i confini attraversati.

Per una semplice ricerca locale possiamo scrivere un frammento che chiude lo stream anche se il filtro lancia un'eccezione:

```
try (var percorsi = Files.list(cartella)) {
    long testi = percorsi.filter(p -> p.getFileName().toString()
            .endsWith(".txt")).count();
    System.out.println("File di testo: " + testi);
}
```

Questo frammento assume che `cartella` sia un `Path` già definito. `Files.list` osserva soltanto i figli diretti, non l'intero albero; `Files.walk` cambierebbe il problema e il costo della visita. Il filtro sui nomi non dimostra che un file sia

davvero testo: un'estensione è una convenzione, non una proprietà verificata del contenuto. Se un altro processo crea o rimuove file mentre iteriamo, non dobbiamo trattare il risultato come una fotografia atomica della directory.

I file temporanei, gli attributi e i permessi sono altre domande sul confine. `Files.createTempFile` lascia scegliere una directory e crea un nome non già occupato; i permessi iniziali e gli attributi disponibili dipendono dal provider. Una vista POSIX dei permessi non è garantita su ogni sistema. `Files.readAttributes` e metodi come `size` forniscono una fotografia: il file può cambiare dopo la lettura. Non confondiamo una misura osservata con una prenotazione della risorsa.

Percorsi, collegamenti e visite: scegliere che cosa stiamo attraversando

La biblioteca riceve il nome relativo `ingressi/../catalogo.txt`. `resolve` e `normalize` aiutano a costruire e semplificare il nome. Se la parte da risolvere è assoluta, però, `resolve` restituisce quella parte: non la costringe dentro la directory di base. Se il nome proviene dall'esterno, il requisito «restare nella cartella autorizzata» richiede controlli ulteriori. Il confronto lessicale dopo la normalizzazione è un primo controllo, ma non gestisce da solo i collegamenti simbolici o una sostituzione concorrente del percorso.

Un collegamento simbolico contiene un riferimento a un altro percorso. Il nome che osserviamo e il file raggiunto possono quindi appartenere a luoghi diversi. `NOFOLLOW_LINKS` modifica il comportamento delle operazioni che lo accettano; non è una modalità globale attivata una volta per tutto il programma. Per una copia decidiamo se interessa il collegamento oppure il suo contenuto. Per una visita ricorsiva `Files.walk` non segue di default i collegamenti simbolici; abilitare `FOLLOW_LINKS` introduce anche il problema dei cicli e dei confini attraversati.

> **Concetto chiave – Sicurezza e portabilità sono contratti.** Un `Path` appartiene a un provider di file system, che traduce le operazioni Java sul sistema concreto. Radici, sensibilità alle maiuscole, separatori e supporto degli attributi possono differire. Un percorso valido non è un'autorizzazione; un percorso normalizzato non è una prenotazione del file. Per operazioni sensibili su directory modificabili da altri attori servono una progettazione dei permessi e, quando disponibile, strumenti come `SecureDirectoryStream`, che operano rispetto a una directory aperta.

Per la scansione del catalogo, `walkFileTree` con un `SimpleFileVisitor` permette di dichiarare che cosa fare in `visitFile`, `visitFileFailed` e `postVisitDirectory`. Quest'ultimo riceve anche l'eventuale errore di visita: cancellare senza controllarlo può nascondere il motivo per cui non abbiamo completato l'albero. I risultati `CONTINUE`, `SKIP_SUBTREE` e `TERMINATE` esprimono una politica, non la riparazione di un errore. Una visita può interrompersi dopo effetti già prodotti; conserviamo un resoconto quando quei risultati devono essere recuperabili.

`Files.lines` restituisce uno stream testuale lazy che va chiuso. Un errore può emergere durante il consumo come `UncheckedIOException`, non soltanto nell'apertura come `IOException`. Una pipeline del capitolo 18 non rende automaticamente pura o priva di risorse la sorgente. La variante utile è un file che scompare o diventa illeggibile durante la scansione: prima scegliamo se il rapporto

deve essere completo oppure parziale e dichiarato, poi scriviamo la gestione dell'errore.

Leggere una risorsa impacchettata

Un'immagine distribuita dentro un JAR non è automaticamente un `Path` del file system ordinario. `Class.getResource` risolve un nome assoluto con `/` dalla radice del class path o del modulo, oppure un nome relativo al package della classe; `ClassLoader.getResource` usa nomi senza `/` iniziale e ha regole di ricerca diverse. Se serve leggere una risorsa impacchettata, `getResourceAsStream` evita di trasformare arbitrariamente il suo URL in un percorso. Il capitolo 21 sui classloader spiega perché la posizione fisica di una classe e delle sue risorse non va presunta.

Immaginiamo un file `config/esempio.txt` distribuito accanto alle classi. `ImportaArchivio.class.getResourceAsStream("/config/esempio.txt")` chiede il nome dalla radice e restituisce uno stream o `null` se la risorsa non è trovata. Prima di passarla a un `InputStreamReader`, controlliamo il caso assente e dichiariamo UTF-8 se il contenuto è testo. Lo stream va chiuso. Se il file deve essere modificato dall'utente, non deve invece essere trattato come risorsa interna immutabile: scegliamo un `Path` esterno configurabile. La prova di trasferimento è impacchettare davvero la risorsa in un JAR e ripetere la lettura senza assumere che il JAR sia stato estratto su disco.

Canali e stato di un buffer

Un `Channel` offre un'altra interfaccia per dati e dispositivi. `ByteBuffer` tiene distinti capacità, posizione e limite: la capacità è lo spazio del buffer, la posizione indica dove avverrà la prossima lettura o scrittura e il limite delimita la parte attiva. `flip()` prepara alla lettura i byte appena scritti nel buffer; `clear()` prepara a riempirlo di nuovo senza cancellare fisicamente i byte. È un modello utile per I/O avanzato, ma un archivio testuale lineare non richiede di sostituire automaticamente un `BufferedReader` con canali e buffer.

Immaginiamo un buffer di capacità otto. Dopo aver scritto tre byte, la posizione vale tre e il limite resta otto. `flip()` porta il limite a tre e la posizione a zero: una lettura successiva vede soltanto i tre byte validi. Se saltassimo `flip()`, tenteremmo di leggere dalla posizione tre fino al limite otto, cioè dalla parte non preparata per la lettura. Se chiamassimo `clear()` pensando che azzeri i dati, potremmo invece osservare ancora i vecchi byte attraverso una vista inappropriata. Questi numeri sono un modello di stato, non una raccomandazione di usare un buffer di otto byte in produzione.

Dai byte al canale: una copia con stato esplicito

Se il requisito è raggiungere una posizione precisa del file o integrare un'elaborazione già basata su buffer, `FileChannel` può essere adatto. Manteniamo separati lo stato del canale, che comprende la posizione nel file, e quello del `ByteBuffer`, che comprende la regione attiva in memoria. Il canale riempie il buffer; `flip()` espone i dati appena ottenuti; la scrittura consuma quella regione; `clear()` prepara il giro seguente.

Nel programma `ContrattiIO` copiamo undici byte con un buffer di capacità quattro. Avremo due blocchi pieni e uno parziale. La scrittura viene ripetuta finché `hasRemaining()` diventa falso: una singola `write` non promette di consumare tutto il buffer. Per questo esempio usiamo canali di file ordinari; con un canale non bloccante il ritorno di zero richiederebbe una strategia di attesa, altrimenti un ciclo potrebbe occupare la CPU senza avanzare.

Il nucleo della copia rende visibili i quattro passaggi. Qui `origine` e `destinazione` sono `FileChannel` aperti nel `try-with-resources` del programma completo; `ByteBuffer` appartiene a `java.nio`.

```
ByteBuffer buffer = ByteBuffer.allocate(4);
while (origine.read(buffer) != -1) {
    buffer.flip();
    while (buffer.hasRemaining()) destinazione.write(buffer);
    buffer.clear();
}
```

Nell'ultimo giro vengono letti tre byte: `flip()` limita a tre la regione da scrivere e impedisce di ricopiare il quarto byte rimasto dal giro precedente. Il confronto di tutti i byte fra ingresso e uscita verifica questo dettaglio, oltre alla lunghezza finale.

Il seguente schema mostra il primo giro, con quattro byte validi. Non cancelliamo i dati quando cambiamo fase: cambiamo gli indici che autorizzano a leggerli o scriverli.

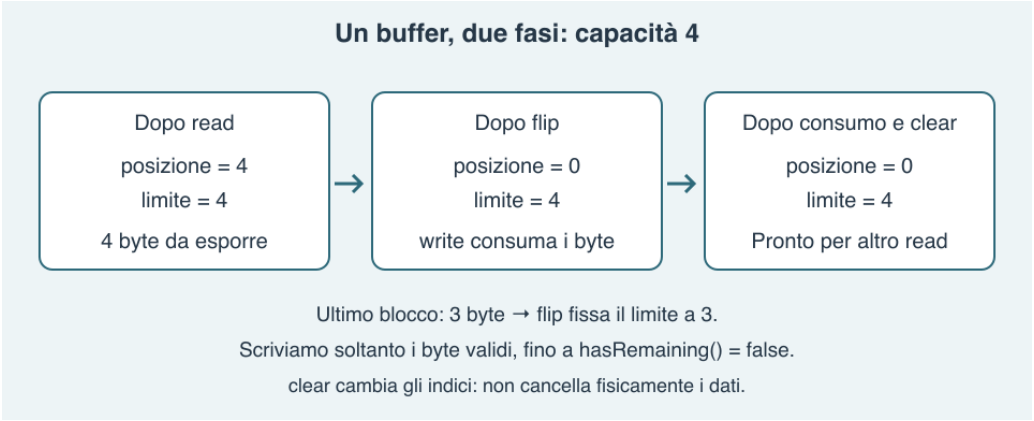


Figura 25.3 – Posizione e limite delimitano i byte validi durante la copia.

`compact()` serve a un problema diverso: abbiamo consumato soltanto parte dei dati e dobbiamo conservare quelli residui per completare, per esempio, un messaggio. Li sposta all'inizio e prepara spazio per altro ingresso. Usare `clear()` in quel punto perderebbe la regione residua dal nostro protocollo di lettura. `rewind()` permette invece di rileggere la regione attiva conservandone il limite. La verifica consiste nel seguire posizione e limite dopo ogni passaggio, non nell'imparare a memoria tre nomi simili.

> **Approfondimento – NIO non significa sempre non bloccante.** Un `FileChannel` non diventa selezionabile solo perché appartiene a NIO. `Selector` coordina canali selezionabili, come alcuni canali di rete, registrati in modalità non bloccante. `AsynchronousFileChannel` propone un altro contratto, basato sul completamento di operazioni: il buffer non va riutilizzato mentre l'operazione che lo usa è ancora in corso. Un buffer diretto o un file mappato in memoria può essere utile in specifici carichi, ma comporta costi e un ciclo di vita diversi dal semplice array. Scegliamoli dopo una misura e una necessità concreta.

Se leggiamo numeri con `ByteBuffer.getInt`, il formato deve stabilire l'ordine dei byte e il buffer deve avere abbastanza byte residui. Il fatto che un intero Java sia lo stesso valore su due macchine non significa che ogni protocollo lo codifichi nello stesso ordine. La copia del nostro esempio evita questo problema perché conserva byte senza interpretarli. La variante è un formato binario con intestazione: prima definiamo lunghezza e ordine, poi aggiungiamo la lettura dei campi.

Osservare una directory e rileggere lo stato

`WatchService` può segnalare cambiamenti a una directory. È utile per aggiornare una vista o avviare una nuova scansione, non per trattare ogni notifica come una registrazione affidabile e completa: eventi possono essere accorpati, persi o consegnati con semantiche dipendenti dal sistema. Quando il servizio segnala overflow, una strategia sensata è riconciliare lo stato effettivo della directory. Il medesimo principio visto nel capitolo 20 torna qui: la notifica invita a verificare una condizione, non sostituisce la condizione.

Supponiamo che la biblioteca aggiorni l'archivio quando un altro programma crea un file in una cartella. Una scansione periodica potrebbe essere sufficiente; `WatchService` diventa interessante quando vogliamo reagire presto. Registriamo la directory per gli eventi pertinenti, poi un ciclo legge le chiavi e i relativi eventi. Il nome riportato in un evento è relativo alla directory registrata: si risolve su quella directory prima di leggere il file. Dopo aver consumato gli eventi si reimposta la chiave, controllando se è ancora valida. Se gli eventi vengono persi, la soluzione corretta è una nuova scansione dello stato, non assumere che il flusso di notifiche ricostruisca da solo tutto ciò che è accaduto. Il polling può essere più semplice quando la tempestività non è critica; questa è una scelta guidata dal problema.

Notifiche, completamento e verifica dello stato

Un evento di creazione non garantisce che chi scrive abbia finito il file. Se importiamo immediatamente, possiamo leggere un archivio ancora incompleto. Un protocollo utile fa scrivere al produttore un provvisorio e pubblicare il nome definitivo soltanto al completamento; il consumatore rilegge lo stato e applica comunque la validazione. Il `WatchService` segnala che vale la pena controllare, mentre il formato e la pubblicazione stabiliscono che cosa accettare.

Nel ciclo, una `WatchKey` raccoglie eventi; dopo `pollEvents()` chiamiamo `reset()` e, se restituisce `false`, la registrazione non è più valida. `OVERFLOW` richiede una scansione di riconciliazione. Registrare una directory non registra automaticamente tutti i suoi discendenti. La chiusura del servizio e l'interruzione

del thread che attende una chiave fanno parte dello spegnimento; non confondiamo un arresto richiesto con un guasto da riprovare all'infinito.

La prova di trasferimento crea rapidamente più file, li rinomina e poi confronta il catalogo con una scansione completa. Non imponiamo come risultato un certo numero di notifiche: verifichiamo che la riconciliazione arrivi allo stato corretto. Se il sistema richiede una storia completa e ordinata di tutte le modifiche, questo servizio non è il registro degli eventi necessario. Occorre un protocollo applicativo che conservi quella storia.

Serializzazione storica e formati esterni

La serializzazione nativa di oggetti Java appartiene al patrimonio di sistemi esistenti, ma lega il formato alle classi e apre problemi di compatibilità e sicurezza. Per un nuovo archivio esterno scegliamo un formato con contratto esplicito e un parser adatto; Java SE non fornisce un unico parser generale per JSON, CSV e XML. Il file dell'esempio è volutamente più semplice e dichiara da sé la sua piccola grammatica.

Immaginiamo che un collega proponga di sostituire il file di titoli con un `ObjectOutputStream`: salverebbe direttamente gli oggetti del catalogo e sembrerebbe evitare il lavoro di definire le righe. Il vantaggio iniziale ha un costo. Quando cambiano le classi, i dati salvati in precedenza possono non essere più leggibili secondo il contratto atteso; quando il file proviene dall'esterno, `ObjectInputStream` ricostruisce grafi di oggetti e non va trattato come un parser innocuo per input non fidati. Il filtro della deserializzazione documentato per Java 25 può restringere classi e dimensioni ammesse in un sistema legacy, ma non crea da solo un formato pubblico semplice e stabile.

Se il catalogo deve essere scambiato con un altro programma, definiamo prima i dati necessari, la codifica, i limiti di dimensione, la versione del formato e la risposta ai campi non validi. Scegliamo poi un parser che rispetti quel contratto e proviamo sia un file valido sia uno troncato. Se invece dobbiamo mantenere un archivio Java già distribuito, partiamo dall'inventario delle classi realmente presenti e da una migrazione controllata; non lo convertiamo alla cieca. Il mattone riusabile è riconoscere **chi leggerà il file e per quanto tempo**: questa domanda guida il formato più del numero di righe necessario a scrivere il primo esempio.

Verificare il mattone e trasferirlo

Cambiamo il problema: i titoli devono essere unici, l'input può contenere milioni di righe e il file finale non deve cambiare se una riga è malformata. La lettura progressiva resta adatta alla memoria disponibile; occorre però decidere come riconoscere i duplicati, con una `Set` in memoria se la dimensione è gestibile oppure con una strategia esterna se non lo è. La pubblicazione del provvisorio rimane l'ultimo passo. Se il consumatore deve vedere *tutti* i titoli o *nessuno*, una scrittura diretta perde la proprietà richiesta anche se ogni riga è corretta.

Una seconda variante sostituisce l'archivio locale con una risorsa nel JAR. Ora non possiamo pubblicarla con `Files.move`: dobbiamo distinguere la risorsa di sola lettura impacchettata dal file di lavoro esterno. Saper formulare questa differenza è la prova del capitolo: associare il problema alla corretta astrazione, riconoscere il limite e cambiare soluzione quando cambia il confine.

Riferimenti essenziali. La specifica `Files` di Java 25 descrive operazioni, opzioni e possibili errori del file system; non garantisce che ogni provider offra le stesse operazioni atomiche. La documentazione dei charset definisce la conversione fra byte e caratteri. Queste fonti sostengono i contratti delle API; la politica dell'archivio, inclusa la scelta di fallire senza ripiego, appartiene al nostro esempio.

Riferimenti per gli approfondimenti. InputStream definisce letture parziali e trasferimenti; CharsetDecoder rende esplicita la politica sugli errori. ByteBuffer descrive gli indici del buffer e FileChannel il contratto dei canali di file. Questi riferimenti permettono di controllare i confini dei piccoli esperimenti, senza trasformare un risultato locale in una garanzia su ogni dispositivo.

26. Tempo, localizzazione e numeri esatti

Una scadenza e un pagamento non sono lo stesso genere di dato

Una biblioteca può dire che un prestito scade il 1° novembre. Un registro di pagamenti, invece, deve dire in quale istante è arrivata una conferma. Le due informazioni sembrano «date», ma rispondono a domande diverse. Se memorizziamo entrambe come una stringa, il programma non sa quali operazioni hanno senso. Se le memorizziamo entrambe come millisecondi dall'epoch, costringiamo una scadenza civile a fingere di essere un istante. La scelta del tipo è il primo mattone del capitolo: descrivere il significato dell'informazione prima di calcolarla.

L'epoch di Java è il punto di riferimento `1970-01-01T00:00:00Z` sulla linea del tempo. `Instant` rappresenta un punto su quella linea; è adatto a registrare quando un evento è accaduto e a confrontare due eventi. `LocalDate` rappresenta una data di calendario senza ora e senza fuso. `LocalTime` rappresenta un'ora civile senza data; `LocalDateTime` unisce data e ora civili, ma ancora non individua necessariamente un solo istante. Nella figura 26.1 la domanda del dominio conduce al tipo, non a un elenco di classi da memorizzare.



Un `ZoneOffset` *come* `+02:00` *è una distanza da UTC in un momento. Un* `ZoneId` *come* `Europe/Rome` *identifica invece regole di un'area geografica, comprese transizioni che cambiano l'offset.* `OffsetDateTime` *lega una data e ora civile a un offset;* `ZonedDateTime` *le lega a una zona con regole. Se dobbiamo mostrare al lettore l'ora locale di un evento registrato come* `Instant`*, applichiamo una zona quando produciamo la vista:* `evento.atZone(zona)`*. La zona scelta fa parte della richiesta di visualizzazione e non cambia l'istante registrato.*

> **Concetto chiave – Un valore locale non è incompleto per definizione.** «La biblioteca chiude alle 18:00» è naturalmente un orario civile; «il pagamento è arrivato alle 18:00» non basta a confrontarlo con un pagamento avvenuto altrove. In un caso il fuso può essere una proprietà del luogo, nell'altro dobbiamo registrare un istante o informazioni sufficienti per ricostruirlo. La completezza di un tipo dipende dalla domanda che l'applicazione deve risolvere.

Ore di calendario e durate trascorse

`Duration` misura una quantità di tempo sulla linea degli istanti, per esempio novanta minuti effettivamente trascorsi. `Period` esprime invece anni, mesi e giorni di calendario, per esempio «un mese dalla data di emissione». `ChronoUnit` consente di nominare unità in varie operazioni. La distinzione appare quando cambia l'ora legale: `plusDays(1)` su un valore con zona conserva l'ora civile e può attraversare ventitré o venticinque ore reali; `plus(Duration.ofHours(24))` conserva ventiquattro ore trascorse e può cambiare l'ora civile mostrata. Nessuna delle due operazioni è universalmente corretta. Per una scadenza «domani alla stessa ora locale» serve la regola civile; per un timer di ventiquattro ore serve la durata.

Una data e ora locale può cadere in un *gap*, quando l'orologio salta avanti e quell'ora non esiste, oppure in una *sovrapposizione*, quando l'orologio torna indietro e la stessa ora locale compare due volte. `ZoneRules.getValidOffsets(localDateTime)` restituisce zero, uno o due offset validi. La figura 26.2 rappresenta i due casi. È un controllo da fare quando un utente sceglie una data e ora locale per un evento che deve avvenire in un preciso istante. Scegliere il primo offset disponibile senza mostrare la decisione all'utente può fissare l'appuntamento un'ora prima di quanto intendesse.



Nel programma di prova, `2026-10-25T02:30` *a Roma è una sovrapposizione. La consultazione delle regole restituisce* `+02:00` *e* `+01:00`*. Costruiamo due* `ZonedDateTime` *con la stessa ora civile ma con offset preferiti diversi; i loro* `Instant` *distano sessanta minuti. Il risultato non dipende dall'orologio corrente del computer. Le regole di fuso sono dati del runtime e possono essere aggiornate: per elaborazioni storiche o scadenze a lunga distanza si deve registrare la versione dei dati e la politica di risoluzione se la riproducibilità esatta è un requisito.*

Un orologio controllabile

`Instant.now()` è comodo, ma se lo chiamiamo nel cuore della logica di una fattura rendiamo la prova dipendente dal momento di esecuzione. `Clock` separa la lettura dell'ora dal calcolo. `Clock.fixed` fornisce un istante costante, utile per un esempio e per un test; `Clock.system(zone)` legge l'orologio di sistema. Il programma usa un clock fissato al 2 ottobre 2026 alle 10:00 UTC con zona di Roma. `LocalDate.now(prova)` produce la data civile di emissione, mentre `Instant.now(prova)` registra l'evento. La scadenza è `emissione.plusDays(30)`, che nell'esempio stampa `2026-11-01`. Se il contratto dicesse «trenta periodi di 24 ore», useremmo un istante e una durata invece di `LocalDate`.

Formattare non deve cambiare il valore. `DateTimeFormatter.ISO_LOCAL_DATE` produce una rappresentazione stabile di una data; un pattern personalizzato richiede attenzione a simboli che hanno significati diversi, per esempio anno di calendario e anno di settimana. Il parsing di input esterni dovrebbe usare un formato dichiarato e gestire `DateTimeParseException` mostrando il campo non valido. Il lettore deve distinguere `LocalDate.parse("2026-11-01")`, che interpreta un formato ISO, dal tentativo di usare la lingua italiana come regola generale per una data ambigua quale `01/11/26`.

Una data ricevuta come `31/02/2026` non è soltanto scritta in un formato insolito: non esiste nel calendario ISO. Il parser deve segnalarla. Per un input che dichiara giorno, mese e anno possiamo creare un `DateTimeFormatter` con il pattern previsto e una strategia di risoluzione rigorosa; non scegliamo silenziosamente il 3 marzo come «correzione» del 31 febbraio. Anche l'assenza del fuso va trattata come informazione mancante quando il risultato richiesto è un istante. In un'interfaccia la locale può determinare l'ordine dei campi mostrati, ma il contratto di scambio fra processi deve dichiarare un formato non ambiguo.

La prova di un calcolo temporale ha almeno due livelli. Un test unitario usa `Clock.fixed` per riprodurre la stessa data e ora. Un test di integrazione controlla che il fuso configurato per l'applicazione sia quello richiesto dal dominio, soprattutto quando il software gira su un server impostato in UTC ma serve utenti di città diverse. Se leggiamo implicitamente `ZoneId.systemDefault()` in un punto nascosto della logica, il test può passare sulla macchina dello sviluppatore e cambiare esito nel server. La zona va scelta o iniettata come dipendenza, non lasciata alla geografia del computer.

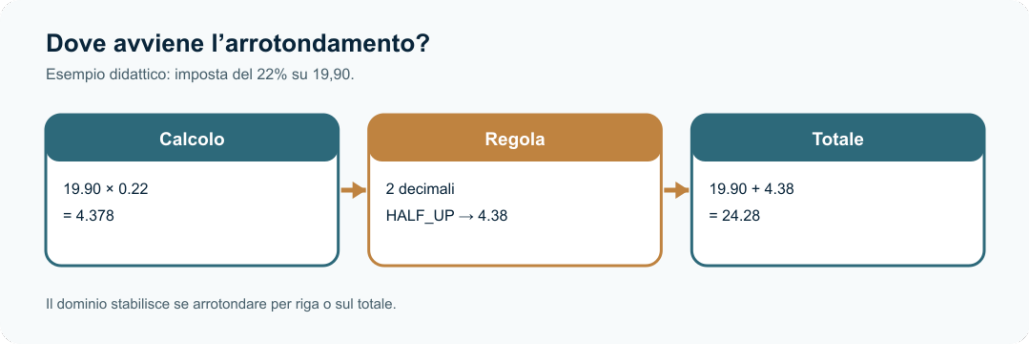
Perché un prezzo non è un `double`

Il tipo `double` rappresenta numeri in virgola mobile binaria. Molte frazioni decimali non hanno una rappresentazione binaria finita; questo rende `double` ottimo per numerosi calcoli scientifici ma inadatto come scelta predefinita per importi con regole decimali di arrotondamento. `BigDecimal` rappresenta un intero non scalato e una scala. In `19.90` l'intero non scalato è `1990` e la scala è `2`: il valore è 1990×10^{-2}. Il simbolo $\times$ indica moltiplicazione e 10^{-2} equivale a dividere per cento. Questo modello conserva esattamente il decimale scritto nella stringa.

La precisione decimale non risolve da sola la concorrenza. Se due richieste cambiano lo stesso saldo, una formula con `BigDecimal` può calcolare esattamente ciascun importo e perdere ugualmente un aggiornamento. Il capitolo 20A insegna a individuare la decisione atomica nella memoria della JVM; quando il saldo è conservato in un archivio, la coerenza dell'operazione deve essere garantita anche dalla transazione persistente. «Numero esatto» e «aggiornamento indivisibile» sono proprietà diverse, entrambe necessarie secondo il caso.

Costruiamo quindi `new BigDecimal("19.90")`. `new BigDecimal(0.1)` parte invece dall'approssimazione binaria già contenuta nel `double`: non recupera il decimale ideale `0.1`. `BigDecimal.valueOf(0.1)` usa la rappresentazione testuale canonica del `double` e può essere utile a un confine, ma quando il dato nasce come testo decimale il costruttore da stringa conserva direttamente l'intenzione. Per quantità monetarie occorre anche un'unità di valuta: il numero `19.90` da solo non dice se siano euro o dollari.

La figura 26.3 collega prezzo, aliquota e arrotondamento. Nel caso di studio `19.90 × 0.22` produce `4.378`: arrotondiamo l'imposta a due cifre decimali con `RoundingMode.HALF_UP`, ottenendo `4.38`; il totale è `24.28`. È una politica didattica dichiarata, non una regola fiscale universale. Se la legge o il contratto impongono arrotondamento per riga, per documento o per valuta, il punto in cui arrotondiamo cambia il risultato. `MathContext` limita invece la precisione complessiva di un'operazione: non è sinonimo di «due cifre dopo la virgola».



Una divisione decimale può avere un'espansione infinita, come `1 / 3`. `BigDecimal.ONE.divide(new BigDecimal("3"))` *senza una politica di arrotondamento lancia* `ArithmeticException`*; non restituisce silenziosamente un numero tagliato. La soluzione richiede una scala e un* `RoundingMode`*, oppure un* `MathContext` *appropriato al dominio.* `BigInteger` *risolve un problema diverso: interi che superano l'intervallo dei primitivi. Non gestisce da solo la virgola decimale; a parità di lavoro, oggetti e operazioni arbitrarie possono costare più dei primitivi.*

Per vedere la differenza fra scala e precisione, consideriamo `123.456`. La scala è tre perché ci sono tre cifre dopo il punto; la precisione è sei perché le cifre significative sono sei. `setScale(2, HALF_UP)` produce `123.46`. Un `MathContext` con precisione quattro produrrebbe `123.5`, cioè quattro cifre significative in totale. Dire «voglio due decimali» e usare una precisione di due è un errore concettuale: si ottiene un numero con due cifre significative, che può perdere anche la parte intera desiderata.

Anche `BigInteger` richiede una decisione motivata. Un contatore di prestiti che non supererà mai i limiti di `long` non trae vantaggio dal suo costo aggiuntivo; un identificatore numerico generato da una formula che cresce senza limite prefissato potrebbe richiederlo. `BigInteger` è immutabile: `totale.add(unita)` restituisce un nuovo oggetto e non modifica `totale`. La stessa regola vale per `BigDecimal`; dimenticare l'assegnazione del risultato lascia invariato il valore, proprio come accade con `String`.

Caso completo: data e importo della fattura

Il programma seguente riunisce i due confini. La data di emissione e la scadenza sono date civili. La registrazione è un istante. Il totale deriva da testo decimale e da una regola di arrotondamento esplicita. Compiliamo con `javac --release 25 FatturaTempo.java` ed eseguiamo con `java FatturaTempo`.

```
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.util.Locale;

public class FatturaTempo {
    public static void main(String[] args) {
        ZoneId zona = ZoneId.of("Europe/Rome");
        Clock prova = Clock.fixed(Instant.parse("2026-10-02T10:00:00Z"), zona);

        LocalDate emissione = LocalDate.now(prova);
        LocalDate scadenza = emissione.plusDays(30);
        System.out.println("Scadenza civile: " + scadenza);
        System.out.println("Registrazione UTC: " + Instant.now(prova));

        LocalDateTime sovrapposizione = LocalDateTime.of(2026, 10, 25, 2, 30);
        System.out.println("Offset possibili: " + zona.getRules()
                .getValidOffsets(sovrapposizione));
        ZonedDateTime prima = ZonedDateTime.ofLocal(
                sovrapposizione, zona, ZoneOffset.ofHours(2));
        ZonedDateTime seconda = ZonedDateTime.ofLocal(
                sovrapposizione, zona, ZoneOffset.ofHours(1));
        System.out.println("Istanti distinti: " +
                Duration.between(prima.toInstant(), seconda.toInstant()).toMinutes()
                + " minuti");

        BigDecimal prezzo = new BigDecimal("19.90");
        BigDecimal aliquota = new BigDecimal("0.22");
        BigDecimal imposta = prezzo.multiply(aliquota)
                .setScale(2, RoundingMode.HALF_UP);
        BigDecimal totale = prezzo.add(imposta);
        System.out.println("Totale: " + totale.toPlainString());
        System.out.println("Locale di stampa: " + Locale.ITALY);
        System.out.println("Stesso numero, scale diverse: "
                + (new BigDecimal("2.0").compareTo(new BigDecimal("2.00")) == 0));
    }
}
```

L'output atteso comprende `Scadenza civile: 2026-11-01`, `Istanti distinti: 60 minuti` e `Totale: 24.28`. L'esempio di sovrapposizione dimostra che la stessa data e ora locale non basta a identificare l'istante. L'ultima riga confronta numericamente `2.0` e `2.00`: `compareTo` restituisce zero, mentre `equals` restituirebbe `false` perché considera anche la scala. In una `HashSet` e in una `TreeSet` questa differenza può produrre comportamenti diversi di unicità; il capitolo 17 sulle collezioni permette di spiegare perché. Prima di usare `BigDecimal` come chiave, fissiamo la semantica richiesta dal dominio e normalizziamo se necessario.

Presentare i valori a persone diverse

`Locale` governa convenzioni di presentazione, come separatore decimale, raggruppamento delle cifre e nomi di date; non cambia il numero o l'istante. `NumberFormat` può formattare un importo per una locale, mentre `Currency` identifica una valuta ISO e alcune sue proprietà. Formattare `24.28` per un lettore italiano può produrre `24,28`; nel file di interscambio usiamo invece il formato concordato, non quello della macchina che esegue il programma. Per i testi traducibili occorrono risorse linguistiche separate; inserire le parole italiane dentro la logica di calcolo impedirebbe di cambiare lingua senza toccare il dominio.

Il simbolo mostrato non è un'identità sufficiente per la valuta: lo stesso simbolo può comparire in contesti diversi. Una fattura conserva un codice di valuta e un importo secondo le regole applicabili, poi decide come presentarli a un destinatario. `NumberFormat.getCurrencyInstance(locale)` fornisce una formattazione orientata alla locale; il programma deve anche impostare o conoscere la valuta effettiva del documento, invece di supporre che locale italiana significhi sempre euro. In un trasferimento dati non salviamo la stringa formattata come valore numerico da ricalcolare: essa incorpora separatori e simboli per persone, non un contratto aritmetico.

L'integrazione con sistemi esistenti può imporre `Date`, `Calendar` o `TimeZone`. Non sono il primo modello da insegnare a codice nuovo, ma bisogna conoscere il confine: `Date.toInstant()` recupera un istante; scegliere una `ZoneId` è un'altra decisione necessaria per mostrare la data civile. Migrare una data locale registrata come timestamp richiede capire con quale fuso fu creata, non soltanto cambiare il nome della classe.

Verifica e trasferimento

Immaginiamo ora una prenotazione alle `02:30` di una notte di cambio d'ora. Il programma deve chiedere se quell'ora esiste e, se corrisponde a due istanti, quale dei due desidera l'utente. Una semplice conversione che sceglie il comportamento predefinito dell'API può essere accettabile per una schermata informativa, ma non per una prenotazione da cui dipende un pagamento. Il problema indica il tipo e la verifica necessaria.

Supponiamo poi che tre righe di fattura abbiano imposte frazionarie. Calcolare l'imposta su ciascuna riga e sommare importi già arrotondati può dare un risultato diverso dal sommare basi imponibili e arrotondare una sola volta. Il lettore deve scrivere la regola del dominio, applicarla in un punto riconoscibile del codice e costruire un esempio in cui i due risultati divergano. Questa variante verifica che non abbia soltanto imparato a digitare `setScale(2, ...)`.

> **Riferimenti essenziali.** La documentazione Java 25 di `ZoneRules` definisce i casi con zero, uno e due offset; non decide per l'applicazione quale scegliere. La specifica di `BigDecimal` definisce rappresentazione, scala, divisione, confronto e arrotondamenti; la regola economica applicabile deve essere fornita dal dominio.

27. Rete e HTTP: chiedere dati a un altro processo

Il catalogo non abita più nello stesso programma

Il catalogo della biblioteca può essere gestito da un servizio remoto. Finché i libri erano in una `Map` locale, cercare un codice significava eseguire un metodo nello stesso processo. Attraverso la rete compaiono nuove possibilità: il server può non rispondere, la risposta può arrivare lentamente, un intermediario può restituire un errore e il testo ricevuto può non avere il formato atteso. La domanda del capitolo è come ottenere un dato senza confondere «nessun libro» con «non sono riuscito a chiedere».

Un *host* è un nodo raggiungibile tramite un nome o un indirizzo; il DNS può tradurre un nome in un indirizzo di rete. Una *porta* distingue servizi sul nodo. TCP fornisce un flusso ordinato di byte fra due estremità; UDP trasporta datagrammi e non promette da solo consegna, ordine o assenza di duplicati. HTTP è un protocollo applicativo sopra un trasporto: definisce richieste, risposte, metodi, status e header. Il corpo di una risposta può essere testo, byte di un'immagine o un formato come JSON. La figura 27.1 separa questi livelli, perché un errore DNS, un timeout TCP, un codice HTTP `404` e un JSON malformato richiedono diagnosi diverse.



Un `URI` identifica una risorsa con uno schema e altre parti, per esempio `https://esempio.invalid/libri/42`. Un URL è una forma di riferimento che indica anche un meccanismo di accesso, ma nel client moderno partiamo dalla rappresentazione `URI`. Non costruiamo componenti arbitrari concatenando stringhe fornite da un utente: spazi, `?`, `#`, slash e caratteri non ASCII hanno significati o regole di codifica. Una query va costruita codificando i valori dei parametri secondo il contratto del servizio, senza applicare una codifica indistinta all'intero URI. La risoluzione di un riferimento relativo con `URI.resolve` è un'operazione semantica, non una semplice giustapposizione di testo.

Per esempio, la chiave libro `Java & Reti` non può essere aggiunta senza preparazione a `...?titolo=`. Il carattere `&` separerebbe parametri, cambiando la domanda inviata al server. Al contrario, codificare un URI completo come un unico valore trasformerebbe anche i suoi delimitatori. Prima si determina in quale componente deve entrare il dato, poi si applicano le regole di codifica di quel componente. La rappresentazione testuale finale si può stampare in un test, evitando però di registrare query che contengono credenziali o dati personali.

Un primo socket e il suo limite

Un `Socket` TCP collega il processo a host e porta e offre flussi di ingresso e uscita, che abbiamo imparato a gestire nel capitolo 25. Se inventiamo un protocollo a righe, dobbiamo stabilire dove finisce una richiesta, quale charset si usa, come si distinguono un errore e un valore e che cosa succede se il peer chiude a metà. Il socket non interpreta HTTP per noi. `DatagramSocket` per UDP è adatto ad altri protocolli, ma la sua assenza di connessione non significa automaticamente «più veloce per qualunque servizio». Per un catalogo HTTP useremo l'API che esprime già richieste e risposte HTTP.

Client, richiesta, risposta

`HttpClient` è l'oggetto configurato che può servire più richieste e riusare risorse come le connessioni. Un `HttpRequest` descrive una richiesta specifica: URI, metodo, header, corpo eventuale e timeout. `HttpResponse<T>` contiene status, header e un corpo di tipo `T` determinato dal `BodyHandler` scelto. La scelta di `BodyHandlers.ofString(StandardCharsets.UTF_8)` dichiara che il corpo va letto come testo UTF-8; `ofByteArray` conserva i byte, `ofFile` scrive su file e `ofInputStream` lascia al chiamante uno stream da consumare o chiudere. Queste forme non cambiano il significato del codice HTTP.

`GET` chiede una rappresentazione della risorsa; una richiesta con corpo usa un `BodyPublisher`, per esempio `BodyPublishers.ofString` quando il protocollo applicativo accetta testo. L'header `Content-Type` descrive il formato mandato o ricevuto, ma non convalida automaticamente il corpo. Se il server dichiara JSON e invia testo non valido, il trasporto può aver funzionato perfettamente mentre l'interpretazione applicativa fallisce. Il client dovrebbe preservare in diagnosi almeno URI senza segreti, metodo, status e causa del parsing; registrare indiscriminatamente l'intero corpo può invece esporre dati riservati o saturare i log.

Nel programma completo avviamo un piccolo server soltanto sull'indirizzo di loopback della macchina, chiediamo `/libri` e stampiamo risposta e corpo. Il server didattico usa `jdk.httpserver`, un modulo del JDK distinto dalla specifica Java SE generale e non un framework per un server di produzione. Si compila con `javac --release 25 --add-modules jdk.httpserver ClientLocale.java` e si esegue con `java --add-modules jdk.httpserver ClientLocale`. La porta `0` fa scegliere al sistema una porta libera: non occorre fissare una porta che potrebbe essere occupata. Il client è chiuso al termine del `try`, il server nel `finally`.

```
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.time.Duration;

public class ClientLocale {
    public static void main(String[] args) throws IOException, InterruptedException {
        HttpServer server = HttpServer.create(
                new InetSocketAddress(InetAddress.getByName("127.0.0.1"), 0), 0);
        server.createContext("/libri", scambio -> {

            byte[] corpo = "libro=Java".getBytes(StandardCharsets.UTF_8);
            scambio.getResponseHeaders().set("Content-Type", "text/plain; charset=utf-8");
            scambio.sendResponseHeaders(200, corpo.length);
            try (var uscita = scambio.getResponseBody()) {
                uscita.write(corpo);
            }
        });
        server.start();
        try (HttpClient client = HttpClient.newBuilder()
                .connectTimeout(Duration.ofSeconds(2)).build()) {
            URI uri = URI.create("http://127.0.0.1:" + server.getAddress().getPort()
                    + "/libri");
            HttpRequest richiesta = HttpRequest.newBuilder(uri)
                    .timeout(Duration.ofSeconds(3)).GET().build();
            HttpResponse<String> risposta = client.send(richiesta,
                    HttpResponse.BodyHandlers.ofString(StandardCharsets.UTF_8));

            System.out.println("HTTP " + risposta.statusCode());
            System.out.println(risposta.body());
        } finally {
            server.stop(0);
        }
    }
}
```

L'output atteso è `HTTP 200` seguito da `libro=Java`. Il corpo non viene letto direttamente dallo stream del socket: il `BodyHandler` lo converte nel tipo richiesto. Se il servizio rispondesse `404`, `send` restituirebbe comunque una risposta HTTP da esaminare; uno status non è di per sé una `IOException`. Se invece il nome host non si risolve o la connessione fallisce, non c'è una risposta HTTP da interpretare. Questa distinzione impedisce di trasformare ogni guasto in `Optional.empty()` come se il libro semplicemente non esistesse.

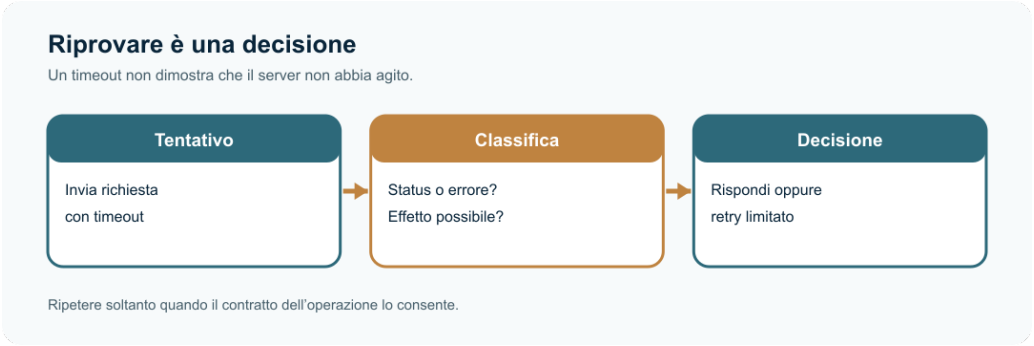
> **Concetto chiave – Uno status è un fatto del protocollo.** Il server può rispondere `200` con un corpo inatteso, `404` per risorsa assente, `429` per un limite di richieste o `503` per indisponibilità temporanea. Il programma deve leggere il contratto del servizio e decidere quali status sono risultati normali e quali richiedono gestione dell'errore. Il metodo `BodyHandlers.ofString` non rifiuta automaticamente i corpi degli status non riusciti.

La configurazione di `HttpClient` comprende una preferenza per HTTP/1.1 o HTTP/2, redirect, proxy e autenticazione. Si configura ciò che serve al sistema; una preferenza di versione non garantisce che il peer userà proprio quella versione. Il timeout di connessione riguarda la fase in cui si stabilisce il collegamento, mentre `HttpRequest.timeout` limita la durata dell'operazione di risposta secondo il contratto dell'API. Sono limiti diversi. Un timeout di dieci secondi non prova che il server non abbia elaborato una richiesta: il client può perdere la risposta dopo che il server ha già compiuto il lavoro.

Attese, errori e retry

`send` attende il risultato e può lanciare `IOException` o `InterruptedException`. Se il thread viene interrotto, il chiamante deve rispettare la politica di interruzione del proprio contesto; un metodo che cattura l'eccezione senza poterla rilanciare dovrebbe almeno ripristinare il flag. `sendAsync` restituisce un `CompletableFuture<HttpResponse<T>>`: permette di comporre lavoro asincrono e fa emergere il fallimento tramite il completamento eccezionale. Non trasforma la rete in un'operazione immediata né esonera dal consumare correttamente un corpo a flusso. Nel capitolo 20 abbiamo visto che un thread virtuale consente di mantenere codice bloccante lineare mentre molte richieste aspettano I/O; resta necessario limitare richieste concorrenti, memoria e capacità del servizio remoto. La figura 27.2 mostra che la decisione di riprovare viene dopo la classificazione del fallimento.

Il capitolo 20A ha mostrato che un CAS locale può far riuscire una sola prenotazione nella memoria della JVM. Se la prenotazione diventa una richiesta HTTP verso un archivio remoto, quel CAS non governa lo stato del server. Dopo un timeout il client potrebbe non sapere se il server abbia completato l'operazione: ritentare senza un contratto di idempotenza o una chiave di richiesta concordata può duplicarla. La garanzia va costruita nel protocollo e nell'archivio che riceve la richiesta.



Ripetere una richiesta può essere utile dopo un errore temporaneo, ma non è una cura generale. Un'operazione idempotente produce lo stesso effetto desiderato se ripetuta nelle stesse condizioni; ciò non significa che ogni tentativo abbia identici log o tempi. Una `GET` *che interroga dati è spesso ripetibile secondo il contratto HTTP, mentre un* `POST` *che crea una fattura può duplicare un effetto se il client non ha ricevuto la prima risposta. Anche quando il metodo è teoricamente idempotente, l'implementazione concreta del servizio conta. Un retry ha un numero massimo di tentativi, attese controllate, eventualmente jitter per evitare sincronizzazione dei client, e un criterio sugli status e sugli errori. Deve essere osservabile; un ciclo infinito che nasconde l'errore aumenta il carico proprio quando il servizio è in difficoltà.*

Con *jitter* intendiamo una piccola variazione casuale dell'attesa fra tentativi. Se mille client attendono tutti esattamente un secondo dopo un `503`, possono tornare insieme e sovraccaricare di nuovo il server; distribuire le attese riduce questa concentrazione. La variazione deve rispettare un limite e non sostituisce il numero massimo di tentativi. Se il server comunica un tempo di attesa con un header come `Retry-After`, il contratto HTTP e quello del servizio guidano la decisione. Un retry automatico deve anche poter essere annullato quando il chiamante non è più interessato al risultato.

Una cancellazione di `CompletableFuture` può interrompere il lavoro in corso in modo dipendente dall'implementazione e dal punto dell'operazione. Non equivale a «il server non ha eseguito». Per una modifica remota occorre un protocollo che permetta di correlare tentativi, per esempio una chiave di idempotenza quando il servizio la supporta. La gestione di un timeout deve conservare questa incertezza, anziché dichiarare un fallimento della modifica senza prove.

HTTPS e identità del peer

HTTPS usa TLS per proteggere il trasporto e verificare l'identità del peer secondo la configurazione di fiducia. Un certificato non fidato è un segnale da diagnosticare, non un invito a disabilitare tutte le verifiche. L'applicazione deve sapere da dove proviene il materiale di fiducia e come vengono aggiornate le credenziali. Un `Authenticator` e un `CookieHandler` possono far parte della configurazione del client; per protocolli di identità più complessi serve un contratto specifico. Le password non vanno inserite nell'URI, nei log o negli esempi distribuiti.

Se il server presenta un certificato valido per un nome diverso dall'host richiesto, il problema non riguarda il parsing del corpo né il codice HTTP: il client non ha ancora una risposta applicativa affidabile. Un sistema che usa una propria autorità

di certificazione deve distribuirne la fiducia con un processo controllato; accettare qualunque certificato rende inutile la verifica dell'identità. L'autenticazione dell'utente è una domanda successiva e diversa: un canale HTTPS può essere protetto anche quando la richiesta non contiene ancora credenziali applicative.

Il formato e la dimensione del corpo

Il corpo è un secondo confine. Il nostro server restituisce testo semplice `libro=Java`; se restituisse JSON, `HttpClient` consegnerebbe stringhe o byte, non un oggetto del dominio già validato. Java SE non impone un parser JSON generale. Serve una libreria dichiarata, una grammatica e una validazione dei campi. Prima di scrivere una risposta su disco, definiamo anche un limite di dimensione: `ofString` e `ofByteArray` portano il corpo in memoria. Per contenuti grandi scegliamo un handler a file o a flusso, gestendo chiusura e pubblicazione come nel capitolo 25.

Il metodo `ofFile` è comodo quando il corpo può essere scritto direttamente nella destinazione scelta. Se il file rappresenta un archivio che altri processi leggono, la comodità non sostituisce la pubblicazione controllata: si può scaricare in un file provvisorio, convalidarne dimensione e contenuto, poi decidere se sostituire il file visibile. Questo è lo stesso mattone del capitolo 25 applicato a un'origine remota. Se la risposta viene consumata con `ofInputStream`, la chiusura del flusso è responsabilità del chiamante; lasciarlo aperto può impedire il rilascio delle risorse e ritardare la chiusura ordinata del client.

Quando servono messaggi continui

`WebSocket` mantiene uno scambio di messaggi in entrambe le direzioni dopo un'apertura specifica. Un listener riceve eventi e deve rispettare la richiesta dei messaggi successivi e la pressione del consumatore; non è una variante con un nome diverso di `HttpClient.send`. È adatto quando il problema richiede aggiornamenti continui, per esempio lo stato in tempo reale di un catalogo, e comporta un protocollo di applicazione per connessione, riconnessione e chiusura.

Se la schermata chiede soltanto il titolo di un libro quando l'utente preme «Cerca», la richiesta HTTP ordinaria ha un inizio e una fine naturali. Se invece la schermata deve ricevere un flusso di cambiamenti del catalogo, un WebSocket può evitare interrogazioni ripetute; occorre però definire che cosa fare dopo una disconnessione. Il server invierà una fotografia completa, oppure eventi dal numero progressivo successivo all'ultimo ricevuto? Senza questa regola, una riconnessione può mostrare un catalogo incoerente anche se il canale di trasporto funziona.

Trasferire la soluzione al servizio reale

Il server locale garantisce un caso positivo controllato. Modifichiamolo prima per rispondere `404`, poi `503` una sola volta e infine `200`. Il client deve distinguere «libro assente» da «servizio temporaneamente indisponibile» e riprovare soltanto se il contratto consente di ripetere la richiesta. Registriamo quante chiamate sono state fatte; un test che guarda solo il corpo finale non vede un ciclo di retry eccessivo. Un secondo esperimento ritarda la risposta oltre il timeout e verifica quale informazione il client possiede davvero sull'eventuale lavoro del server.

Infine chiediamo a dieci thread virtuali di interrogare il servizio con un limite di tre richieste simultanee. Il punto non è dimostrare che dieci sia un numero magico: è mostrare che il modello di concorrenza del client e la capacità del peer sono limiti diversi. Il lettore ha trasferito il mattone quando sa scegliere fra `send`, `sendAsync` e un thread virtuale per il proprio programma, e sa spiegare quali errori restano possibili in ciascun modello.

> **Riferimenti essenziali.** La specifica Java 25 di `HttpClient` descrive riuso, risorse e chiusura; la specifica dei `BodyHandlers` chiarisce che gli handler predefiniti non esaminano lo status. La politica di retry e la forma del corpo dipendono dal contratto del servizio, non dall'API del client.

28. Memoria esterna e funzioni native: attraversare il confine della JVM

Quando Java deve parlare con una libreria C

Un'applicazione Java può aver bisogno di una libreria di calcolo già scritta in C, oppure di dati mantenuti fuori dall'heap della JVM. Nei capitoli precedenti gli oggetti sono stati creati, letti e raccolti secondo le regole della piattaforma Java. Qui il confine cambia: una funzione nativa segue convenzioni di chiamata della macchina, e un blocco di memoria esterna ha una dimensione e una durata che dobbiamo rispettare esplicitamente. Il problema da risolvere è usare quel confine senza fingere che i controlli ordinari sugli oggetti rendano automaticamente sicuro il codice nativo.

La Foreign Function and Memory API, o FFM, vive nel package stabile `java.lang.foreign` dalla versione 22 ed è disponibile in Java 25. Ha due famiglie collegate: la gestione di segmenti di memoria e il collegamento a funzioni esterne. Storicamente Java ha usato anche JNI, la Java Native Interface. JNI rimane una tecnologia valida per integrazioni esistenti e per casi particolari; FFM offre un modello diverso, con API Java per descrivere memoria, firme e durata. Non basta sostituire una chiamata JNI con una chiamata FFM per rendere portabile una libreria nativa.

> **Concetto chiave – Tre livelli da non confondere.** Un programma Java può essere portabile a livello di bytecode. La FFM API è parte della piattaforma. La funzione C chiamata resta invece legata a una libreria, a un sistema operativo, a un'architettura e a un'ABI, cioè l'insieme delle convenzioni con cui i valori passano fra funzioni sulla macchina. Il contratto C deve essere noto e verificato; il linker non può indovinarlo dal nome della funzione.

La figura 28.1 mostra tre oggetti distinti: l'arena governa la durata, il segmento delimita una regione e il layout descrive come interpretare i byte. Se il segmento rappresenta tre interi, la sua dimensione in byte dipende dal layout scelto. Un `MemorySegment` può anche riferirsi a memoria dell'heap; «segmento» non significa sempre «fuori heap». Nell'esempio usiamo una regione nativa per rendere osservabile il ciclo di vita.



Allocare e leggere con un limite esplicito

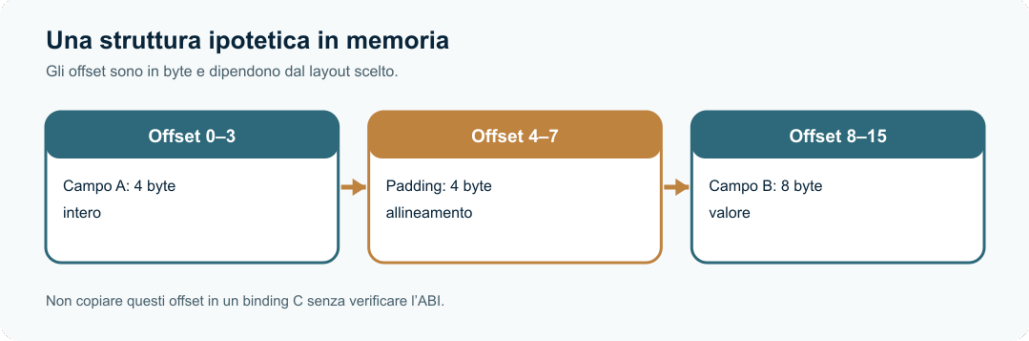
`Arena.ofConfined()` crea un'arena accessibile dal thread proprietario e chiudibile in modo deterministico. Il `try-with-resources` chiude l'arena anche se un'operazione interna lancia un'eccezione. I segmenti allocati da essa condividono la durata dell'arena: dopo la chiusura non si possono più usare. `Arena.ofShared()` consente accesso da più thread e richiede più disciplina sul coordinamento; `Arena.ofAuto()` delega il recupero alla gestione automatica; `Arena.global()` dura per tutta la vita del processo. Scegliamo la forma meno estesa che copre il bisogno reale. Per un buffer usato da un solo metodo, l'arena confinata rende leggibile il confine temporale.

«Condivisa» descrive chi può accedere all'arena, non una garanzia che più modifiche allo stesso segmento formino una decisione atomica. Nel capitolo 20A abbiamo separato accessibilità dello stato e protocollo di aggiornamento: la stessa domanda torna qui, con in più durata del segmento, layout e possibili chiamate native. Se due thread o la libreria C scrivono nello stesso spazio, bisogna definire un coordinamento compatibile con entrambi i lati del confine; un `AtomicInteger` che vive altrove nella JVM non protegge quei byte per il solo fatto di esistere.

`ValueLayout.JAVA_INT` descrive il layout di un `int` Java per l'accesso al segmento. `arena.allocate(ValueLayout.JAVA_INT, 3)` alloca lo spazio per tre elementi con quel layout. `setAtIndex` e `getAtIndex` controllano l'indice rispetto al segmento; un accesso oltre i limiti fallisce invece di leggere memoria adiacente per caso. Il ciclo dell'esempio scrive `1`, `2`, `3` e poi somma i valori ottenendo `6`. Si potrebbe usare un normale `int[]` per questa somma: l'esempio usa memoria nativa soltanto per mostrare i confini dell'API. Quando un programma non deve condividere dati con codice nativo né controllare un layout esterno, un array Java resta una scelta più semplice.

Un segmento può essere tagliato con `asSlice` per lavorare su una parte. Il taglio non rende indipendente la durata della memoria sottostante: se l'arena che la governa viene chiusa, anche la vista è invalida. Un layout di struttura può comporre valori, sequenze e padding; il padding è lo spazio aggiunto per rispettare vincoli di allineamento. Un offset è la distanza in byte dall'inizio della regione. Un errore di offset o di layout può far leggere un campo come se fosse un altro, anche quando l'accesso resta entro i limiti spaziali. La figura 28.2 mostra un layout didattico con byte di riempimento. Prima di usarlo per una struttura C reale occorre verificarne ABI, dimensioni e allineamento sul compilatore e sulla piattaforma del programma.

La differenza fra dimensione logica e disposizione fisica è decisiva. Una struttura con un campo a quattro byte seguito da uno a otto può avere spazio di riempimento fra i campi per allineare il secondo. La figura usa quattro byte di padding come *ipotesi didattica*, non come legge universale per qualunque compilatore. `MemoryLayout.structLayout` permette di descrivere una sequenza di membri; `paddingLayout` rende esplicito lo spazio richiesto dal layout scelto. Per un array di strutture occorre conoscere anche la dimensione complessiva di ciascun elemento, non soltanto gli offset interni. Una descrizione che produce i valori giusti su un singolo computer può essere falsa su un altro ABI.



Un `VarHandle` ottenuto da un percorso nel layout può rendere riusabile l'accesso a un campo annidato. Il nome somiglia ai metodi handle del capitolo 24, ma lo scopo qui è leggere o scrivere una posizione descritta da un layout di memoria. Reflection osserva metadati di tipi e membri Java; FFM descrive dati e chiamate oltre il confine della JVM. La disponibilità di un handle non sostituisce la scelta corretta di durata, allineamento e tipo.

Dalla firma C alla chiamata

Per chiamare una funzione nativa dobbiamo sapere in quale libreria si trova, quale simbolo la identifica e quale firma binaria usa. `SymbolLookup` cerca il simbolo e produce un indirizzo rappresentato come `MemorySegment`. `Linker.nativeLinker()` conosce l'ABI della piattaforma corrente. `FunctionDescriptor` descrive tipi di ritorno e parametri al livello nativo. Infine `downcallHandle` produce un `MethodHandle` che il codice Java può invocare. La figura 28.3 collega questi passaggi; saltarne uno significa affidarsi a una supposizione non verificata.



Nel programma completo usiamo `strlen`*, una funzione della libreria C che conta i byte prima del terminatore zero di una stringa C.* `arena.allocateFrom("Java")` *crea una stringa codificata e terminata secondo il contratto dell'API; poiché le lettere ASCII dell'esempio occupano un byte ciascuna in UTF-8, il risultato è* 4*. Se scrivessimo caratteri accentati,* `strlen` *conterebbe byte, non caratteri Unicode né code point: è proprio il limite da capire. La funzione viene cercata nel lookup predefinito del linker nativo. Questo esempio è stato eseguito sul macOS e sul JDK 25 di lavoro; altre piattaforme possono esporre simboli e tipi C in modo diverso.*

Per costruire un binding riproducibile non basta conoscere `strlen` a memoria. La firma C ufficiale restituisce `size_t` e riceve un puntatore a `char` terminato da zero. Il descrittore FFM traduce il puntatore in `ValueLayout.ADDRESS`, mentre la stringa allocata nell'arena fornisce memoria valida fino alla fine del `try`. Il risultato viene invocato con `invokeExact`, che richiede tipi Java esatti nella chiamata; il cast a `long` rende esplicita la firma del method handle costruito in questo ambiente. Se la funzione conservasse il puntatore dopo il ritorno, l'arena confinata chiusa a fine blocco sarebbe troppo breve: avremmo risolto la chiamata immediata ma non il contratto di durata.

```
import java.lang.foreign.Arena;
import java.lang.foreign.FunctionDescriptor;
import java.lang.foreign.Linker;
import java.lang.foreign.MemorySegment;
import java.lang.foreign.ValueLayout;
import java.lang.invoke.MethodHandle;

public class MemoriaNativa {
    public static void main(String[] args) throws Throwable {
        try (Arena arena = Arena.ofConfined()) {
            MemorySegment numeri = arena.allocate(ValueLayout.JAVA_INT, 3);
            for (int indice = 0; indice < 3; indice++) {
                numeri.setAtIndex(ValueLayout.JAVA_INT, indice, indice + 1);
            }
            int somma = 0;
            for (int indice = 0; indice < 3; indice++) {
```

```
                somma += numeri.getAtIndex(ValueLayout.JAVA_INT, indice);
            }
            System.out.println("Somma nativa: " + somma);

            Linker linker = Linker.nativeLinker();
            MemorySegment simbolo = linker.defaultLookup().find("strlen")
                    .orElseThrow(() -> new IllegalStateException("strlen non disponibile"));
            MethodHandle strlen = linker.downcallHandle(simbolo,
                    FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS));
            MemorySegment testo = arena.allocateFrom("Java");
            long lunghezza = (long) strlen.invokeExact(testo);
            System.out.println("Byte prima dello zero: " + lunghezza);
        }
    }
}
```

Il comando di compilazione è `javac --release 25 MemoriaNativa.java`. Per eseguire una *restricted method* in un programma sul class path usiamo `java --enable-native-access=ALL-UNNAMED MemoriaNativa`. L'opzione abilita l'accesso nativo per tutti i moduli senza nome di quel processo; in un'applicazione modulare si dovrebbe indicare il modulo interessato. La compilazione con `-Xlint:all` segnala il metodo ristretto: l'avviso è informazione pertinente, non un difetto da nascondere. Abilitare l'accesso non verifica automaticamente che il descrittore di `strlen` sia corretto.

> **Attenzione – La firma dipende dall'ABI.** In questa prova `size_t`, il tipo C restituito da `strlen`, viene descritto come `JAVA_LONG`, coerentemente con l'ABI a 64 bit usata nell'ambiente di verifica. Non è una dichiarazione universale: su ABI a 32 bit o con rappresentazioni differenti il binding va adattato. Una firma sbagliata al confine nativo può corrompere memoria o terminare il processo. Il capitolo mostra il metodo di ragionamento e un caso controllato, non una libreria di binding pronta per ogni piattaforma.

Una *upcall* percorre la direzione opposta: una funzione nativa richiama codice Java attraverso uno stub creato dal linker. Lo stub deve restare valido finché la libreria nativa può usarlo; chiudere l'arena troppo presto rende il callback invalido. Anche l'eccezione che nasce nel codice Java richiamato deve essere gestita secondo il contratto del confine, non lasciata attraversare come se la funzione C conoscesse le eccezioni Java. Questa è una ragione per introdurre le upcall solo dopo aver compreso segmenti e downcall.

Immaginiamo una libreria che registra una funzione di notifica e la richiama dopo il ritorno della chiamata di registrazione. Se creiamo lo stub dentro un `try (Arena arena = Arena.ofConfined())` che termina subito dopo la registrazione, consegniamo alla libreria un indirizzo destinato a diventare invalido prima del callback. La correzione non è semplicemente usare `Arena.global()`: bisogna stabilire chi annulla la registrazione, attendere che non ci siano callback in corso e poi chiudere l'arena appropriata. Se la notifica arriva da un thread nativo diverso, serve inoltre verificare i vincoli di accesso e di concorrenza. Il problema determina la durata, non il contrario.

Che cosa controlla l'API e che cosa resta nostro

L'API può controllare limiti del segmento, durata della sua arena e, per un'arena confinata, il thread autorizzato. Non può garantire che una libreria C rispetti la dimensione di un buffer ricevuto, conservi un puntatore soltanto per il tempo consentito o interpreti il layout che noi intendevamo. La sicurezza *spaziale* riguarda gli indirizzi e i limiti; quella *temporale* riguarda il momento in cui la memoria resta valida. Un controllo sul lato Java riduce errori importanti, ma il codice nativo può ancora violare il contratto.

Questo spiega anche perché la verifica non può limitarsi a un output positivo. Occorrono prove dei confini: segmenti troppo piccoli, arena chiusa, simbolo mancante, firma non disponibile sulla piattaforma e funzione nativa che restituisce un codice di errore. Le prime tre possono essere riprodotte in un test Java controllato; una firma ABI errata non va «provata per vedere se funziona» nel processo principale, perché il fallimento può essere un crash o una corruzione non immediatamente visibile. In quel caso si parte da header, documentazione della libreria e binding verificato, poi si usano prove isolate sulla piattaforma dichiarata.

Uno strumento come `jextract` può generare binding da header C, evitando molte trascrizioni manuali. È uno strumento del percorso di integrazione, non una promessa che ogni funzione diventi automaticamente portabile o sicura. Il build deve fissare versione dello strumento, header, compilatore, architettura e librerie usate. Se cambia l'ABI, i binding e le prove vanno ricontrollati. L'API Vector del capitolo 23 risolve un altro problema: elaborare dati con operazioni SIMD. Si possono usare vettori su dati che arrivano da segmenti, ma questo non fonde stabilità, rischi e finalità delle due API.

Verificare e trasferire

La prima verifica modifica il numero di elementi allocati da tre a due lasciando tre scritture: il programma deve fallire al controllo dei limiti, non stampare una somma inventata. La seconda conserva un segmento in una variabile, chiude l'arena e prova a leggerlo: deve fallire perché la durata è terminata. Non inseriamo la seconda prova nel flusso normale, dove interromperebbe il resto dell'esempio. Questi due esperimenti distinguono errore spaziale ed errore temporale.

Il trasferimento cambia la funzione C. Una libreria richiede un puntatore a una struttura con due interi, conserva il puntatore fino a un callback e poi restituisce un codice di stato. Prima di scrivere `downcallHandle`, il lettore deve rispondere: quale layout corrisponde alla struttura su questa ABI? Quale arena copre l'intera durata del puntatore? Quale thread può accedervi? Come si interpreta il codice di stato? Quale test controlla la chiusura? Se una risposta manca, un programma che «compila» non ha ancora risolto il problema.

> **Riferimenti essenziali.** Il package `java.lang.foreign` di Java 25 descrive segmenti, layout e funzioni esterne; la specifica di `Arena` definisce durata e accesso; la specifica di `Linker` chiarisce il ruolo dell'ABI e dei metodi ristretti. Il caso `strlen` resta vincolato all'ambiente nativo dichiarato.

29. Dal sorgente al programma che un altro può eseguire

Il progetto che funziona solo sul nostro computer

Fin qui abbiamo compilato molti esempi con una riga di `javac`. È un buon modo per vedere dove finisce il sorgente e dove comincia il programma eseguibile. Quando i file diventano dieci, arrivano risorse e test, e un collega deve ricostruire lo stesso risultato, quella riga non racconta più tutto. Dove mettiamo i test? Quale versione di Java usiamo? Come otteniamo JUnit? Che cosa contiene il JAR che consegniamo? Se la risposta è «sul mio computer funziona», il progetto non è ancora trasmissibile.

Costruiamo un piccolo catalogo con un titolo per riga. L'applicazione legge un file UTF-8, normalizza gli spazi e stampa i titoli. Non è un sistema bibliotecario completo: il comportamento è deliberatamente piccolo, così possiamo seguire ogni passaggio dal repository all'artefatto. Nel capitolo 25 abbiamo già affrontato file, charset e risorse; qui la domanda è diversa. Vogliamo che una seconda persona possa partire dai file versionati, lanciare i controlli e ottenere un programma distribuibile senza indovinare istruzioni nascoste.

> **Definizione – Build riproducibile.** In questo capitolo «riproducibile» significa che sorgenti, configurazione, versioni dichiarate, ambiente e comandi permettono di ricostruire e verificare il comportamento atteso. Non promettiamo automaticamente byte identici in ogni JAR: timestamp degli archivi, plugin, sistema operativo e altri input possono influire sui byte finali. Se l'identità binaria è un requisito, va progettata e confrontata separatamente.

Prima separiamo ciò che scriviamo da ciò che generiamo

La figura 29.1 mostra il percorso del nostro esempio. I sorgenti di produzione sono sotto `src/main/java`; i test sotto `src/test/java`; il file dimostrativo sta in `dati/`. Maven scrive classi, rapporti e JAR in `target/`. Quest'ultima cartella è un risultato, non una fonte da correggere a mano. Se modifichiamo un `.class` o un rapporto dentro `target`, la prossima build lo sostituirà e nessun collega ritroverà la modifica nel sorgente.

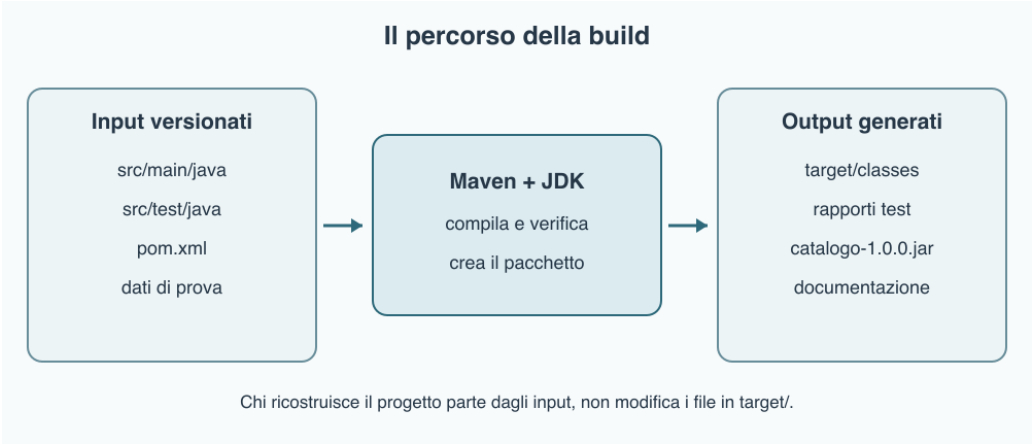


Figura 29.1 – Sorgenti, test e configurazione sono input versionati; classi, rapporti e JAR sono output della build.

Nel progetto completo `catalogo` troviamo tre classi di produzione, due classi di test, il file dimostrativo e `pom.xml`. Un progetto reale può avere risorse in `src/main/resources` e dati di prova in `src/test/resources`; non creiamo quelle directory vuote solo per rispettare un disegno. La distinzione importante è la responsabilità: un test può usare JUnit senza trascinare JUnit nel programma distribuito, e un file generato non viene scambiato per il sorgente della verità.

```
catalogo/
  pom.xml
  dati/titoli.txt
  src/main/java/esempio/catalogo/
    Titoli.java  ArchivioTitoli.java  CatalogoCli.java
  src/test/java/esempio/catalogo/
    TitoliTest.java  ArchivioTitoliTest.java
  target/                 ← creato dalla build
```

Concetto chiave – Tre livelli. Il JDK contiene `javac`, `java`, `jar`, `javadoc`, `jdeps`, `jlink` e altri strumenti. Maven è un programma esterno che organizza fasi e dipendenze, chiamando anche strumenti del JDK tramite plugin. JUnit è una dipendenza esterna che offre il modello dei test e il motore per eseguirli. Una pipeline di integrazione continua ripete i comandi in un ambiente dichiarato. La figura 29.2 distingue questi livelli: un errore di compilazione, un test fallito e un artefatto mancante chiedono diagnosi diverse.

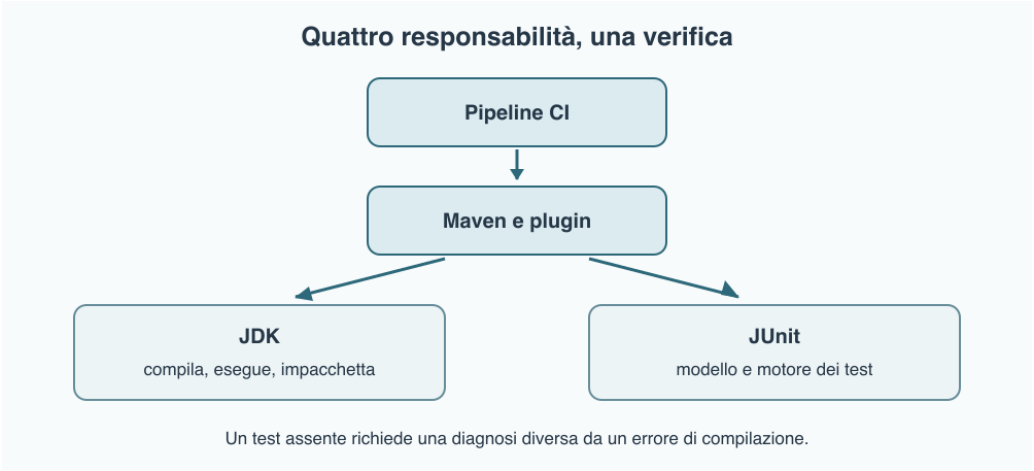


Figura 29.2 – La pipeline avvia la build; Maven orchestra plugin e dipendenze; il JDK compila ed esegue; JUnit definisce e lancia i test attraverso la piattaforma.

Prima ancora di Maven: compilare e capire i percorsi

Il compilatore `javac` legge i `.java` e produce file `.class` per la JVM. Se siamo nella cartella `catalogo`, possiamo isolare i file generati con questo comando:

```
mkdir -p target/manuale
javac --release 25 -Xlint:all -d target/manuale \
  src/main/java/esempio/catalogo/*.java
java -cp target/manuale esempio.catalogo.CatalogoCli dati/titoli.txt
```

`-d target/manuale` indica dove mettere le classi. Il loro percorso interno segue il package `esempio.catalogo`: il compilatore crea `esempio/catalogo` sotto la directory di output. `-cp` o `--class-path` dice al launcher dove cercare le classi dell'applicazione. `--release 25` seleziona il livello di linguaggio, il formato delle classi e le API della release dichiarata per la compilazione; non installa Java 25 su un altro computer e non rende disponibili da sola librerie di terze parti. `-Xlint:all` rende visibili categorie di avvisi utili, da leggere e correggere o motivare. Una build che sopprime tutti gli avvisi per ottenere una schermata pulita perde informazione.

Nel capitolo 6 abbiamo visto il *module path*. Il nostro esempio resta sul class path per rendere il percorso di build iniziale trasparente; il JAR ottenuto non è un modulo nominato. Se un progetto dichiara `module-info.java`, compilazione, risoluzione dei moduli e comando di avvio cambiano: occorre progettare `requires` ed `exports`, non aggiungere `--module-path` a caso. `jlink`, che useremo più avanti, può creare un runtime anche per questa applicazione sul class path scegliendo i moduli del JDK necessari; ciò non trasforma il JAR in un modular JAR.

L'esecuzione prevista stampa `Titoli letti: 2`, poi `Java moderno` ed `È tempo di imparare`. Il secondo titolo verifica che la lettura sia davvero UTF-8 nel caso di prova, ma un solo file positivo non prova che qualsiasi input sia accettabile. Nel capitolo 30 torneremo proprio sul confine dei dati non fidati.

Le tre classi del caso di studio

La prima classe, `Titoli`, rappresenta una regola di dominio piccola ma verificabile. Riceve una stringa, elimina gli spazi ai bordi, riduce sequenze interne di spazi e rifiuta un titolo vuoto o oltre ottanta unità `char` Java. Il limite è una regola didattica, non una norma universale per i titoli dei libri. `String.length()` conta unità UTF-16, non necessariamente caratteri percepiti da una persona: se il dominio richiede un altro criterio, il test e la regola devono cambiarlo esplicitamente.

```
package esempio.catalogo;

import java.util.Objects;

/** Regole didattiche per i titoli del piccolo catalogo. */
public final class Titoli {
    private Titoli() {
    }

    /**
     * Elimina spazi ai bordi e riduce sequenze di spazi interni.
     *
     * @param ingresso titolo da normalizzare
     * @return titolo non vuoto e lungo al massimo 80 caratteri Java
     * @throws NullPointerException se ingresso è null
     * @throws IllegalArgumentException se il titolo è vuoto o troppo lungo


     */
    public static String normalizza(String ingresso) {
        Objects.requireNonNull(ingresso, "ingresso");
        String titolo = ingresso.strip().replaceAll("\\s+", " ");
        if (titolo.isEmpty() || titolo.length() > 80) {
            throw new IllegalArgumentException("Titolo vuoto o troppo lungo");
        }
        return titolo;
    }
}
```

`Objects.requireNonNull` rende distinto il caso `null` da una stringa vuota. `strip()` rimuove spazi ai bordi secondo le regole Unicode del metodo; `replaceAll("\\s+", " ")` segue invece la definizione di `\s` dell'espressione regolare Java usata qui. Non promettiamo con questa riga di normalizzare qualunque forma di spazio Unicode in modo identico: se il catalogo riceve dati internazionali complessi, specifichiamo i caratteri da accettare e aggiungiamo esempi. La Javadoc del metodo documenta il contratto pubblico e le eccezioni, così una modifica della regola può essere riconosciuta nei test e nella documentazione generata.

La seconda classe, `ArchivioTitoli`, collega quella regola al file. `Files.lines` restituisce uno stream che usa una risorsa da chiudere: il `try-with-resources` la chiude anche se una riga provoca un'eccezione. Le righe vuote vengono saltate, le altre passano per `Titoli.normalizza`. Il risultato è una copia non modificabile della lista raccolta; non significa che il file originale non possa cambiare dopo la lettura.

```
package esempio.catalogo;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

/** Legge un archivio didattico con un titolo per riga. */
public final class ArchivioTitoli {
    private ArchivioTitoli() {
    }

    /**
     * Restituisce i titoli non vuoti nell'ordine del file.

     *
     * @param percorso file UTF-8 con un titolo per riga
     * @return lista non modificabile dei titoli normalizzati
     * @throws IOException se la lettura fallisce
     */
    public static List<String> leggi(Path percorso) throws IOException {
        List<String> titoli = new ArrayList<>();
        try (var righe = Files.lines(percorso, StandardCharsets.UTF_8)) {
            for (String riga : (Iterable<String>) righe::iterator) {
                if (!riga.isBlank()) {
                    titoli.add(Titoli.normalizza(riga));
                }
            }
        }
        return List.copyOf(titoli);
    }

}
```

Il ciclo usa un adattamento `Iterable` per percorrere lo stream nel blocco che lo chiude. È una scelta locale dell'esempio: in un progetto si potrebbe usare una pipeline Stream, purché la gestione della risorsa resti visibile. Questa versione accumula tutti i titoli in memoria e non impone un limite al numero di righe né alla dimensione della riga letta. Va bene per un file dimostrativo controllato; non è ancora un importatore sicuro per file arbitrari. Il capitolo 30 mostrerà come cambiare il requisito senza fingere che una build verde equivalga a una verifica di sicurezza.

La terza classe, `CatalogoCli`, è il punto d'ingresso. Riceve il percorso del file come argomento, chiama l'importazione e stampa il risultato. La scelta di richiedere esattamente un argomento rende ripetibile la prova. Un'applicazione per utenti finali dovrebbe tradurre un errore di input o di lettura in un messaggio e in un codice di uscita appropriato; qui lasciamo visibile la distinzione fra fallimento della build e fallimento a tempo di esecuzione.

```
package esempio.catalogo;

import java.io.IOException;
import java.nio.file.Path;

/** Punto di ingresso dell'esempio di build e distribuzione. */
public final class CatalogoCli {
    private CatalogoCli() {
    }

    /**
     * Legge e stampa i titoli del file indicato.
     *
     * @param args un percorso di file
     * @throws IOException se la lettura fallisce
     */

    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            throw new IllegalArgumentException("Uso: CatalogoCli <file-titoli>");
        }
        var titoli = ArchivioTitoli.leggi(Path.of(args[0]));
        System.out.println("Titoli letti: " + titoli.size());
        for (String titolo : titoli) {
            System.out.println(titolo);
        }
    }
}
```

Perché introdurre Maven

Abbiamo appena compilato manualmente tre file. Potremmo fare lo stesso con i test, ma dovremmo trovare JUnit, includere ogni suo JAR e avviare il motore corretto. Poi dovremmo ripetere comandi coerenti per documentazione, pacchetto e analisi. Un sistema di build registra questa sequenza. Maven usa un file di progetto chiamato *POM*, `pom.xml`, che descrive coordinate, dipendenze, plugin e configurazione. Le coordinate `groupId:artifactId:version` identificano il nostro artefatto; nel caso di studio sono `esempio:catalogo:1.0.0`.

Nel POM completo fissiamo la release del compilatore a 25, JUnit 5.14.1 per i soli test, e le versioni dei plugin che ci servono. Dichiarare le versioni è una parte della ripetibilità: affidarsi alle versioni implicite del computer o del parent POM può cambiare il comportamento quando quell'ambiente cambia. Il file non nasconde che Maven e JUnit sono strumenti esterni a Java SE.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                     https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>esempio</groupId>
    <artifactId>catalogo</artifactId>
    <version>1.0.0</version>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.release>25</maven.compiler.release>
        <junit.version>5.14.1</junit.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>

            <artifactId>junit-jupiter</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.14.1</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.5.4</version>

            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.4.2</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>esempio.catalogo.CatalogoCli</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

`<scope>test</scope>` impedisce che JUnit diventi una dipendenza ordinaria dell'applicazione in esecuzione. Il plugin compiler usa `--release 25`; Surefire scopre ed esegue i test Jupiter; il plugin JAR scrive nel manifest la classe principale. Se un test non compare nel rapporto, il problema non si risolve contando soltanto le righe `BUILD SUCCESS`: occorre controllare convenzioni dei nomi, motore di test e rapporto dei test eseguiti. Nel nostro caso la build riporta

cinque invocazioni di test. In una macchina priva delle dipendenze nella cache Maven, la prima build deve poter raggiungere un repository autorizzato o un mirror aziendale; l'opzione `-o` usata nella verifica locale funziona soltanto perché gli artefatti sono già presenti.

`mvn clean verify` chiede prima di eliminare l'output precedente e poi percorre le fasi necessarie fino a `verify`. Maven esegue i goal associati alle fasi secondo il POM e il tipo di packaging. La parola `verify` non aggiunge per magia test di integrazione o controlli di sicurezza: li esegue solo se sono stati configurati nel ciclo. `mvn test` ferma il percorso ai test; `mvn package` costruisce il JAR; `mvn install` copia l'artefatto nel repository locale del computer. `deploy` lo invia a un repository remoto configurato: non è un sinonimo di «eseguire l'applicazione». Per una prova pulita del progetto, dalla sua cartella usiamo:

```
mvn clean verify
java -jar target/catalogo-1.0.0.jar dati/titoli.txt
```

> **Approfondimento – Maven e Gradle risolvono lo stesso bisogno organizzativo.** Gradle descrive attività e dipendenze con un modello diverso, mentre Maven ha convenzioni e fasi dichiarate nel POM. In questo volume usiamo Maven come percorso riproducibile unico, così codice, test e output restano verificabili con una sola configurazione. Chi usa Gradle deve tradurre i requisiti, non copiare letteralmente le fasi Maven: release Java 25, separazione dei test, versioni fissate, JAR avviabile e prova su ambiente pulito rimangono gli stessi obiettivi. Il wrapper dello strumento e la verifica delle distribuzioni sono parti del progetto quando si adotta Gradle.

Una dipendenza arriva spesso con altre dipendenze

JUnit compare nel POM come dipendenza diretta. Le sue componenti possono portare altre librerie *transitive*: Maven legge i POM delle dipendenze e costruisce un grafo. Se due rami chiedono versioni diverse della stessa libreria, la mediazione della versione può scegliere un risultato che il lettore non si aspettava. Perciò «ho dichiarato solo una libreria» non significa «il progetto usa un solo artefatto esterno». `mvn dependency:tree` è il primo modo di osservare il grafo quando il plugin necessario è disponibile; si registrano anche repository, versioni risolte e politica di aggiornamento. Un `scope` sbagliato può lasciare fuori dal runtime una libreria che il programma usa, oppure portare nel pacchetto una libreria usata solo nei test.

La risoluzione non è una garanzia di fiducia. Un repository può essere indisponibile; un artefatto può essere sostituito o compromesso; una versione fissata può contenere una vulnerabilità nota. Per una build professionale, fissiamo origine e integrità secondo gli strumenti disponibili, conserviamo un inventario delle dipendenze e rivalutiamo gli aggiornamenti. Il capitolo 30 riprenderà la catena di fornitura come problema di sicurezza e manutenzione. Qui ci basta imparare a leggere il POM come parte del codice del progetto, non come un file amministrativo da lasciare a chi «si occupa della build».

Testare il comportamento, non soltanto eseguire metodi

JUnit non è incluso nel JDK. Lo usiamo perché descrive casi e risultati attesi in un formato eseguibile e leggibile. Il test di `Titoli` copre due input positivi con un test parametrizzato e due limiti negativi. `@CsvSource` fornisce coppie ingresso–atteso; `@Test` marca gli altri casi; `assertThrows` verifica che un titolo vuoto o lungo 81 unità venga rifiutato. La ripetizione dei casi positivi serve a mostrare che la stessa regola si trasferisce a stringhe diverse, non a gonfiare il numero di test.

```
package esempio.catalogo;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class TitoliTest {
    @ParameterizedTest
    @CsvSource({"'  Java   moderno  ', 'Java moderno'",
                "'Il mattone', 'Il mattone'"})
    void normalizzaSpazi(String ingresso, String atteso) {
        assertEquals(atteso, Titoli.normalizza(ingresso));
    }

    @Test
    void rifiutaTitoloVuoto() {
        assertThrows(IllegalArgumentException.class,
                () -> Titoli.normalizza("   "));
    }

    @Test
    void rifiutaTitoloTroppoLungo() {
        assertThrows(IllegalArgumentException.class,
                () -> Titoli.normalizza("a".repeat(81)));
    }
}
```

Il test di `ArchivioTitoli` usa `@TempDir`. JUnit crea una directory di prova, e il test scrive lì un file UTF-8: non dipende da un percorso personale, da un file lasciato da un'esecuzione precedente o dall'ordine dei test. Qui vengono esercitati insieme file system, decodifica, filtro delle righe e normalizzazione. È più vicino a un test d'integrazione fra componenti locali rispetto ai test puri di `Titoli`, pur restando piccolo e veloce. La parola «integrazione» non impone automaticamente un database o un server remoto.

```
package esempio.catalogo;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.io.TempDir;

class ArchivioTitoliTest {
    @TempDir Path cartella;

    @Test

    void leggeRigheNonVuoteInUtf8() throws IOException {
        Path file = cartella.resolve("titoli.txt");
        Files.writeString(file, "  Java  moderno  \n\nÈ tempo di imparare\n",
                StandardCharsets.UTF_8);
        assertEquals(List.of("Java moderno", "È tempo di imparare"),
                ArchivioTitoli.leggi(file));
    }
}
```

Il test unitario risponde a una domanda stretta: a questo input, la funzione produce il risultato o l'errore pattuito? Il test d'integrazione controlla che più parti collaborino. La prova di sistema avvia il JAR come lo farà un utente e verifica manifest, class path e file reale. Nel nostro caso `java -jar ... dati/titoli.txt` è proprio quella prova di fumo: se fallisce, i test unitari verdi non salvano l'artefatto. Una pipeline robusta esegue controlli nei livelli che corrispondono ai rischi, senza trasformare la «piramide dei test» in una legge sul numero esatto di prove per categoria.

Per test deterministici, il tempo va introdotto tramite `Clock`, come nel capitolo 26, invece di leggere `Instant.now()` nel cuore della regola. I file temporanei devono essere creati e ripuliti dal test. Le prove di rete del capitolo 27 usano server locali controllati, non un sito pubblico che può cambiare o essere irraggiungibile. La concorrenza dei capitoli 20 e 20A richiede verifiche di invarianti, non confronti con l'ordine di stampa di thread diversi. Questo non elimina i difetti intermittenti, ma evita di aggiungere instabilità artificiale ai test.

> **Concetto chiave – Copertura non è correttezza.** Una percentuale di righe eseguite dice quali parti del codice sono state attraversate dal test. Non dice se l'asserzione riconoscerebbe un risultato sbagliato, se un limite è stato provato o se il requisito era corretto. Se un test chiama `normalizza("Java")` senza confrontare il risultato, può aumentare la copertura senza controllare quasi nulla. Un modo utile di giudicare la prova è chiedere quale modifica difettosa farebbe fallire quel test. La mutation testing può automatizzare parte di questo esperimento, ma anche i mutanti uccisi non sostituiscono la scelta dei requisiti da verificare.

Dal codice compilato al JAR

Un JAR è un archivio che contiene classi e risorse con una struttura definita. `jar --list --file target/catalogo-1.0.0.jar` mostra le tre classi del programma, il manifest e metadati Maven. I test non sono nel JAR ordinario, perché si trovano nell'output di test. Nel manifest, `Main-Class` indica `esempio.catalogo.CatalogoCli`: è ciò che permette `java -jar`. Senza quella voce, un JAR può essere valido come libreria e tuttavia non avviarsi con quel comando. La figura 29.3 distingue tre risultati spesso confusi: JAR, runtime image e pacchetto applicativo.

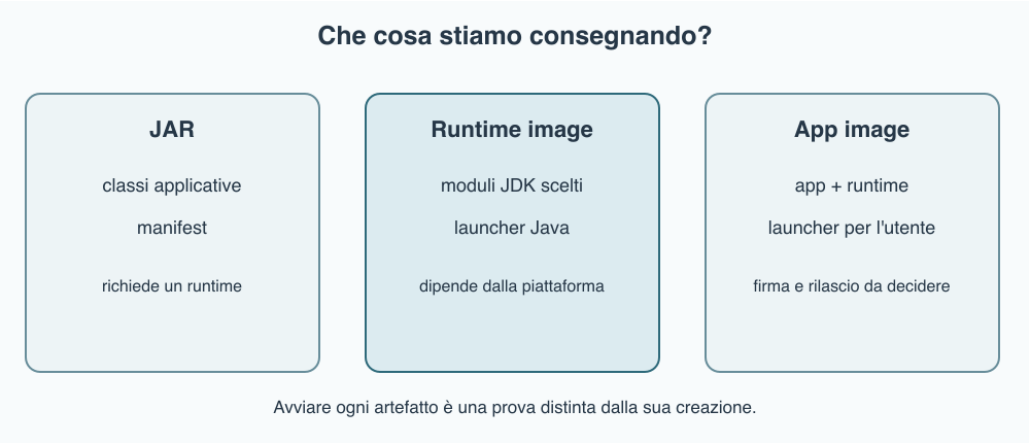


Figura 29.3 – Il JAR porta l'applicazione, il runtime image porta i moduli JDK scelti, il pacchetto riunisce ciò che serve all'utente della piattaforma di destinazione.

Il JAR dell'esempio non contiene JUnit e non è un *fat JAR* che ingloba tutte le dipendenze esterne. Può avviarsi da solo perché il programma di produzione usa soltanto API del JDK. Se aggiungessimo una libreria di produzione, dovremmo scegliere come distribuirla: class path accanto al JAR, layout applicativo, modulo o pacchetto che la includa. Il manifest non scarica le dipendenze al momento dell'avvio. Guardare il contenuto dell'artefatto è quindi una verifica, non una curiosità.

Documentazione e strumenti di analisi del JDK

La Javadoc di `Titoli.normalizza` dichiara ingresso, valore restituito ed errori. `javadoc` può generare pagine HTML a partire dai commenti `/** ... */`; un commento che ripete il nome del metodo senza spiegare il contratto non aiuta. `javadoc -d target/docs src/main/java/esempio/catalogo/*.java` produce la documentazione in una cartella generata. La verifica della documentazione deve controllare avvisi e link, oltre a leggere la pagina prodotta: una pagina HTML esistente può comunque descrivere una regola obsoleta. `javap -c -p -classpath target/classes esempio.catalogo.Titoli` mostra il bytecode e i membri della classe compilata; non è un sostituto della lettura del sorgente, ma aiuta a capire che cosa è finito nell'artefatto.

`jdeps --print-module-deps target/catalogo-1.0.0.jar` riporta `java.base` per questo caso. È un'analisi delle dipendenze staticamente osservabili verso moduli

della piattaforma. Non può dedurre sempre un plugin caricato per nome, una classe cercata da configurazione o una libreria nativa aperta più tardi. `jdeprscan --release 25 target/catalogo-1.0.0.jar` cerca usi di API deprecate nel bytecode secondo la release scelta; l'assenza di righe nel nostro caso non prova assenza di ogni problema di migrazione. Quando un'analisi produce un risultato, lo colleghiamo alla riga di codice o alla dipendenza che lo causa invece di usare il numero di avvisi come unico indicatore.

`jshell` è utile per provare una chiamata o un'ipotesi con feedback immediato. Un debugger permette di fermarsi su una riga e osservare valori e stack; `jdb` è l'interfaccia a riga di comando del JDK, mentre gli IDE offrono viste più comode. Nessuno dei due sostituisce un test da conservare: l'esperimento interattivo diventa conoscenza del progetto quando trasformiamo l'osservazione in un caso ripetibile. Nel capitolo 30 useremo `jcmd` e Java Flight Recorder per diagnosticare un processo già in esecuzione; anche lì la domanda precede il comando.

Un runtime per il programma

Una macchina che riceve il solo JAR ha comunque bisogno di un runtime Java compatibile. `jlink` costruisce una runtime image a partire dai moduli JDK scelti. Nel nostro caso `jdeps` indica `java.base`, e il programma non carica altri moduli dinamicamente. Dalla cartella `catalogo` possiamo provare:

```
jlink --add-modules java.base --output target/runtime
target/runtime/bin/java -jar target/catalogo-1.0.0.jar dati/titoli.txt
```

La seconda riga è la prova decisiva: non basta che `jlink` crei una directory. Un'applicazione che usa reflection, provider di servizi, charset particolari o funzionalità caricate dinamicamente può richiedere moduli che `jdeps` non vede nel percorso statico. Il runtime va provato con i casi d'uso reali su un ambiente pulito e sulla piattaforma di destinazione. Una runtime image creata per macOS non è automaticamente un runtime per Windows o Linux: i binari del JDK dipendono dalla piattaforma.

`jpackage` può combinare l'applicazione e un runtime in un'immagine applicativa o in un formato installabile supportato dal sistema operativo. Sul Mac usato per la verifica del libro il seguente comando ha prodotto un'app image e il relativo launcher ha eseguito il file di prova:

```
jpackage --type app-image --name CatalogoDidattico \
  --input target --main-jar catalogo-1.0.0.jar \
  --runtime-image target/runtime --dest target/pacchetto
```

La posizione del launcher e il formato del pacchetto cambiano con il sistema operativo. Per distribuire a terzi, oltre a lanciare il programma occorre decidere versione, firma, provenienza, licenze, aggiornamento e rimozione. Un'app image locale non equivale a un installer firmato e verificato su tutte le piattaforme. Il comando mostra il passo tecnico, non sostituisce una politica di rilascio.

Il controllo che un collega può ripetere

Integrazione continua significa far ripetere a una macchina, a ogni cambiamento proposto, i controlli definiti dal progetto. Per il nostro catalogo una pipeline minima fa checkout del repository, seleziona il JDK 25, esegue `mvn -B clean verify`, conserva rapporti dei test e JAR, poi svolge una prova di avvio dell'artefatto. `-B` chiede una modalità adatta a un ambiente non interattivo; non cambia la semantica dei test. Il job deve sapere da dove arrivano JDK, Maven e dipendenze, e quale ambiente di rete o cache usa. Se la build locale funziona solo grazie a un file installato a mano, la pipeline rivela un input mancante.

Passo	Evidenza osservabile	Domanda se fallisce
Compilazione	classi e avvisi dichiarati	Quale sorgente o API non coincide con Java 25?
Test	numero di casi, errori, rapporti	La regola è sbagliata o il test dipende dall'ambiente?
Pacchetto	JAR e manifest ispezionati	La classe principale e le dipendenze sono presenti?
Avvio	output del JAR e dell'immagine runtime	Il programma parte senza l'ambiente di sviluppo?
Analisi	moduli, deprecazioni, dipendenze	Quale rischio è reale e dove va corretto?

Il controllo non deve interrompersi alla scritta `BUILD SUCCESS`: se Surefire esegue zero test, il requisito «test eseguiti» non è soddisfatto. Analogamente, una build può riuscire pur producendo un JAR non avviabile o un programma che non trova un file nel pacchetto. Per questo il caso di studio ha più prove, ciascuna legata a una promessa precisa. Le attività del capitolo 29 chiedono di introdurre un difetto alla volta e identificare in quale passaggio dovrebbe emergere.

Trasferiamo il metodo

Immaginiamo di sostituire il file locale con un client HTTP del capitolo 27. Il codice di dominio resta in `src/main/java`, ma il test che usa una risposta di rete deve controllare un server locale e i suoi tempi. Il runtime potrebbe richiedere moduli diversi; il pacchetto dovrebbe conservare configurazione e certificati in modo appropriato. Oppure immaginiamo di trasformare il catalogo in un modulo nominato: il `pom.xml`, il JAR e il comando di avvio dovrebbero riflettere `module-info.java`. In entrambi i casi non copiamo alla cieca l'ultimo comando del capitolo: ricostruiamo il grafo di input, prove e output che l'altro programmatore deve poter seguire.

La domanda con cui chiudiamo è semplice: **quali file e quali istruzioni servono a ricostruire il comportamento che abbiamo promesso?** Se non possiamo rispondere guardando repository, POM, test e rapporto di build, manca ancora un mattone. Il capitolo 30 riparte da qui: una build ripetibile ci dà una base affidabile per cercare difetti, proteggere confini e osservare il programma mentre lavora.

Riferimenti essenziali

Riferimento essenziale – Strumenti e librerie. Le specifiche Oracle di `javac`, `jar`, `javadoc`, `jdeps`, `jdeprscan`, `jlink` e `jpackage` definiscono i comandi usati nel JDK 25. Apache Maven documenta il ciclo di build e la risoluzione delle dipendenze. La guida JUnit 5.14.1 documenta i test parametrizzati del progetto. Queste fonti precisano i contratti degli strumenti; i comandi del caso di studio sono stati verificati nell'ambiente dichiarato e devono essere riprovati sulla piattaforma di distribuzione.

30. Qualità, sicurezza e osservabilità: capire quando il programma non basta

La build è verde; il problema resta

Il capitolo 29 ci ha consegnato un catalogo che si compila, supera cinque test e si avvia da un JAR e da un'applicazione impacchettata. È un progresso reale: un collega può ripetere quei passaggi. Possiamo però consegnare quel programma a chiunque carichi un file arbitrario? `ArchivioTitoli` usa uno stream di righe, normalizza i titoli e accumula tutto in una lista. Non impone un massimo alle righe lette, né al numero di titoli. Un file molto grande può consumare memoria; una riga enorme può essere letta prima che `Titoli.normalizza` la rifiuti. I cinque test non chiedevano questo comportamento, quindi il loro successo non risponde alla nuova domanda.

Il nostro caso di studio cambia ora una condizione: il file potrebbe arrivare da un utente e non essere fidato. Dobbiamo stabilire che cosa accettiamo, quanto lavoro siamo disposti a fare, che cosa registriamo quando qualcosa va storto e come raccogliamo prove se il processo rallenta. Queste decisioni intrecciano qualità, sicurezza e operabilità, ma non sono la stessa proprietà. Una regola di input corretta può essere scritta in codice illeggibile; un log dettagliato può esporre un dato riservato; una build pulita può distribuire una dipendenza vulnerabile. Per questo il capitolo continua il percorso del catalogo invece di offrire una lista di comandi da spuntare.

> **Definizione – Qualità osservabile.** Correttezza significa rispettare il comportamento dichiarato. Leggibilità e manutenibilità riguardano la possibilità di comprendere e cambiare quel comportamento. Prestazioni riguardano tempo e risorse su un carico definito. Sicurezza riguarda il trattamento di dati, permessi e confini di fiducia. Operabilità significa poter capire e governare un programma in esecuzione. Nessuna cifra unica dimostra tutte queste proprietà: per ognuna dobbiamo dire quale evidenza cerchiamo.

La figura 30.1 collega cinque passi. Preveniamo un difetto fissando un contratto; lo rileviamo con test e strumenti; lo diagnostichiamo con prove; lo correggiamo in un cambiamento controllato; conserviamo una prova di regressione. La freccia torna all'inizio perché un nuovo requisito, come il file non fidato, modifica ciò che dobbiamo prevenire.



Figura 30.1 – Una correzione affidabile collega il problema osservato a una prova che impedirà di reintrodurlo.

Un requisito nuovo obbliga a rivedere il codice

Nell'importatore del capitolo 29 il metodo `Files.lines` legge una riga per volta, ma questo non costituisce un limite di lunghezza: la libreria deve comunque formare la stringa della riga. Inoltre la lista cresce con il numero dei titoli. Se il file è interno, piccolo e controllato, il compromesso può essere ragionevole. Se invece l'utente può caricarlo, definiamo prima il contratto: nel nostro esperimento accettiamo al massimo cento titoli non vuoti e ottanta unità UTF-16 per riga. Il numero è una scelta dimostrativa; un servizio reale deve sceglierlo in base al dominio, ai costi, ai messaggi d'errore e alla capacità disponibile.

La figura 30.2 distingue i confini del file: percorso, byte, decodifica, righe, titoli normalizzati e oggetti del dominio. Ogni confine può fallire in modo diverso. Un percorso può uscire dalla directory permessa; byte malformati possono non essere testo UTF-8; una riga può essere troppo lunga; un titolo può essere sintatticamente valido ma non rispettare le regole del catalogo. Non basta un unico `catch` che trasformi tutto in «file non valido», perché così perdiamo la causa utile per decidere se correggere l'input, i permessi o il programma.

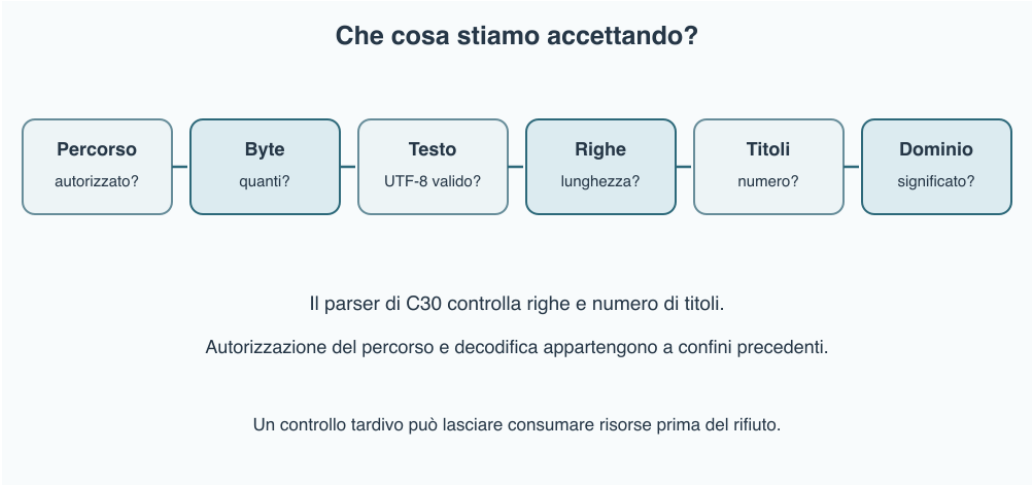


Figura 30.2 – La validazione segue l'ingresso dal nome del file fino alla regola di dominio; ciascun passaggio ha rischi e limiti propri.

Il programma completo `ImportazioneControllata` lavora su un `Reader`, non su un percorso. Il chiamante decide da dove arrivano i caratteri; il metodo si occupa delle righe e dei limiti. Nel `main` usiamo `StringReader` per una prova piccola e indipendente dal file system. In un'applicazione reale si costruirebbe il `Reader` con una decodifica esplicita, per esempio `Files.newBufferedReader(percorso, StandardCharsets.UTF_8)`, dopo aver definito quale percorso è autorizzato. Separare questi due compiti rende più facile provare il parser senza fingere di aver risolto il problema dei percorsi.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.List;

public final class ImportazioneControllata {
    private static final int MAX_RIGA = 80;
    private static final int MAX_TITOLI = 100;
    private static final System.Logger LOG =
            System.getLogger(ImportazioneControllata.class.getName());

    private ImportazioneControllata() {
    }
```

```
    public static List<String> leggi(Reader sorgente) throws IOException {
        List<String> titoli = new ArrayList<>();
        StringBuilder riga = new StringBuilder();
        try (BufferedReader lettore = new BufferedReader(sorgente)) {
            int letto;
            while ((letto = lettore.read()) != -1) {
                if (letto == '\n') {
                    aggiungi(riga, titoli);
                    riga.setLength(0);
                } else if (letto != '\r') {
                    if (riga.length() == MAX_RIGA) {
                        throw new IOException("Riga troppo lunga");
                    }
                    riga.append((char) letto);
                }
            }

            if (!riga.isEmpty()) {
                aggiungi(riga, titoli);
            }
        }
        LOG.log(System.Logger.Level.INFO,
                "Importati {0} titoli", titoli.size());
        return List.copyOf(titoli);
    }

    private static void aggiungi(StringBuilder riga, List<String> titoli)
            throws IOException {
        String titolo = riga.toString().strip().replaceAll("\\s+", " ");
        if (titolo.isEmpty()) {
            return;
        }
        if (titoli.size() == MAX_TITOLI) {

            throw new IOException("Troppi titoli");
        }
        titoli.add(titolo);
    }

    public static void main(String[] args) throws IOException {
        String dati = "  Java   moderno  \nÈ tempo di imparare\n";
        List<String> titoli = leggi(new StringReader(dati));
        System.out.println("Titoli validi: " + titoli.size());
    }
}
```

`BufferedReader` evita di chiedere un carattere alla sorgente fisica per ogni chiamata. Il ciclo legge un'unità `char` per volta: prima di aggiungerla a `StringBuilder` controlla se la riga ha già raggiunto ottanta unità. Così una riga di migliaia di caratteri non viene accumulata per intero nel nostro `StringBuilder` prima del rifiuto. Il lettore può comunque avere un proprio buffer interno di dimensione limitata; il contratto è sullo stato che il parser conserva e sul punto in cui interrompe la lettura, non sul fatto che nessun byte successivo sia mai stato prelevato dal sistema operativo. A ogni `\n` la riga viene normalizzata e inserita soltanto se non è vuota. `\r` viene ignorato per consentire il finale di riga CRLF del

caso didattico. Questo semplice formato non è un parser universale: una regola su righe terminate solo da CR o su altri separatori andrebbe dichiarata e provata.

`aggiungi` controlla il numero di titoli **prima** di inserire il centunesimo. Se supera il limite, il metodo lancia `IOException` e la lista parziale non viene restituita. Il `try-with-resources` chiude il `Reader` passato: è parte del contratto del metodo, da documentare per non sorprendere chi volesse riutilizzare la stessa sorgente. `List.copyOf` evita che il chiamante modifichi la lista restituita; non rende però atomica una scrittura successiva su database. Il capitolo 20A ci ha insegnato a specificare proprio questo confine.

Resta un limite da progettare prima di usare questo codice come ingresso pubblico: un file con milioni di righe vuote non supera mai `MAX_TITOLI`, e il metodo continua a leggere. La memoria delle righe resta contenuta, ma il tempo e il numero totale di caratteri letti no. Un servizio deve aggiungere un massimo complessivo di byte o caratteri, un massimo di righe anche vuote e, se la sorgente è una rete, un timeout al livello che governa la lettura. Quei limiti devono avere errori e test propri. Il nostro esempio prova due confini precisi; chiamarlo «importatore sicuro» senza dichiarare gli altri sarebbe un'affermazione falsa.

I due limiti non sostituiscono il controllo semantico. Potremmo rifiutare titoli duplicati, caratteri di controllo o stringhe che non corrispondono a una politica editoriale; oppure conservarli intenzionalmente. Prima si scrive la regola, poi si implementa e si testa. La validazione non è «ripulire tutto» finché un dato pericoloso sembra innocuo: una trasformazione silenziosa può cambiare il significato di un titolo. Nel programma normalizziamo gli spazi perché il caso di studio lo richiede; non eliminiamo arbitrariamente segni o accenti.

Un test che nasce dal nuovo rischio

Il programma `ProvaImportazioneControllata` verifica tre proprietà. Un input valido con una riga vuota produce due titoli; una riga di 81 unità è rifiutata; 101 righe non vuote sono rifiutate. `rifiutato` non si limita a catturare qualunque eccezione: controlla il messaggio atteso, così un `NullPointerException` accidentale non apparirebbe come il rifiuto progettato. La prova è un piccolo harness del JDK, eseguibile anche senza JUnit; in un progetto Maven sarebbe naturale trasferire gli stessi casi in test JUnit parametrizzati e con asserzioni appropriate.

```
import java.io.IOException;
import java.io.StringReader;
import java.util.List;

public final class ProvaImportazioneControllata {
    private ProvaImportazioneControllata() {
    }

    private static void rifiutato(String dati, String messaggio) {
        try {
            ImportazioneControllata.leggi(new StringReader(dati));
            throw new AssertionError("Input accettato: " + messaggio);
        } catch (IOException previsto) {
            if (!previsto.getMessage().equals(messaggio)) {
                throw new AssertionError("Errore diverso", previsto);
            }
```

```
        }
    }

    public static void main(String[] args) throws IOException {
        List<String> validi = ImportazioneControllata.leggi(
                new StringReader("  Java   moderno  \n\nReti\n"));
        if (!validi.equals(List.of("Java moderno", "Reti"))) {
            throw new AssertionError("Titoli inattesi: " + validi);
        }
        rifiutato("x".repeat(81), "Riga troppo lunga");
        rifiutato("x\n".repeat(101), "Troppi titoli");
        System.out.println("Input valido e due limiti verificati");
    }
}
```

Compilando dalla cartella `esempi/C30` con `javac --release 25 -Xlint:all -d build *.java`, poi eseguendo `java -cp build ProvaImportazioneControllata`, otteniamo `Input valido e due limiti verificati`. Può apparire anche una riga di log informativa su `stderr`; formato, data e lingua dipendono dal provider di logging dell'ambiente, perciò la prova non confronta quei byte. Il test verifica invece il risultato e i due rifiuti, che fanno parte del contratto. Un'esecuzione positiva da sola non avrebbe esposto la crescita senza limite del codice precedente.

> **Concetto chiave – Sintassi, semantica e risorse.** Un input può rispettare la forma e violare una regola del dominio; può rispettare entrambe ma costare troppo da elaborare; può essere valido in sé ma arrivare da un percorso che il chiamante non deve poter leggere. I controlli vanno posti al confine pertinente. Spostare un controllo da una fase all'altra senza capirne il costo può lasciare aperto proprio il problema che volevamo risolvere.

Rendere il codice più chiaro senza cambiare la promessa

L'esempio mostra anche una scelta di struttura: `leggi` governa il flusso, `aggiungi` contiene la regola della riga, `main` offre una dimostrazione. Una classe con un solo metodo di centinaia di righe che legge file, valida, salva, registra e stampa sarebbe più difficile da provare. Parliamo di *coesione* quando una parte del codice ha una responsabilità comprensibile; di *accoppiamento* quando una parte dipende da dettagli di un'altra. Non misuriamo qualità contando solo i metodi: dividere senza motivo può disperdere la logica quanto ammucchiarla.

Un *refactoring* modifica la struttura interna mantenendo il comportamento osservabile concordato. Per esempio, potremmo sostituire il `StringBuilder` con un componente `LettoreRigheLimitate`, purché i test sul limite, sulle righe vuote e sulla chiusura continuino a passare. Introdurre invece il limite di ottanta unità è una modifica funzionale: cambia gli input accettati e richiede una decisione di requisito, non può essere nascosta sotto il nome «pulizia del codice». Fare piccoli passi e conservare un commit leggibile per ciascuna decisione aiuta a distinguere l'errore introdotto dalla correzione intenzionale.

La duplicazione non è sempre il nemico principale. Due righe simili in contesti diversi possono rappresentare contratti diversi: una normalizzazione per i titoli del catalogo e una per i nomi di file non devono diventare per forza la stessa funzione generica. Cerchiamo prima quale conoscenza è davvero ripetuta. Nomi come `MAX_RIGA`, `MAX_TITOLI`, `aggiungi` e `sorgente` rendono visibile la scelta; un nome generico come `processa` costringerebbe il lettore a ricostruirla dal corpo del metodo. Il codice chiaro è uno strumento di sicurezza perché facilita la revisione, ma la leggibilità da sola non prova che ogni confine sia difeso.

Il compilatore e gli analizzatori: segnali, non verdetti

`javac -Xlint:all` abilita avvisi su varie categorie. Un warning non è automaticamente un difetto sfruttabile; è una richiesta di attenzione che va collegata a codice e rischio. Un warning di deprecazione può richiedere migrazione, mentre un warning generato da un esempio storico isolato può essere documentato. Una soppressione con `@SuppressWarnings` è una decisione locale: dovrebbe nominare la categoria pertinente e spiegare perché il caso è accettato. Sopprimere tutto in una configurazione globale rende più difficile notare un warning nuovo.

Un *linter* controlla convenzioni; un analizzatore di bug pattern cerca forme associate a difetti; uno strumento SAST, cioè di analisi statica della sicurezza, cerca flussi e usi pericolosi senza eseguire il programma. Checkstyle, PMD e SpotBugs sono esempi esterni al JDK, con scopi e regole differenti. Una regola può dare falsi positivi e un controllo pulito può lasciare falsi negativi. Per questo scegliamo un insieme gestibile, salviamo la configurazione con il codice e annotiamo le eccezioni motivate. Il gate non è «zero avvisi qualunque sia il costo»: è sapere quali avvisi rimangono, chi li ha valutati e quale decisione è stata presa.

L'analisi statica non vede tutto ciò che accade in esecuzione, come un plugin caricato da configurazione o una particolare combinazione di input e permessi. I test non vedono tutti i percorsi possibili. Una revisione umana può individuare un requisito sbagliato ma può non osservare una race condition rara. Queste prove si completano, ciascuna con un limite dichiarato. La tabella che accompagna una correzione dovrebbe dire quale proprietà viene difesa e quale prova la controlla, non solo elencare strumenti «passati».

Il confine del file e degli altri dati non fidati

Il nostro parser riceve caratteri da un `Reader`; non decide se un percorso è lecito. Se un servizio riceve un nome di file da un utente, `Path.resolve` e `normalize` costruiscono e semplificano percorsi, ma non dimostrano da soli che il file aperto resti dentro una cartella consentita. Segmenti `..`, percorsi assoluti, collegamenti simbolici e modifiche concorrenti del file system possono invalidare una verifica ingenua «il percorso normalizzato inizia con la cartella base». La strategia dipende dal requisito: spesso è meglio non accettare un percorso libero, ma un identificatore che il server traduce in un file già noto. Se il file system è modificabile da un avversario, controllo e apertura devono essere progettati insieme con API e permessi appropriati; il capitolo 25 aveva già distinto il nome dal file reale.

Lo stesso principio vale per un comando esterno. Costruire una stringa come `"programma " + inputUtente` e passarla a una shell mette l'input dentro la sintassi della shell. `ProcessBuilder` con programma e argomenti separati evita quella particolare interpretazione, ma non garantisce che l'argomento sia semanticamente sicuro per il programma chiamato. Per XML non fidato occorre configurare il parser rispetto a entità esterne e risorse accessibili, non assumere che «è XML» significhi «è solo testo». Per URL HTTP il client deve decidere host consentiti, redirect, timeout e dimensione delle risposte in base al contesto. La domanda comune è sempre: quale parte del dato controlla un altro soggetto e quale capacità le stiamo concedendo?

La serializzazione Java degli oggetti merita un'attenzione distinta. Un `ObjectInputStream` che legge byte non fidati può costruire un grafo di oggetti e attivare comportamenti durante la deserializzazione. Per nuove interfacce fra sistemi preferiamo formati con schema o parsing esplicito, insieme a limiti di dimensione e validazione. Se dobbiamo mantenere un formato serializzato esistente, `ObjectInputFilter` e la configurazione dei filtri possono ridurre le classi e le dimensioni accettate; non convertono un flusso sconosciuto in un input innocuo per definizione. Non mostriamo un esempio positivo di deserializzazione arbitraria che il lettore potrebbe copiare fuori dal suo contesto.

> **Nota bene – Segreti e log.** Una password, una chiave API o un token non devono essere incorporati nel sorgente o stampati in un'eccezione per comodità. Spostarli in una variabile d'ambiente o in un archivio dedicato non elimina il bisogno di controllare accessi, rotazione e visibilità durante l'esecuzione. Anche un log apparentemente innocuo può includere percorsi personali, titoli riservati o corpi HTTP. Nel nostro esempio registriamo soltanto il numero di titoli importati, non il contenuto del file.

> **Riferimento essenziale – Novità di sicurezza nel JDK 25.** La Key Derivation Function API è stabile e offre un contratto standard per derivare chiavi crittografiche da materiale segreto e altri dati. Risponde a un problema diverso dal limitare le righe dell'importatore o dal nascondere un token nel log: quando un'applicazione deve ottenere una chiave per un protocollo, algoritmo, parametri e gestione del segreto vanno scelti e verificati nel loro contesto. La stessa guida segnala l'API per codificare oggetti crittografici in formato PEM come **preview**. Qui le registriamo per collocare correttamente le novità di Java 25, senza trasformare un esempio di importazione di libri in una ricetta crittografica incompleta. Un uso reale richiederebbe un caso di sicurezza definito, i contratti delle API e prove specifiche.

Dipendenze: il progetto include anche ciò che non abbiamo scritto

Nel POM del capitolo 29 JUnit è una dipendenza di test, con una versione fissata. Anche una libreria usata solo per i test va aggiornata e acquisita da un'origine affidabile; una libreria di produzione incide inoltre sul programma consegnato. Leggere il grafo transitivo, registrare versioni e provenienza, verificare integrità degli artefatti e osservare avvisi di vulnerabilità note sono compiti ricorrenti. Una *distinta base software*, spesso chiamata SBOM, elenca componenti e versioni di una build; è utile per rispondere rapidamente a «questa versione vulnerabile è presente qui?». Non dimostra da sola che il componente sia stato usato in un percorso pericoloso né che il pacchetto sia sicuro.

Aggiornare una dipendenza è un cambiamento di programma, non una formalità. Si leggono note di versione e compatibilità, si esegue la build pulita, si verificano i casi di uso, si aggiorna la distinta e si conserva una decisione sul rischio residuo. Se una vulnerabilità segnalata riguarda codice irraggiungibile nel nostro caso, la valutazione va registrata con evidenze, non nascosta togliendo il controllo. Se l'artefatto arriva da un repository non previsto, prima di adottarlo occorre capire chi lo pubblica e come viene verificato. Il capitolo 29 ha dato la base ripetibile per fare queste domande a un insieme preciso di file.

Registrare eventi utili senza confondere log e risultato

`System.getLogger` restituisce un logger dell'API di piattaforma. L'implementazione effettiva può inoltrare i messaggi a un backend diverso secondo l'ambiente; non ci affidiamo alla forma testuale del messaggio come interfaccia del programma. In `ImportazioneControllata`, `LOG.log(Level.INFO, "Importati {0} titoli", titoli.size())` registra un fatto utile a fine importazione senza ripetere nel log i titoli. Un errore di input potrebbe essere registrato da un livello superiore con identificatore della richiesta, categoria e causa, evitando di pubblicare tutto il contenuto rifiutato. Se ogni metodo registra e rilancia lo stesso errore, un singolo fallimento può generare molte righe identiche: prima decidiamo chi possiede la responsabilità del log.

I livelli aiutano a distinguere rumore ordinario, eventi informativi, problemi recuperabili e fallimenti. Il contesto di una richiesta, per esempio un identificatore opaco, aiuta a collegare più operazioni senza scrivere dati personali. Un log strutturato rende più facile filtrare campi rispetto a frasi libere, ma schema, retention e accessi ai log sono decisioni del sistema, non di `System.Logger` da sola. In una prova automatica confrontiamo il comportamento dell'operazione, non la data o la lingua scelta dal backend del logger. Quando il log è esso stesso un requisito di audit, lo testiamo con un contratto dedicato e un provider controllato.

Dal messaggio al comportamento della JVM

Supponiamo che l'importazione sia corretta ma lenta. Il log «Importati 2 titoli» non dice dove sia passato il tempo. Un *thread dump* mostra dove si trovano i thread in un istante; `jcmd <pid> Thread.print` è uno dei comandi diagnostici che la JVM può offrire. Un singolo dump può trovare un'attesa evidente, ma non racconta da solo quanto spesso sia accaduta. Java Flight Recorder, o JFR, raccoglie eventi nel tempo. Può mostrare attività della JVM, attese e eventi applicativi che definiamo noi. La diagnostica ha un costo e può contenere dati sensibili: in produzione si scelgono impostazioni, durata e accessi secondo il problema.

Il programma `DiagnosiImportazione` definisce un evento `esempio.catalogo.Importazione` con il numero di titoli. Avvia una registrazione JFR, abilita l'evento senza soglia minima di durata per questo esempio, importa tre volte lo stesso piccolo testo e salva il file indicato sulla riga di comando. `evento.begin()` segna l'inizio della misura; `commit()` conclude e registra l'evento. Non inseriamo nel JFR i titoli, soltanto il loro numero. Il programma non è un benchmark: le tre durate dipendono da avvio, JIT, sistema operativo e carico.

```
import java.io.StringReader;
import java.nio.file.Path;
import java.time.Duration;
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.Recording;

public final class DiagnosiImportazione {
    @Name("esempio.catalogo.Importazione")
    @Label("Importazione del catalogo")
    static class EventoImportazione extends Event {
        @Label("Titoli")
        int titoli;
    }

    private DiagnosiImportazione() {
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            throw new IllegalArgumentException("Uso: DiagnosiImportazione <file.jfr>");
        }
        try (Recording registrazione = new Recording()) {
            registrazione.enable(EventoImportazione.class)
                    .withThreshold(Duration.ZERO);
            registrazione.start();
            for (int i = 0; i < 3; i++) {
                EventoImportazione evento = new EventoImportazione();
                evento.begin();
                evento.titoli = ImportazioneControllata.leggi(
                        new StringReader("Java moderno\nReti\n")).size();
```

```
                evento.commit();
            }
            registrazione.stop();
            registrazione.dump(Path.of(args[0]));
        }
        System.out.println("Registrate 3 importazioni");
    }
}
```

Dalla cartella `esempi/C30`, dopo la compilazione già descritta, possiamo usare `java -cp build DiagnosiImportazione build/catalogo.jfr` e poi `jfr print --events esempio.catalogo.Importazione build/catalogo.jfr`. Nella registrazione verificata sono presenti tre eventi con `titoli = 2`. La figura 30.3 è una lettura annotata di quel risultato: cerchiamo il nome dell'evento, il campo di dominio, il thread e le durate variabili. Non usiamo i valori temporali della figura come soglia per un altro computer.

Tre eventi della prova JFR

Evento	Titoli	Thread	Durata
esempio.catalogo.Importazione	2	main	12,2 ms
esempio.catalogo.Importazione	2	main	0,718 ms
esempio.catalogo.Importazione	2	main	0,228 ms

Durate della singola esecuzione: non sono una soglia né un benchmark.

Fonte: registrazione riproducibile del programma DiagnosiImportazione, 2 ottobre 2026.

Figura 30.3 – Tre eventi `esempio.catalogo.Importazione` registrano due titoli ciascuno; le durate osservate sono dati della singola prova, non promesse di prestazione.

`jcmd` e JFR rispondono a domande diverse. Se il processo è bloccato adesso, un dump può indicare l'attesa corrente. Se la lentezza appare a intervalli, una registrazione può mostrare distribuzione degli eventi, allocazioni e attese nel periodo osservato, a seconda delle impostazioni. Se manca il nostro evento, controlliamo che fosse abilitato, che la registrazione coprisse il momento giusto e che il filtro `--events` indichi il nome corretto. «Non vedo eventi» non equivale immediatamente a «il codice non è stato eseguito». Il modulo `jdk.jfr` fa parte del JDK usato nella prova; un runtime ridotto creato con `jlink` deve includerlo se l'applicazione usa direttamente questa API.

Misurare una prestazione significa dichiarare un esperimento

Il capitolo 20A aveva messo in guardia dal concludere che `LongAdder` sia sempre più veloce di `AtomicLong`. Lo stesso vale per il parser. Cronometrare una sola chiamata a `ImportazioneControllata.leggi` con `System.nanoTime()` mescola avvio della JVM, compilazione JIT, lettura dell'input e rumore del sistema. Un microbenchmark serio richiede warm-up, più misure, controllo dei dati consumati e un ambiente dichiarato. JMH, lo strumento esterno del progetto OpenJDK per i benchmark sulla JVM, aiuta a costruire quell'esperimento; non trasforma un confronto mal posto in una risposta utile. Se il problema reale è il tempo per importare un archivio di cento megabyte su disco, occorre anche una prova di carico rappresentativa, non solo la velocità di una funzione su due righe in memoria.

Una prestazione ha più dimensioni: latenza di una richiesta, throughput di molte richieste, memoria, CPU, code e pause. Ottimizzare la media mentre una piccola quota di utenti attende troppo può peggiorare l'esperienza. Prima di cambiare codice, descriviamo carico e misura, riproduciamo il collo di bottiglia, confrontiamo una modifica alla volta e conserviamo i risultati con macchina, JDK e configurazione. Una registrazione JFR può suggerire dove indagare; un benchmark o un test di carico può verificare se la modifica aiuta nel contesto scelto. Nessuno dei due decide al posto nostro quale proprietà conta per il lettore o per l'utente.

Un incidente diventa conoscenza quando lasciamo una prova

Immaginiamo che un'applicazione in produzione termini durante l'importazione di un file molto lungo. Prima di cambiare una riga, raccogliamo condizioni: versione dell'artefatto, JDK, dimensione e formato dell'input, messaggio di errore, log pertinenti, eventuale registrazione JFR e limiti della macchina. Riduciamo il caso a un file che riproduce il fallimento senza conservare dati riservati. Se il problema è la riga illimitata, scriviamo una prova che la supera e osserviamo il difetto nella versione precedente; poi introduciamo il limite, eseguiamo di nuovo e controlliamo che gli input validi continuino a funzionare. La modifica entra in una build che un collega può ripetere grazie al capitolo 29.

Non ogni incidente nasce da una sola riga difettosa. Un servizio remoto può aver rallentato, una dipendenza può aver cambiato comportamento, un file system può aver esaurito spazio. Il rapporto finale distingue causa osservata, ipotesi escluse, correzione, prova di regressione e rischio residuo. Se la causa non è ancora dimostrata, la scriviamo come ipotesi invece di trasformarla in certezza per chiudere il ticket. Questa disciplina rispetta chi dovrà mantenere il programma dopo di noi.

Le attività dei capitoli 29–30 chiedono di riconoscere in quale confine nasce un difetto, aggiungere un test che lo renda visibile e scegliere una diagnosi adeguata. Il trasferimento finale cambia ancora il problema: al posto di un file arrivano righe da una risposta HTTP e il catalogo deve registrare su database soltanto l'insieme completo e valido. I limiti del parser restano utili, ma non bastano. Occorre definire timeout e dimensione della risposta, errore di rete, transazione dell'archivio e

comportamento in caso di retry. Il lettore ha costruito il modello quando sa estendere le domande al nuovo confine senza credere che una build verde o una classe chiamata «sicura» risolva tutto.

> **Domanda finale – Quale prova manca?** Per ogni modifica chiediamo: quale problema concreto affronta, quale invariante o rischio governa, come vediamo che funziona e quale caso vicino la farebbe fallire? Il buon codice non è un trofeo statico. È un insieme di decisioni che altri possono leggere, verificare e migliorare senza dover indovinare perché furono prese.

Riferimenti essenziali

> **Riferimento essenziale – Contratti e diagnosi.** La specifica di `System.Logger` chiarisce l'API di logging; la guida Java 25 ai filtri della serializzazione e la specifica di `ObjectInputFilter` descrivono il perimetro difensivo per codice legacy. Le specifiche di `jcmd`, `jfr`, `Event` e `Recording` sostengono la prova diagnostica. Il progetto OpenJDK JMH documenta lo strumento per i benchmark. Nessuna di queste fonti fornisce da sola una politica di sicurezza o una soglia di prestazione valida per ogni applicazione: tali decisioni dipendono dal sistema e dalle prove raccolte.

Attività e soluzioni ragionate C01-C06

Queste attività accompagnano i capitoli C01–C06. Ogni consegna dichiara che cosa osservare; le soluzioni sono esempi motivati, da confrontare con il proprio tentativo. Tutto il codice del percorso principale usa Java 25 senza preview. Per compilare i sorgenti e controllare gli output in un ambiente pulito si può eseguire `verifica-primi-sei.py` dalla radice del progetto.

C01-A01 – Scegliere il percorso e verificare l'ambiente

Consegna. Scrivi in tre frasi che cosa conosci già di programmazione, quale sezione del libro leggerai per prima e quale risultato ti farà dire che hai capito il capitolo. Poi esegui `java --version` e `javac --version` e conserva i due output insieme alla data. Non inserire percorsi personali o dati riservati nella consegna.

Criterio. Il percorso scelto deve essere compatibile con i prerequisiti dichiarati. Entrambi i comandi devono essere disponibili e mostrare una versione coerente del JDK. L'attività non assegna un voto alle conoscenze di partenza: serve a scegliere da dove cominciare.

Suggerimenti. Se non sai scegliere, parti dai capitoli C02 e C03 e rimanda i box di approfondimento. Se un comando non viene trovato, distingui il JDK assente dal percorso degli eseguibili non configurato. Se le versioni differiscono, controlla quale installazione viene trovata per prima dal terminale.

Soluzione commentata. Una risposta possibile è: «Conosco variabili e cicli, ma non ho mai usato Java. Leggo C02-C05 in ordine e provo ogni programma completo. Considero capito C03 quando so spiegare perché `javac` e `java` sono comandi diversi». Gli output dipendono dal computer, quindi non esiste una stringa da copiare. La prova osservabile è poter ripetere i due comandi e spiegare che il secondo verifica il compilatore.

C01-G01 – Usare le convenzioni del libro

Consegna. Apri la sezione sulle convenzioni e scegli un programma completo, un frammento di codice, una nota e un approfondimento nei capitoli C02-C06. Per ciascuno scrivi quale operazione ti è richiesta: compilarlo, inserirlo in un contesto, usarlo per evitare un errore o rimandarlo dopo il percorso principale. Spiega anche dove cercheresti il sorgente di un programma completo.

Criterio. La risposta distingue un programma avviabile da un frammento e non tratta note e approfondimenti come istruzioni da copiare nel terminale. Il sorgente completo viene ritrovato nei materiali del capitolo.

Suggerimenti. Cerca un blocco che dichiara `public class` e uno che mostra soltanto poche righe. Leggi il paragrafo prima e dopo ciascun blocco.

Soluzione commentata. Un programma completo si salva nel file indicato e si verifica con i comandi riportati; un frammento richiede il contesto di una classe o di un metodo. Una nota richiama un rischio concreto, mentre un approfondimento

aggiunge un dettaglio che si può leggere dopo il percorso principale. I file in `esempi` sono la versione da compilare dei programmi completi.

C02-A01 – Riconoscere responsabilità e relazioni

Consegna. Un lettore prende in prestito un libro. Indica in quale oggetto registreresti la data del prestito e perché. Poi completa a parole la differenza fra «Biblioteca è una Collezione» e «Biblioteca ha una collezione». Non serve scrivere codice.

Criterio e feedback. La risposta deve collegare la data all'evento del prestito e distinguere ereditarietà da composizione. Se la data finisce nel solo `Libro`, chiediti come rappresenteresti due prestiti dello stesso libro in momenti diversi. Se usi «è un» soltanto per riusare metodi, rileggi il paragrafo sulle responsabilità.

Soluzione commentata. `Prestito` collega un libro a un lettore in un intervallo di tempo; perciò conosce la data iniziale. Dire «Biblioteca ha una collezione» descrive una parte del suo stato, mentre «Biblioteca è una Collezione» attribuisce alla biblioteca il tipo e il contratto di una collezione. La seconda frase richiede una giustificazione più forte del semplice riuso.

C02-P01 – Modellare una biblioteca

Consegna. A partire da un lettore che chiede un libro e lo restituisce, proponi almeno tre entità del programma. Per ciascuna indica una responsabilità, un dato che conosce e un'operazione che compie. Disegna le relazioni fra le entità. Poi valuta la frase «La biblioteca estende la collezione di libri perché così riusa i metodi della collezione».

Criterio. Il modello deve distinguere `Libro`, `Lettore` e `Prestito`; una relazione di ereditarietà deve significare davvero «è un». Una regola deve impedire uno stato impossibile, per esempio due restituzioni dello stesso prestito senza una decisione esplicita. La motivazione vale più di una gerarchia unica.

Suggerimenti. Chiediti a quale evento appartiene una data di restituzione. Prova a leggere ad alta voce le relazioni come frasi. Se «Biblioteca è una Collezione» sembra forzato, valuta «Biblioteca ha una collezione».

Soluzione commentata. `Libro` conosce titolo e autore; `Lettore` identifica chi prende in prestito; `Prestito` collega i due e conosce inizio ed eventuale restituzione. La biblioteca coordina ricerca e registrazione e **ha** una collezione di libri. L'ereditarietà proposta nella consegna confonde riuso del codice e appartenenza concettuale. Una verifica umana applica la rubrica del piano delle esercitazioni: correttezza, motivazione, qualità del modello e robustezza delle regole.

C02-D01 – Una gerarchia costruita per riusare codice

Consegna. Un progetto propone `Prestito extends Libro` perché entrambi hanno una data, e `Biblioteca extends ArrayList<Libro>` perché la biblioteca conserva libri. Individua i due rapporti concettualmente fragili e riscrivili in prosa senza usare ereditarietà dove non c'è un rapporto «è un».

Criterio. La diagnosi distingue il prestito dal libro e la biblioteca dalla collezione che usa. Non basta dire che `extends` è sintatticamente possibile: occorre spiegare quali promesse farebbe ai chiamanti.

Suggerimenti. Leggi ad alta voce «un prestito è un libro» e «una biblioteca è una lista». Chiediti dove appartiene la data di restituzione.

Soluzione commentata. `Prestito` collega libro e lettore e conosce le date dell'operazione; **ha un** riferimento al libro. `Biblioteca` coordina un catalogo e **ha una** collezione, ma non deve esporre automaticamente tutte le operazioni di `ArrayList`. La composizione rende più chiari i due contratti.

C03-A01 – Sorgente, classe, esecuzione

Consegna. Metti nell'ordine di lavoro `Saluto.java`, `javac`, `Saluto.class`, `java` e JVM. Spiega poi perché cambiare soltanto il file `.java` non modifica l'output del programma già compilato.

Criterio e feedback. La spiegazione deve distinguere il file sorgente dal file compilato e il compilatore dal programma di avvio. Se attribuisci a `java` la produzione del `.class` nell'esempio del capitolo, ripercorri i due comandi stampati in C03.

Soluzione commentata. `javac` legge `Saluto.java` e produce `Saluto.class`; il comando `java` avvia la JVM, che carica la classe e ne esegue il metodo `main`. Una modifica al sorgente ha effetto su quel file, ma non riscrive da sola il `.class` già presente.

C03-G01 – Compilare, osservare, diagnosticare

Consegna. Entra nella cartella che contiene `Saluto.java` e `Argomenti.java`, poi compila il primo con `javac --release 25 -d build Saluto.java`. Avvialo con `java -cp build Saluto` e osserva la struttura della classe con `javap -classpath build Saluto`. Modifica soltanto la frase nel sorgente e riesegui senza ricompilare; infine ricompila e riprova. Compila anche `Argomenti.java` con `javac --release 25 -d build Argomenti.java` ed esegui `java -cp build Argomenti primo`. Spiega quali parti della riga di comando sono opzioni del comando `java` e quale parte è l'argomento del programma.

Criterio. L'output dopo la prima compilazione è `Ciao, Java 25!`. `javap` mostra il metodo `main`. La modifica al solo `.java` non cambia il comportamento del `.class` già prodotto; la ricompilazione lo aggiorna. Il secondo programma stampa `Primo argomento: primo`. Registra comandi, versione JDK e codici di uscita.

Suggerimenti. Se `java` non trova la classe, verifica la cartella passata a `-cp`. Se `javac` non trova il sorgente, verifica la cartella da cui lanci il comando. Se modifichi una parola chiave e il compilatore fallisce, sei ancora nella fase di compilazione: la JVM non ha eseguito il nuovo programma.

Soluzione commentata. `javac` legge il testo e produce `build/Saluto.class`; `java` legge il file compilato nella cartella indicata. Per questo un cambiamento nel sorgente diventa osservabile soltanto dopo un'altra compilazione. `javap` non ricostruisce il testo originale, ma mostra la struttura della classe compilata e rende visibile la presenza di `main`. Nella seconda esecuzione `-cp build` è un'opzione di

`java`, `Argomenti` è il nome della classe da avviare e `primo` è il testo consegnato al suo metodo `main`.

C03-D01 – Un file compilato che non viene trovato

Consegna. In una cartella pulita compila `Saluto.java` con `javac --release 25 -d build Saluto.java`. Se lanci `java -cp . Saluto` dalla cartella che contiene solo il sorgente e `build/`, il programma non trova la classe. Spiega perché e correggi il comando senza ricompilare.

Criterio. La risposta indica che il file `Saluto.class` è in `build/` e usa `java -cp build Saluto`. Non attribuisce l'errore al contenuto del metodo `main`.

Suggerimenti. Osserva l'effetto dell'opzione `-d build` e il ruolo di `-cp` nel comando di avvio.

Soluzione commentata. `javac` ha prodotto `build/Saluto.class`; `-cp .` cerca la classe nella cartella corrente e non dentro `build/`. `java -cp build Saluto` indica dove trovare la classe già compilata. Non serve cambiare il sorgente.

C04-A01 – Prevedere tipo e indice

Consegna. Senza eseguire codice, indica il primo indice valido di `int[] voti = {20, 25, 30};`, il valore di `voti.length` e il tipo dell'espressione `a + b` quando `a` e `b` sono variabili `byte`. Spiega in una frase perché `var` non rende dinamico il tipo di una variabile locale.

Criterio e feedback. Le tre risposte sono `0`, `3` e `int`. Se confondi lunghezza e ultimo indice, disegna tre caselle e scrivi sopra `0`, `1`, `2`. Se pensi che `var` permetta di cambiare tipo in seguito, rileggi la distinzione fra tipo inferito e tipo effettivo.

Soluzione commentata. Gli indici dell'array partono da zero, quindi la terza casella ha indice `2` pur essendoci tre elementi. Nell'addizione i due `byte` sono promossi a `int`. Con `var` il compilatore ricava un tipo dall'inizializzatore; quel tipo resta quello della variabile per tutta la sua durata.

C04-P01 – Quattro voti e una conversione

Consegna. Modifica `MediaVoti.java` usando `{20, 25, 28, 30}` e calcola la media. Prima di eseguire, prevedi il risultato. Poi spiega che cosa cambierebbe togliendo `(double)` e perché `byte c = a + b;` può non compilare anche quando `a` e `b` sono piccoli. Infine esegui `ConversioniNumeriche.java` e confronta l'output con la tua previsione.

Criterio. La media è `25.75`; la somma include quattro elementi. La spiegazione distingue divisione intera, conversione esplicita e promozione di `byte` a `int`. I valori osservati in `ConversioniNumeriche` sono confrontati con quelli attesi nel capitolo.

Suggerimenti. Somma prima i quattro numeri su carta. Controlla il tipo dell'operando sinistro di `/`. Ricorda che il compilatore decide il tipo di `a + b` usando le regole dell'espressione, non la grandezza dei valori in un'esecuzione particolare.

Soluzione commentata. `20 + 25 + 28 + 30` vale `103`; `103 / 4.0` vale `25.75`. Senza il cast, `103 / 4` produce l'intero `25`, che viene poi convertito in `25.0` se assegnato a un `double`. Un'espressione aritmetica con due `byte` viene promossa a `int`, quindi l'assegnazione diretta a `byte` richiede un controllo o una conversione, che può

perdere informazione. Nel programma sulle conversioni `Long.MAX_VALUE` ristretto a `int` diventa `-1`; l'allargamento di `Integer.MAX_VALUE` a `long` conserva il valore.

C04-D01 – Un indice oltre l'ultimo elemento

Consegna. Valuta il frammento `int[] voti = {20, 25, 30}; for (int i = 0; i <= voti.length; i++) System.out.println(voti[i]);`. Indica le righe che compaiono prima dell'errore, il primo indice non valido e la correzione minima della condizione del ciclo.

Criterio. Sono stampati `20`, `25` e `30`; l'accesso con indice `3` fallisce durante l'esecuzione. La condizione diventa `i < voti.length`.

Suggerimenti. Scrivi gli indici sotto i tre elementi. Il valore di `length` conta le caselle ma non è un indice valido in questo array.

Soluzione commentata. Gli indici validi sono `0`, `1` e `2`. Con `<=`, il ciclo prova anche `i == 3`; l'array non ha quella casella e Java lancia `ArrayIndexOutOfBoundsException`. Con `<` il ciclo si ferma dopo l'ultimo elemento.

C04-P02 – Rappresentazione, tipo e copia

Consegna. Prima di eseguire `BitPassoPasso.java`, prevedi i risultati di `100 >> 1`, `100 ^ 125`, `~100` e `-8 >>> 1`. Indica quali risultati si capiscono osservando soltanto gli otto bit finali e quali richiedono di ricordare che il tipo è `int`. Poi crea `int[] primo = {24, 27}; int[] secondo = primo;`, assegna `30` a `secondo[0]` e prevedi il valore di `primo[0]`. Ripeti dopo aver ottenuto `secondo` con `primo.clone()`.

Criterio e feedback. I quattro valori sono `50`, `25`, `-101` e `2_147_483_644`; `~` e `>>>` sul valore negativo richiedono la rappresentazione completa dell'`int`. Nel primo caso i due nomi indicano lo stesso array e `primo[0]` diventa `30`; dopo la copia resta `24`.

Soluzione commentata. Lo shift destro di `100` produce `50`; XOR conserva i bit diversi di `100` e `125`, ottenendo `25`. `~` inverte tutti i 32 bit e `>>>` inserisce zeri a sinistra, perciò gli otto bit disegnati sul foglio non bastano per i due risultati. L'assegnazione di un riferimento a un altro non copia gli elementi; `clone()` crea qui un nuovo array di valori primitivi indipendente dal primo.

C05-A01 – Scelta o ripetizione?

Consegna. Scegli una struttura di controllo per ciascun caso e spiega la scelta: assegnare un'etichetta a un numero da `1` a `7`; continuare a chiedere un dato finché non è valido; visitare ogni voto di un array. Per il primo caso indica anche dove metteresti il valore da usare quando il numero non è nell'intervallo.

Criterio e feedback. Devono comparire una scelta con `default`, un ciclo e una visita dell'array. Se usi `if` per ogni voto già noto nell'array, chiediti quale parte del codice cambierebbe quando i voti diventano cento.

Soluzione commentata. Una espressione `switch` rende visibili i sette casi e un `default` copre gli altri interi. Un `while` ripete la richiesta finché la condizione resta vera; la condizione va definita in modo che il ciclo possa terminare. Un `for` esteso visita i voti quando non serve conoscere l'indice.

C05-G01 – Dalla scelta ai cicli

Consegna. Esegui `MediaVotiConCiclo.java` con i tre voti iniziali `{24, 27, 30}`, poi con `{20, 25, 28, 30}` e infine con un array vuoto. Modifica l'array nel sorgente e ricompila fra una prova e l'altra; prima di eseguire, predici ogni output. Poi osserva `DamaIniziale.java`: spiega perché produce dodici pedine per colore e quali due righe della tavola restano vuote. Riscrivi infine una classificazione di giorno feriale/fine settimana con un'espressione `switch`, usando i casi `1`-`7` e un `default` per gli altri interi.

Criterio. Nessun caso vuoto provoca una divisione; la media non dipende dalla lunghezza fissata nel sorgente; lo `switch` produce un valore per ogni intero; la tavola conta dodici pedine bianche e dodici nere.

Suggerimenti. La guardia dell'array vuoto deve precedere la divisione. Il `for` esteso visita ogni valore senza usare un indice esplicito. Nella dama, una riga ha quattro caselle per cui `(riga + colonna) % 2 == 1`.

Soluzione commentata. I tre voti iniziali producono `Media: 27.0`; i quattro voti producono `Media: 25.75`. Con `int[] voti = {};` il programma stampa `Nessun voto disponibile` e termina il metodo prima della divisione. La dama riempie le righe `0`-`2` e `5`-`7`; le righe `3` e `4` restano vuote. Lo `switch` può usare `case 6, 7 -> "fine settimana"; case 1, 2, 3, 4, 5 -> "feriale"; default -> "valore non valido";`, come spiegato nel testo.

C05-G02 – Gli undici casi e i confini

Consegna. Esegui `InteroComeStringaComparato.java` con `0`, `3`, `10`, `-1` e `11`, annotando prima la previsione delle due righe per ogni input. Prova poi ad avviarlo senza argomenti. Aggiungi il caso `11` a entrambe le forme e spiega perché il ramo finale resta necessario.

Criterio e feedback. Le versioni `if` e `switch` concordano per tutti e cinque i valori; senza argomenti compare l'istruzione d'uso. Dopo l'aggiunta, `11` ha una parola esplicita, ma `-1` e `12` restano fuori intervallo. Se la prova del numero negativo stampa ancora «maggiore di dieci», il messaggio non descrive il dominio corretto.

Soluzione commentata. I primi tre valori producono `zero`, `tre` e `dieci` in entrambe le versioni; `-1` e `11` producono `fuori intervallo`. La guardia su `args.length` evita l'accesso a `args[0]` quando manca l'argomento. Aggiungendo `11`, il caso diventa parte del dominio, ma un programma che riceve un `int` può ancora ricevere molti altri valori: il ramo finale mantiene definito il comportamento ai confini.

C05-P01 – Pattern, record e casi dominati (avanzato)

Consegna. Parti da `EsitiConPattern.java`. Prima di eseguirlo, scrivi l'output previsto per un voto `27`, un voto `15`, un esito sospeso e `null`. Poi inverti i due casi `Valido` e osserva il messaggio del compilatore. Spiega perché uno dei casi non può più essere raggiunto. Ripristina l'ordine corretto e verifica la compilazione senza preview.

Criterio. I quattro rami sono esercitati; il caso con guardia precede quello generale; il `switch` è esaustivo sulla gerarchia `sealed` e tratta `null` esplicitamente. L'errore introdotto è descritto come dominanza, non come difetto della JVM.

Suggerimenti. Leggi `Valido(int voto)` come estrazione del campo dal record. Chiediti se un `Valido` con voto `27` corrisponde anche al pattern senza guardia. Un caso che cattura tutti i valori del successivo deve comparire dopo.

Soluzione commentata. L'output è `promosso: 27`, `da recuperare: 15`, `sospeso: documenti mancanti`, `assente`. Se `case Valido(int voto)` viene prima, comprende anche i valori per cui `voto >= 18` ed elimina il ruolo del caso più specifico. Il compilatore rileva questa dominanza. Il materiale resta etichettato avanzato perché record e gerarchie `sealed` vengono sviluppati nei capitoli successivi; qui interessa il controllo di flusso.

C05-D01 – Un `continue` che non fa avanzare il ciclo

Consegna. Senza eseguire il frammento `int i = 0; while (i < 3) { if (i == 1) continue; i++; }`, segui tre passaggi del ciclo. Spiega perché non termina e riscrivilo in modo che `i` raggiunga `3` anche quando salta il resto del corpo.

Criterio. La risposta identifica `i == 1` come punto fisso: `continue` ricomincia il `while` prima di `i++`. Una correzione sposta l'incremento prima del ramo oppure usa un `for` il cui aggiornamento avviene dopo `continue`.

Suggerimenti. Annota `i` all'inizio di ogni iterazione. Chiediti quale istruzione viene saltata quando il ramo è vero.

Soluzione commentata. Dopo la prima iterazione `i` vale `1`. Alla successiva il `continue` salta `i++`, quindi la condizione resta vera e il valore non cambia. Scrivere `i++; if (i == 1) continue;` nel corpo elimina quel punto fisso; in un programma reale occorre anche verificare che l'ordine delle altre operazioni resti quello voluto.

C06-A01 – Tre nomi, tre funzioni

Consegna. Spiega che cosa cambia quando sposti `Catalogo` in un altro package, quando aggiungi un `import` per `ArrayList` e quando un modulo dichiara `exports` per il package che contiene una classe pubblica. Indica quale di queste operazioni abbrevia un nome nel sorgente.

Criterio e feedback. Le risposte devono distinguere identità della classe, scrittura del riferimento e accessibilità fra moduli. Se rispondi che `import` sposta `ArrayList`, confronta il suo nome qualificato prima e dopo l'import: resta `java.util.ArrayList`.

Soluzione commentata. Cambiare il package di `Catalogo` cambia il suo nome qualificato e richiede di aggiornare dichiarazione, cartelle e comandi di avvio. L'import di `ArrayList` consente di scrivere solo `ArrayList` nel file che lo dichiara; la classe rimane in `java.util`. `exports` permette ai moduli che leggono il modulo del catalogo di accedere ai tipi pubblici del package esportato.

C06-G01 – Due moduli e un servizio

Consegna. Compila i sorgenti in `esempi/C06/servizio` dalla radice di `javamattone/revisione-java-25` con `javac --release 25 --module-source-path esempi/C06/servizio/src -d build/mods -m javamattone.catalogo,javamattone.app`. Avvia con `java --module-path build/mods -m javamattone.app/javamattone.app.Main`. Leggi i due `module-info.java` e indica chi usa il servizio, chi lo fornisce e quale package è esportato. Come diagnosi, rimuovi temporaneamente `exports`, poi `requires`, e confronta i due errori di compilazione. Ripristina i file alla fine.

Criterio. L'output positivo è `Ciao dal modulo catalogo`. L'applicazione dichiara `requires` e `uses`; il catalogo dichiara `exports` e `provides`. Il package dell'implementazione rimane interno. I due errori introdotti devono essere spiegati come mancanza di accessibilità e di leggibilità, rispettivamente.

Suggerimenti. `import` abbrevia un nome nel sorgente: non sostituisce `requires` e `exports`. `ServiceLoader` cerca un'implementazione dichiarata al sistema dei moduli. Se l'avvio fallisce, verifica che entrambe le cartelle dei moduli siano sotto il percorso passato a `--module-path`.

Soluzione commentata. Il modulo `javamattone.app` legge `javamattone.catalogo` e usa l'interfaccia `Saluto`. Il modulo del catalogo esporta `javamattone.catalogo.api` e fornisce `SalutoItaliano`, che appartiene a un package non esportato. `ServiceLoader` ottiene il servizio senza che l'applicazione nomini la classe concreta. Rimuovere `exports` impedisce all'altro modulo di usare l'interfaccia pubblica; rimuovere `requires` impedisce al modulo chiamante di leggere il modulo che la contiene. Non serve aprire il package interno con `opens` per questo esempio: `provides` dichiara la relazione necessaria.

C06-C01 – Decidere il confine di un catalogo

Consegna. Progetta due moduli: uno offre un catalogo di libri, l'altro lo usa. Il catalogo ha un package `api` con un'interfaccia pubblica e un package `interno` con l'implementazione. Scrivi a parole quali package esportare e quale modulo deve dichiarare `requires`. Motiva la scelta dal punto di vista di chi usa il catalogo.

Criterio. Il modulo del catalogo esporta soltanto il package dell'API; il modulo chiamante dichiara di leggere il catalogo. L'implementazione interna non diventa pubblica per caso. La risposta distingue visibilità del tipo Java e accessibilità attraverso il confine del modulo.

Suggerimenti. Parti dal codice che il chiamante deve scrivere e identifica il tipo che nomina. Un package interno può restare accessibile ai componenti del proprio modulo senza essere esportato.

Soluzione commentata. Nel `module-info.java` del catalogo si dichiara `exports catalogo.api;`; nel modulo dell'applicazione `requires catalogo;`. La classe dell'implementazione in `catalogo.interno` resta nascosta agli altri moduli. Se il programma vuole caricarla come servizio, si aggiungono `uses` e `provides` con un contratto esplicito, come nell'esempio guidato.

C06-D01 – Il percorso sbagliato all'avvio

Consegna. Hai compilato i moduli del programma guidato in `build/mods`. Un collega tenta `java -cp build/mods javamattone.app.Main` e riceve un errore di classe non trovata. Spiega perché il comando non rispecchia la struttura compilata e scrivi l'avvio corretto.

Criterio. La risposta usa `--module-path build/mods` e `-m javamattone.app/javamattone.app.Main`. Distingue il class path, che cerca classi, dal module path, che presenta moduli nominati al sistema.

Suggerimenti. Guarda le cartelle prodotte dall'opzione `-d` nella compilazione modulare e leggi il `module-info.java` del modulo da avviare.

Soluzione commentata. `build/mods` contiene directory di moduli compilati, non direttamente la classe nel percorso che il comando con `-cp` sta cercando. L'avvio è `java --module-path build/mods -m javamattone.app/javamattone.app.Main`: identifica il percorso dei moduli e il modulo con la sua classe principale.

Attività C07–C12

Questa raccolta accompagna i capitoli C07–C12: classi e oggetti, incapsulamento, ereditarietà e composizione, tipi di uso comune, stringhe ed eccezioni. La versione di riferimento è Java 25 senza preview.

C07-A01 – Classe, oggetto e riferimento

Consegna. Senza eseguire il programma `Libro.java`, indica quante dichiarazioni di classe, quante variabili di tipo `Libro` e quanti oggetti `Libro` vengono creati nel suo `main`. Spiega perché `primo == secondo` è vero.

Criterio. La risposta distingue la dichiarazione della classe dalle due variabili e dall'unica istanza creata con `new`. Il confronto è spiegato come identità dei riferimenti, non come confronto dei titoli.

Suggerimenti. Cerca le righe che contengono `new Libro`. Poi leggi l'assegnazione di `secondo` da destra a sinistra: quale valore riceve? La figura 16 offre il modello logico.

Soluzione commentata. Il file dichiara una classe `Libro`. Nel metodo `main` compaiono due variabili di tipo `Libro`, `primo` e `secondo`. Solo la prima riga contiene `new Libro`, quindi nasce un oggetto. La seconda assegna a `secondo` lo stesso riferimento conservato in `primo`; per questo il confronto con == produce `true`.

C07-G01 – Aggiungere un metodo al libro

Consegna. In una copia di `Libro.java`, aggiungi `public void registraPrestiti(int quanti)` che chiama `registraPrestito()` tante volte quante indica il parametro. Nel `main` sostituisci la singola chiamata con `primo.registraPrestiti(2)`. Prevedi l'output prima di compilare.

Criterio. Il metodo usa il parametro come limite di un ciclo; il titolo stampato resta lo stesso e il numero dei prestiti diventa 2. `primo == secondo` resta `true`, perché non è stato creato un secondo libro.

Suggerimenti. Parti da `for (int i = 0; i < quanti; i++)`. Il metodo nuovo può riusare quello già esistente, senza modificare direttamente il campo nel `main`.

Soluzione commentata. `registraPrestiti` esegue `for (int i = 0; i < quanti; i++) registraPrestito();`. Con `quanti` uguale a due, lo stesso oggetto aumenta il contatore due volte; `secondo.descrizione()` stampa `Java Mattone dopo Mattone - prestiti: 2`. Il confronto dei riferimenti resta vero. Nel capitolo 8 discuteremo come rifiutare un parametro negativo se l'operazione diventa parte di un contratto pubblico.

C07-P01 – Due libri con lo stesso titolo

Consegna. Modifica una copia di `Libro.java` in modo che `secondo` riceva il risultato di una nuova espressione `new Libro("Java Mattone dopo Mattone")`. Prima di compilare, prevedi le due righe stampate. Poi compila con `javac --release 25 -Xlint:all -d build Libro.java` dalla cartella della copia e avvia con `java -cp build Libro`. Spiega perché la stessa stringa del titolo non basta a rendere veri i riferimenti confrontati con ==.

Criterio. Il programma stampa `Java Mattone dopo Mattone - prestiti: 0` e `false`: il prestito registrato su `primo` non cambia il contatore dell'altro oggetto. La spiegazione distingue identità, stato e contenuto del titolo. Il comando di compilazione termina senza warning.

Suggerimenti. Conta le espressioni `new`. Disegna due riquadri, uno per ciascun oggetto, e scrivi il contatore iniziale in entrambi. La chiamata `primo.registraPrestito()` ha un destinatario preciso: segna su quale riquadro agisce.

Soluzione commentata. Dopo la modifica, `primo` e `secondo` puntano a due istanze diverse. Entrambe hanno lo stesso titolo, ma solo il contatore della prima passa a uno. `secondo.descrizione()` legge il contatore della seconda e stampa zero; `primo == secondo` è falso perché confronta l'identità. Nel capitolo C10 si vedrà come definire un criterio di uguaglianza del contenuto tramite `equals`, ma non va usato per rispondere alla domanda sull'identità.

C07-D01 – Un riferimento assente

Consegna. In una copia del `main`, inserisci `Libro assente = null;` seguito da `System.out.println(assente.descrizione());`. Prevedi se l'errore sarà segnalato dal compilatore o durante l'esecuzione, poi prova. Ripristina la copia originale alla fine.

Criterio. La compilazione riesce; la chiamata sul riferimento `null` provoca `NullPointerException` durante l'esecuzione. Il lettore sa indicare la riga che tenta la chiamata e spiega che `null` non identifica un oggetto `Libro`.

Suggerimenti. Il tipo di `assente` è valido e l'assegnazione di `null` è permessa. Chiediti quale oggetto dovrebbe eseguire `descrizione()`. Il capitolo C12 mostrerà come leggere lo stack trace senza nascondere il problema.

Soluzione commentata. Il compilatore può verificare che `Libro` abbia il metodo `descrizione`, ma non può garantire che una variabile di riferimento non sia `null` in ogni esecuzione. Quando la JVM arriva alla chiamata, manca l'istanza destinataria e lancia `NullPointerException`. Un controllo esplicito o un contratto che escluda `null` può prevenire l'errore; la scelta dipende dal significato del dato nel programma.

C08-A01 – Individuare l'invariante

Consegna. Leggi `RegistroVoti.java`. Scrivi le tre condizioni che devono essere vere dopo la costruzione di un registro valido. Indica quale riga impedisce al chiamante di cambiare i voti attraverso l'array passato al costruttore e quale impedisce di cambiarli attraverso il risultato di `voti()`.

Criterio. Il nome dello studente non è assente né vuoto, esiste almeno un voto e ogni voto è fra zero e trenta. Le due direzioni della copia sono riconosciute separatamente.

Suggerimenti. Cerca le istruzioni che lanciano `IllegalArgumentException`. Poi segui l'array dal parametro al campo e dal campo al valore restituito. La figura 17 rappresenta questi passaggi.

Soluzione commentata. Il costruttore rifiuta uno studente nullo o composto soltanto da spazi, un array nullo o vuoto e qualunque voto fuori intervallo.

`this.voti = voti.clone()` protegge l'ingresso; `return voti.clone()` protegge l'uscita. I campi `private` e `final` aiutano, ma non sostituiscono queste due copie.

C08-P01 – Dimostrare la copia difensiva

Consegna. Compila ed esegui `RegistroVoti.java` dalla cartella del sorgente con `javac --release 25 -Xlint:all -d build RegistroVoti.java` e `java -cp build RegistroVoti`. In una copia del file, elimina solo la copia nel costruttore e prevedi la media; ripristinala, elimina solo la copia nel metodo `voti()` e prevedi di nuovo la media. Prova entrambe le varianti separatamente. Infine passa un voto `31` al costruttore a un voto e descrivi quando e perché il programma si interrompe.

Criterio. Il programma originale stampa `Ada: 27.0`; senza copia all'ingresso stampa `Ada: 19.0`; senza copia all'uscita stampa `Ada: 18.0`. Il voto `31` è rifiutato prima che venga creato un registro valido. Le varianti sono isolate e il sorgente originale viene ripristinato.

Suggerimenti. Calcola le tre somme: `24+27+30`, `0+27+30`, `24+0+30`. Chiediti quale assegnazione nel `main` agisce sull'array condiviso in ogni variante. Se due cambiamenti sono presenti insieme, non puoi attribuire con sicurezza la differenza a una sola copia mancante.

Soluzione commentata. Le copie rendono indipendente l'array del registro dagli array che il chiamante può modificare. Senza la prima copia, `originali[0] = 0` cambia il dato interno e la media diventa `57/3`, cioè `19.0`. Senza la seconda, `ricevuti[1] = 0` cambia il dato interno e la media diventa `54/3`, cioè `18.0`. Il costruttore alternativo controlla `31` e lancia `IllegalArgumentException` prima di delegare; nel capitolo C12 vedremo come leggere e gestire correttamente l'eccezione.

C08-R01 – Togliere una validazione duplicata

Consegna. In una copia di `RegistroVoti.java`, estrai il controllo dell'intervallo `0-30` in un metodo `private static void validaVoto(int voto)`. Usalo sia nel ciclo del costruttore con array sia nel costruttore con un solo voto, prima di `this(...)`. Mantieni invariati l'output `Ada: 27.0` e il rifiuto del voto `31`.

Criterio. Il programma compila su Java 25 senza preview; i due costruttori applicano la stessa regola. La modifica ai costruttori non elimina le copie difensive né cambia la media dei dati validi.

Suggerimenti. Il metodo di supporto non deve leggere campi dell'istanza; per questo può essere `static` e può essere chiamato nel prologo del costruttore flessibile.

Soluzione commentata. Il metodo `validaVoto` lancia `IllegalArgumentException("voto fuori intervallo")` quando `voto < 0 || voto > 30`. Nel ciclo si chiama `validaVoto(voto);` nel costruttore singolo la chiamata precede `this(studente, new int[] {voto})`. L'array viene comunque ricontrollato nel costruttore delegato: la prima verifica serve a rifiutare l'argomento prima della delega, la seconda conserva il contratto del costruttore con array.

C09-A01 – Leggere due relazioni

Consegna. Nel programma dei veicoli, indica quali classi hanno una relazione «è un» con `Veicolo` e quale oggetto `Auto` conserva come parte. Spiega perché `Motore` non estende `Veicolo`.

Criterio. La risposta identifica `Auto` e `Bicicletta` come sottoclassi e `Motore` come componente dell'auto; collega la scelta al significato del dominio.

Suggerimento. Leggi `extends` e poi il tipo del campo `motore`. Confronta le due frecce nella figura 19.

Soluzione commentata. `Auto` e `Bicicletta` specializzano `Veicolo`. `Auto` conserva un `Motore` perché un motore è una sua parte; non è un tipo di veicolo.

C09-P01 – Dispatch e casi chiusi

Consegna. Prevedi le tre righe stampate da `DemoVeicoli.java`. Poi sostituisci la costruzione dell'auto con `new Bicicletta("B")`, esegui di nuovo e descrivi perché il ramo `instanceof Auto` non stampa una riga. In una copia separata elimina il caso `Bicicletta` dallo `switch` e compila.

Criterio. La prima esecuzione stampa `Auto A con motore elettrico`, `elettrico`, `auto`. La variante con bicicletta stampa la sua descrizione e `bicicletta`. La variante senza caso non compila per mancata esaustività.

Suggerimento. Il tipo dichiarato di `mezzo` resta `Veicolo`; segui la classe dell'oggetto creato. Leggi separatamente il controllo `instanceof` e lo `switch`.

Soluzione commentata. Il metodo ridefinito viene scelto in base all'oggetto concreto. Con `Bicicletta`, il pattern per `Auto` fallisce e il ramo viene saltato; lo `switch` sceglie `Bicicletta`. Togliere quel caso lascia scoperto un sottotipo ammesso dalla gerarchia sealed, quindi il compilatore rifiuta l'espressione.

C09-D01 – Costruire base e figlio

Consegna. Prevedi le due righe del programma sull'ordine dei costruttori. Cambia poi 7 in 12 e verifica quali righe cambiano.

Criterio. L'ordine resta `base`, poi `figlio: 12`: la parte base viene costruita prima del completamento del figlio.

Suggerimento. Parti da `new FiglioOrdine(12)` e segui `super()` prima dell'assegnazione al campo `valore`.

Soluzione commentata. Il costruttore della base stampa `base`; quello del figlio assegna il campo e stampa `figlio: 12`. Il valore cambia soltanto la seconda riga.

C09-R01 – Proteggere una gerarchia valida

Consegna. Rivedi una copia di `DemoVeicoli.java` affinché `Veicolo` rifiuti un nome nullo o vuoto e `Auto` rifiuti un `Motore` nullo. Mantieni le tre righe dell'esecuzione valida e verifica separatamente i due nuovi casi errati. Spiega in quale costruttore appartiene ciascun controllo.

Criterio. `new Bicicletta(" ")` e `new Auto("A", null)` non creano oggetti validi; l'esecuzione con `Auto("A", new Motore("elettrico"))` resta invariata. La sottoclasse non tenta di assegnare direttamente il campo privato `nome` della base.

Suggerimenti. La classe base è responsabile del proprio nome. Il controllo del motore è specifico di `Auto`. Ricorda che `super(nome)` precede l'assegnazione del campo della sottoclasse.

Soluzione commentata. Il costruttore `Veicolo` può lanciare `IllegalArgumentException` quando `nome == null || nome.isBlank()`, poi assegnare il campo. Nel costruttore di `Auto`, dopo `super(nome)`, si controlla `motore == null` prima di assegnarlo. Il percorso positivo usa gli stessi dati e stampa le stesse tre righe; i nuovi controlli rendono esplicito il contratto della gerarchia.

C10-A01 – Identità e valore

Consegna. Prima di avviare DemoValore.java, prevedi le prime quattro righe. Spiega perché la terza riga non basta, da sola, a dire che due etichette sono uguali.

Criterio. Le righe sono `false`, `true`, `true`, `Etichetta[Java]`. La risposta distingue identità, uguaglianza di dominio e hash, e sa che hash uguali possono avere oggetti diversi.

Suggerimento. Conta le espressioni `new Etichetta`; poi leggi da quali campi sono calcolati `equals` e `hashCode`.

Soluzione commentata. Due costruzioni danno due oggetti, perciò == è falso. I testi sono uguali e `equals` è vero. Gli hash devono coincidere per oggetti uguali, ma una collisione può far coincidere hash di oggetti non uguali. `toString` produce la descrizione leggibile.

C10-D01 – Un contratto rotto

Consegna. In una copia del sorgente sostituisci il corpo di `hashCode()` con `return System.identityHashCode(this);`. Spiega quale regola del contratto viene violata, indipendentemente dai numeri osservati in una singola esecuzione. Ripristina il sorgente.

Criterio. Il lettore riconosce che due istanze uguali secondo `equals` devono avere lo stesso hash, mentre un hash legato all'identità non garantisce questa proprietà.

Suggerimento. Non basare la conclusione su una coppia di numeri vista una sola volta: confronta le definizioni dei due metodi.

Soluzione commentata. `equals` confronta `testo`, ma `identityHashCode` usa l'identità degli oggetti. Il codice non garantisce più che oggetti uguali producano lo stesso hash; un'eventuale coincidenza numerica non ripara il contratto.

C10-P01 – Un'enumerazione che conosce una regola

Consegna. Esegui `DemoValore.java` e osserva l'ultima riga. Aggiungi nel `main` una stampa di `Stato.IN_PRESTITO.consentePrestito()`. Prevedi il risultato, poi modifica il metodo per accettare il prestito anche da un eventuale nuovo stato `RICONSEGNATO` e spiega perché la modifica resta dentro l'enumerazione.

Criterio. Prima della modifica il metodo restituisce `true` per `NUOVO` e `false` per `IN_PRESTITO`. Dopo aver aggiunto `RICONSEGNATO`, restituisce `true` anche per quel valore, senza confrontare stringhe nel chiamante.

Suggerimenti. Il metodo può restituire il risultato di una condizione booleana; non serve costruire un oggetto nuovo per ogni chiamata.

Soluzione commentata. Con due stati, `return this == NUOVO;` distingue il libro nuovo da quello già prestato. Dopo l'aggiunta del terzo stato, `return this == NUOVO || this == RICONSEGNATO;` esprime la nuova regola nel tipo che rappresenta gli stati. Il codice esterno chiede `consentePrestito()` senza conoscere i nomi interni delle costanti.

C11-A01 – Due misure di una stringa

Consegna. Spiega le righe `2` e `1` stampate da DemoStringhe.java per il simbolo U+1F600. Indica che cosa conta ogni metodo e perché nessuno dei due è, in generale, un contatore dei simboli percepiti dall'utente.

Criterio. `length()` conta due unità UTF-16; `codePointCount` conta un code point. La risposta riconosce la possibile presenza di sequenze di più code point per un unico simbolo visibile.

Suggerimento. Usa la figura 21 per seguire la coppia di unità. Pensa poi a una lettera e a un accento combinante.

Soluzione commentata. Il code point supplementare dell'esempio è codificato con due unità UTF-16. La differenza fra unità, code point e simboli percepiti emerge con testo che include combinazioni o emoji composte.

C11-P01 – Immutabilità osservabile

Consegna. Sostituisci la stampa di `testo.strip()` con la sola chiamata `testo.strip();`, poi ristampa `testo`. In una seconda prova scrivi `testo = testo.strip();` e ristampa. Motiva la differenza.

Criterio. La prima prova conserva gli spazi; la seconda stampa il testo pulito perché la variabile riceve il nuovo valore. Non si descrive `strip()` come modifica dell'oggetto originale.

Suggerimento. Segui quale riferimento viene assegnato a `testo` dopo la chiamata.

Soluzione commentata. Una `String` è immutabile. Chiamare `strip()` senza usare il risultato lascia `testo` com'era; assegnare il risultato fa riferire la variabile alla stringa pulita.

C11-D01 – Righe di un text block

Consegna. Compila DemoTextBlock.java e osserva le due righe stampate. Sposta il delimitatore di chiusura immediatamente dopo `Java!` e confronta l'output come sequenza di caratteri, includendo la possibile interruzione di riga finale.

Criterio. La risposta distingue testo visibile e carattere di fine riga; riconosce che la posizione del delimitatore influisce sul valore della stringa.

Suggerimento. Una stampa normale può nascondere la differenza finale: aggiungi `System.out.print("[fine]")` dopo il messaggio per renderla visibile.

Soluzione commentata. Nel sorgente originale la chiusura è su una nuova riga e il valore termina con un'interruzione di riga; `[fine]` compare sulla riga successiva. Con la chiusura subito dopo `Java!`, il valore non contiene quell'ultima interruzione e `[fine]` segue `Java!` sulla stessa riga.

C12-A01 – Ordine degli eventi

Consegna. Prevedi l'output del programma sulle risorse prima di avviarlo. Identifica l'eccezione principale e quella soppressa.

Criterio. Le quattro righe sono `lavoro`, `chiusura`, `principale: errore durante il lavoro`, `secondaria: errore durante la chiusura`. Il lettore spiega perché `close()` viene chiamato anche dopo il primo errore.

Suggerimento. Segui prima il corpo del `try`, poi la chiusura automatica, infine il `catch`.

Soluzione commentata. Il corpo chiama `usa()` e poi lancia l'errore di lavoro. Prima del `catch`, Java chiude la risorsa; anche `close()` lancia un errore, che viene conservato come soppresso. Il `catch` riceve l'errore del corpo e stampa entrambi.

C12-P01 – Se fallisce solo la chiusura

Consegna. In una copia del sorgente elimina soltanto il `throw new IOException("errore durante il lavoro")` dal corpo del `try`. Prevedi l'output e il numero di eccezioni soppresse; poi verifica.

Criterio. L'output è `lavoro`, `chiusura`, `principale: errore durante la chiusura`; non ci sono eccezioni soppresse.

Suggerimento. Chiediti quale eccezione esiste quando il corpo termina normalmente, prima di chiamare `close()`.

Soluzione commentata. Il corpo non lancia più errori. La chiamata automatica a `close()` fallisce e quell'errore diventa il principale; `getSuppressed()` restituisce un array vuoto.

C12-D01 – Eccezione del dominio e causa

Consegna. Esegui DemoVoto.java. Per ciascuno dei tre input indica se il metodo restituisce un voto o lancia `VotoNonValidoException`. Spiega perché solo uno degli errori mostra anche una causa.

Criterio. `27` produce `voto: 27`; `31` produce `errore: fuori intervallo: 31`; `xx` produce `errore: non è un numero: xx` e `causa: NumberFormatException`.

Suggerimento. Segui prima `Integer.parseInt`, poi il controllo dell'intervallo. Distingui l'eccezione creata direttamente dalla conversione di un errore ricevuto dalla libreria.

Soluzione commentata. `27` supera entrambi i controlli. `31` è un intero ma non un voto ammesso, quindi il metodo crea direttamente l'eccezione del dominio. `xx` non è convertibile; il `catch` riceve `NumberFormatException` e la conserva come causa nella nuova eccezione.

C12-D02 – Leggere una traccia

Consegna. Riprendi la variante con `Libro assente = null` dell'attività C07-D01 ed eseguila senza un `catch`. Nello stack trace individua tipo dell'eccezione, messaggio e prima riga del tuo sorgente. Descrivi quale riferimento manca.

Criterio. La diagnosi cita `NullPointerException` e la chiamata a `assente.descrizione()`, e distingue il punto di manifestazione dall'origine del valore `null`.

Suggerimento. Parti dalla prima riga della traccia, poi cerca il nome della tua classe nelle righe successive.

Soluzione commentata. Il tipo è `NullPointerException`: la variabile `assente` contiene `null`, quindi nessun oggetto può ricevere `descrizione()`. La riga della chiamata identifica dove l'errore emerge; l'assegnazione precedente spiega perché quel riferimento è assente.

C12-D03 – Seguire l'errore della pila

Consegna. Compila insieme `Pila.java` e `DemoPropagazione.java`. Senza eseguire, prevedi il messaggio e i quattro nomi di metodo stampati. Poi inserisci `pila.inserisci(9);` prima del `try`: quale percorso verrà seguito e perché il `catch` non stamperà nulla?

Criterio. Con pila vuota l'output è `pila vuota`, `estrai`, `leggi`, `mostra`, `main`. Con un elemento viene stampato `9`; l'estrazione riesce e nessuna eccezione raggiunge il gestore.

Suggerimenti. La figura 23 mostra il percorso dell'eccezione, non quello del valore restituito. Parti dal metodo che lancia e risali ai chiamanti.

Soluzione commentata. `estrai` rifiuta la pila vuota; `leggi` e `mostra` non hanno un gestore, quindi l'eccezione arriva a `main`. La traccia elenca prima il punto di lancio e poi i chiamanti. Dopo l'inserimento, `estrai` restituisce `9` e la catena termina normalmente: il ramo `catch` non viene eseguito.

Attività C13–C18

Queste prove accompagnano i capitoli C13–C18. Si eseguono con JDK 25 senza preview, su copie dei sorgenti: le varianti volutamente errate non devono restare nei programmi positivi. Ogni attività indica una risposta attesa, così il lettore può confrontare il ragionamento con il comportamento osservato.

C13-A01 – Due implementazioni, un riferimento

Consegna. In `DemoInterfacce.java`, assegna una `Temperatura` a un riferimento `Misurabile` e prova a chiamare prima `valore()`, poi `descrizione()`. Spiega quale chiamata compila e quale no, pur riferendosi allo stesso oggetto.

Criterio e soluzione. `valore()` compila perché è nel contratto `Misurabile`; `descrizione()` non è visibile dal tipo dichiarato del riferimento. L'oggetto resta una `Temperatura`, ma il compilatore usa il tipo del riferimento per stabilire le operazioni disponibili. Una nuova variabile `Descrivibile` può riferirsi allo stesso oggetto e chiamare `descrizione()`.

C13-P01 – Conflitto dei default

Consegna. Prima di compilare `DemoConflittoDefault.java`, prevedi la riga stampata. In una copia togli l'override di `nome()` da `Scelta` e compila.

Criterio e soluzione. La versione corretta stampa `primo+secondo`. Senza override la classe riceverebbe due metodi default incompatibili e il compilatore rifiuta la dichiarazione; l'ordine di `implements` non risolve il conflitto.

C13-D01 – Un esito nuovo

Consegna. Aggiungi `Sospeso(String motivo)` alle implementazioni consentite dall'interfaccia sealed di `DemoEsito.java`. Compila prima di cambiare lo `switch`, poi aggiungi il ramo mancante.

Criterio e soluzione. Dopo l'aggiunta del tipo, lo `switch` non copre più tutti gli esiti ammessi e non compila. Un ramo `case Sospeso sospeso -> ...` ristabilisce l'esaustività. Il lettore sa spiegare perché un `default` nasconderebbe la necessità di rivedere il significato del nuovo esito.

C14-A01 – Fino a quando esiste un'annotazione?

Consegna. Esegui `DemoAnnotazioni.java` con `Etichetta.java`. In una copia cambia la retention da `RUNTIME` a `CLASS`, ricompila e riesegui.

Criterio e soluzione. La versione iniziale stampa `catalogo` e `didattica`; la seconda non stampa etichette perché la normale reflection non le vede a runtime. La compilazione resta valida: la retention controlla la disponibilità successiva del metadato, non il significato delle due stringhe.

C14-G01 – Riconoscere un processor

Consegna. Compila i tre sorgenti di C14, poi invoca esplicitamente `EtichettaProcessor` con `-processor EtichettaProcessor -proc:only`. Indica se la nota del compilatore compare prima o dopo l'avvio di `main`.

Criterio e soluzione. La nota `etichetta presente su DemoAnnotazioni` compare durante `javac`; `main` non viene avviato da `-proc:only`. Un processor lavora sul modello del programma in compilazione, mentre `getAnnotationsByType` è una lettura a runtime.

C14-P01 – Un controllo dichiarato

Consegna. Progetta `@Responsabile` per un metodo: scegli `@Target`, `@Retention` e un elemento `String value()`. Descrivi quale componente leggerà l'annotazione, senza sostenere che Java invierà automaticamente una notifica al responsabile.

Criterio e soluzione. `ElementType.METHOD` limita la collocazione. `RUNTIME` è giustificata se un'applicazione la leggerà con reflection; `SOURCE` o `CLASS` possono bastare per un controllo di compilazione. Il valore è un metadato e richiede codice separato per produrre effetti.

C14-D01 – Ereditarietà e annotazioni ripetibili

Consegna. Prevedi le quattro righe di `DemoPolitiche.java`, poi le due etichette di `DemoPromemoria.java`. Spiega perché `getDeclaredAnnotationsByType` e `getAnnotationsByType` rispondono a domande diverse sulla sottoclasse e perché due promemoria non attivano da soli uno scheduler.

Criterio e soluzione. L'annotazione della base può essere visibile sulla derivata grazie a `@Inherited`, ma non diventa una dichiarazione scritta sulla derivata. Le due annotazioni ripetute restano due valori leggibili in ordine dichiarato; per usarli come orari di esecuzione occorre un componente applicativo separato.

C15-A01 – Perché il cast sparisce?

Consegna. In `DemoGenerics.java`, individua dove il compilatore conosce il tipo restituito da `titoli.estrai()`. Prova poi a passare `3` a `titoli.inserisci` in una copia del file.

Criterio e soluzione. `Pila<String>` fissa `T` a `String` nelle firme dei metodi usate dal chiamante. L'inserimento dell'intero non compila; non occorre attendere l'estrazione per scoprire l'incompatibilità. La spiegazione non attribuisce il controllo a un cast scritto a mano.

C15-P01 – Leggere e scrivere con wildcard

Consegna. Modifica `copiaNumeri` per leggere da `List<? extends Integer>` in una variabile `Integer` e ricevere `List<? super Integer>` come destinazione. Prova `List<Number>` e `List<Object>`; poi tenta di leggere direttamente un `Integer` dalla destinazione.

Criterio e soluzione. Entrambe le liste possono ricevere un `Integer`, ma il tipo garantito in lettura dal parametro con `super` è soltanto `Object`. La direzione della wildcard segue l'operazione che il metodo deve compiere.

C15-D01 – Un'assegnazione tentatrice

Consegna. Aggiungi `List<Number> numeri = List.of(1, 2);` e, su una riga separata, prova ad assegnare una variabile `List<Integer>` a `List<Number>`. Distingui i due casi.

Criterio e soluzione. Il primo può compilare perché il compilatore costruisce una lista nel contesto del tipo richiesto. Il secondo non compila: una `List<Integer>` già tipizzata non diventa una `List<Number>`, altrimenti si potrebbe inserirvi un `Double`.

C15-G01 – Il ponte dopo l'erasure

Consegna. Compila `DemoPonte.java` e ispeziona `DepositoTitoli` con `javap -p`. Individua il metodo con parametro `String` e quello con parametro `Object`. Spiega quale problema di dispatch risolve il secondo, senza sostenere che il codice sorgente abbia dichiarato due overload volontari.

Criterio e soluzione. Dopo l'erasure la firma ereditata usa `Object`; il compilatore aggiunge un metodo ponte che adatta la chiamata al metodo specializzato con `String`. La relazione generica resta verificata dal compilatore al punto d'uso.

C16-A01 – Leggere la composizione

Consegna. Prima di eseguire `DemoFunzioni.java`, segui `" Java "` attraverso `strip`, `toUpperCase`, il predicato e la stampa. Ripeti per `" Go "`.

Criterio e soluzione. Il primo testo diventa `JAVA` e produce `JAVA: 4`; il secondo diventa `GO` ma non supera la lunghezza minima di 3. La risposta distingue trasformazione, decisione ed effetto di stampa.

C16-P01 – Cattura di un valore locale

Consegna. Sposta il controllo della lunghezza in `Predicate<String> abbastanzaLungo = testo -> testo.length() >= lunghezzaMinima;`. Poi prova a incrementare `lunghezzaMinima` dopo la dichiarazione.

Criterio e soluzione. La lambda funziona finché la variabile locale è effectively final. Dopo l'incremento, il compilatore rifiuta la cattura. Togliere `final` dalla dichiarazione non cambia questa regola.

C16-D01 – Ricorsione con arresto

Consegna. Disegna le chiamate di `fattoriale(3)` e prevedi il risultato. Prova poi `fattoriale(-1)` in una copia e spiega perché il controllo iniziale è necessario.

Criterio e soluzione. Le chiamate arrivano a `fattoriale(0)`, che restituisce 1; risalendo otteniamo 1, 2 e 6. Il valore negativo lancia `IllegalArgumentException`, evitando una discesa senza un caso di arresto adatto.

C16-G01 – La stessa operazione in tre forme

Consegna. In `DemoPercorsoLambda.java`, segui i valori 10 e 3 nelle implementazioni anonima, lambda e method reference. Cambia `aggiungi(2)` in `aggiungi(5)` e prevedi l'ultima riga. Poi esegui `DemoCattura.java` e spiega perché le sue due righe sono diverse.

Criterio e soluzione. Le prime due implementazioni sottraggono e stampano 7, il riferimento a `somma` produce 13; `aggiungi(5)` produce 18. In `DemoCattura` il valore locale 3 resta stabile, mentre il campo `totale` cambia fra le chiamate, producendo 3 e 6. Il lettore distingue valore catturato e stato mutabile dell'oggetto.

C17-A01 – Duplicati e ordine

Consegna. Prevedi le prime tre righe di `DemoCollezioni.java`. Indica quale contratto conserva il secondo `Java` e quale lo elimina. Poi trasferisci la scelta a un nuovo caso: la biblioteca deve memorizzare due prestiti successivi dello stesso libro e, separatamente, i codici dei libri già mostrati in vetrina. Scegli i due tipi di collezione e spiega perché non puoi usarne uno solo senza perdere una proprietà richiesta.

Criterio e soluzione. Le righe sono `[Java, Reti, Java]`, `[Java, Reti]`, `[Reti, Java]`. `List` conserva duplicati e posizioni; `LinkedHashSet` conserva unicità e ordine d'incontro; `reversed()` offre la vista opposta. I due prestiti sono eventi distinti e restano in una `List<Prestito>`, anche se riguardano lo stesso libro. I codici già esposti formano invece un `Set<String>`; `LinkedHashSet` serve se conta la prima esposizione. La risposta deve associare ogni scelta al requisito, non solo nominare una classe.

C17-P01 – La pila della libreria

Consegna. In una copia del programma, aggiungi un terzo valore alla `Deque` e prevedi l'ordine di tre chiamate a `pop()`. Spiega in che cosa questo uso si collega alla pila costruita nei capitoli precedenti. Ora cambia il problema: le richieste devono uscire nell'ordine di arrivo. Quale contratto useresti e quali operazioni sostituiresti?

Criterio e soluzione. I valori escono nell'ordine inverso agli inserimenti con `push`. L'interfaccia è generica e l'implementazione `ArrayDeque` sostituisce per uso ordinario la pila didattica, senza cambiare il principio LIFO. Per la nuova richiesta FIFO si usa il contratto `Queue` o `Deque`, inserendo in coda con `offer` e prelevando dalla testa con `poll` o `remove` secondo la politica scelta per la coda vuota. La struttura concreta può restare `ArrayDeque`; cambia il significato delle operazioni.

C17-D01 – Il comparatore che perde un titolo

Consegna. Togli `thenComparing(Comparator.naturalOrder())` dal `TreeSet` del programma ed esegui. Spiega perché uno dei titoli di quattro lettere manca. Proponi poi un criterio che ordini prima per lunghezza e, a parità, conservi titoli diversi; indica anche un caso in cui sceglierei direttamente un `HashSet` e ordinare soltanto la presentazione sarebbe più semplice.

Criterio e soluzione. `Java` e `Reti` hanno la stessa lunghezza; un comparatore che guarda soltanto la lunghezza restituisce zero e il set li tratta come equivalenti. Un secondo confronto alfabetico distingue i due titoli. Se il requisito è solo evitare duplicati e l'ordine serve in una stampa occasionale, si può conservare un `HashSet` e ordinare una copia al momento della presentazione. Il criterio di ordinamento di `TreeSet` decide anche l'unicità in quella implementazione.

C18-A01 – Sorgente e risultato

Consegna. Prevedi le prime due righe di `DemoStream.java` e indica quale operazione crea la nuova lista.

Criterio e soluzione. La prima è `[1, 2, 4, 5, 6]`; la seconda è `[1, 2, 3, 4, 5, 6, 3]`. `filter` descrive la selezione; `toList` consuma lo stream e materializza un risultato senza modificare la sorgente.

C18-P01 – Finestre e aggregazioni

Consegna. Spiega perché `windowFixed(3)` produce tre gruppi e quale sia la somma stampata da `teeing`. Poi cambia la dimensione della finestra a 2 e prevedi i gruppi.

Criterio e soluzione. Sette elementi producono finestre `[1, 2, 3]`, `[4, 5, 6]`, `[3]`; la somma è 24. Con dimensione 2 otteniamo `[1, 2]`, `[3, 4]`, `[5, 6]`, `[3]`. I due collector di `teeing` leggono la stessa sequenza logica per contare e sommare.

C18-D01 – Un stream già consumato

Consegna. Salva `dati.stream()` in una variabile e invoca due volte `toList()` sulla stessa variabile. Prova poi a chiamare `dati.stream()` due volte in due espressioni separate.

Criterio e soluzione. Lo stesso oggetto `Stream` non può essere consumato due volte; la seconda terminale fallisce. La `List` rimane disponibile e può produrre due stream distinti. La risposta distingue il ciclo di vita della pipeline da quello della collezione.

C18-G01 – Quattro domande sul catalogo

Consegna. Prima di eseguire `DemoCatalogoStream.java`, prevedi i titoli disponibili ordinati, il numero di libri del reparto Informatica, la somma delle pagine e il titolo disponibile più breve per pagine. Aggiungi poi un libro non disponibile con 50 pagine: quali risposte cambiano?

Criterio e soluzione. I risultati iniziali sono `[Algebra, Algoritmi, Java]`, 3, 1150 e `Algebra`. Se il nuovo libro appartiene a Informatica, il conteggio del reparto sale a 4 e la somma delle pagine a 1200; la lista dei disponibili e il minimo disponibile restano invariati. Se scegli un altro reparto, devi adeguare la previsione del conteggio: il dato scelto fa parte del problema.

Attività C19–C24

Queste attività accompagnano i capitoli C19–C24. Si svolgono su copie dei sorgenti con JDK 25. Le prove C19–C21 e C24 usano API stabili; C23 richiede il modulo incubator dichiarato nel capitolo. I criteri aiutano a distinguere un risultato osservato da una garanzia dell'API.

C19-A01 – Un ripiego non richiesto

Consegna. Nel programma Optional, prevedi quante volte viene chiamato `nomeDiDefault()` quando la ricerca trova un libro e il chiamante usa prima `orElse(nomeDiDefault())`, poi `orElseGet(DemoOptional::nomeDiDefault)`. Esegui e confronta. Ripeti con una ricerca vuota.

Criterio e soluzione. Nel caso presente `orElse` valuta comunque l'argomento; `orElseGet` non chiama il fornitore. Nel caso vuoto entrambe le forme calcolano il ripiego una volta per chiamata. La differenza riguarda il momento di valutazione dell'argomento, non il valore finale se il fornitore restituisce sempre la stessa stringa.

C19-P01 – Quale assenza stiamo descrivendo?

Consegna. Aggiungi un libro senza autore e confronta `cerca(codice).map(DemoOptional::autore)` con `cerca(codice).flatMap(DemoOptional::autore)`. Scrivi il tipo di entrambe le espressioni e spiega cosa significa un risultato vuoto dopo `flatMap`.

Criterio e soluzione. `map` produce `Optional<Optional<String>>`; `flatMap` produce `Optional<String>`. Quest'ultimo può essere vuoto perché manca il libro o perché il libro non ha autore. Se l'applicazione deve spiegare quale dei due casi è avvenuto, l'appiattimento da solo non basta.

C20-A01 – `run` non avvia il thread

Consegna. Prima di eseguire AvvioEJoin, prevedi le righe con il nome del thread e gli stati prima di `start`, dopo `run` diretto e dopo `join`. Spiega perché `join` rende affidabile l'ultimo stato osservato.

Criterio e soluzione. Il lavoro chiamato direttamente stampa `main` e non cambia `NEW`. Il lavoro dopo `start` stampa `lavoratore`; dopo `join` l'esecuzione è conclusa e lo stato è `TERMINATED`. La spiegazione deve riferirsi al contratto di `start` e `join`, non alla velocità della macchina.

C20-P01 – Contare senza affidarsi alla fortuna

Consegna. Nel contatore protetto, individua il monitor usato dai due thread e spiega perché il risultato è `20000`. In una copia rimuovi `synchronized` soltanto da `incrementa`, esegui alcune volte e annota i risultati, senza assumere che ogni esecuzione mostri il difetto.

Criterio e soluzione. La versione corretta protegge gli incrementi con lo stesso monitor e `main` attende entrambi i lavoratori. La versione modificata contiene una

data race e non garantisce `20000`, anche se può stamparlo per caso. Il test non deve fallire perché una singola esecuzione difettosa appare corretta.

C20-D01 – La fine di due consumatori

Consegna. Nel produttore-consumatore, immagina di avviare due consumatori sulla stessa coda. Quante sentinelle di fine servono perché entrambi possano uscire dal ciclo? Spiega perché non puoi prevedere quale consumatore prenderà ciascun valore.

Criterio e soluzione. Con quel protocollo serve una sentinella per ogni consumatore. La coda assegna ciascun elemento a un solo `take`, ma la scelta del consumatore dipende dalla pianificazione. Un test corretto verifica che i valori siano consumati una volta e che i due lavoratori terminino, non l'ordine dei loro nomi nelle stampe.

C20-V01 – Quando scegliere un thread virtuale

Consegna. Un servizio riceve in media 2.000 richieste al secondo. Ciascuna dura 50 millisecondi, soprattutto perché aspetta un archivio remoto. Stima quante richieste sono contemporaneamente in corso e spiega quale limite dei thread di piattaforma motiva l'uso dei thread virtuali. L'archivio accetta però al massimo 20 accessi simultanei: proponi un limite esplicito. Ripeti infine la scelta per un lavoro che calcola senza attese su tutti i core disponibili.

Criterio e soluzione. In media sono in corso circa 100 richieste (`2.000 × 0,050`). Un thread di piattaforma per richiesta trattiene un thread del sistema operativo anche durante l'attesa; un thread virtuale può sospendersi e liberare il carrier, mantenendo il codice lineare. Il maggior numero di thread virtuali non aumenta la capacità dell'archivio: un `Semaphore` con 20 permessi può limitare gli accessi. Per un lavoro che usa continuamente la CPU, il limite è nei core e non nell'attesa; creare molti thread virtuali non accelera il singolo calcolo. La risposta deve distinguere concorrenza, throughput e latenza.

C21-A01 – Il `null` del bootstrap

Consegna. Esegui OsservaClassLoader. Individua quale classe restituisce `null` da `getClassLoader()` e spiega perché questo non indica una classe mancante. Evita di usare il suffisso dell'oggetto loader stampato come dato da memorizzare.

Criterio e soluzione. Nell'ambiente verificato `ArrayList` è definita dal bootstrap, rappresentato da `null` in questa API. I nomi interni e le identità stampate dei loader dipendono dall'implementazione e dall'esecuzione.

C21-P01 – Due errori, due richieste

Consegna. Spiega il diverso punto di partenza di `Class.forName("esempio.Mancante")` e di una classe `B` già compilata che contiene `new A()` ma viene eseguita dopo la rimozione di `A.class`. Distingui tipo e natura del fallimento.

Criterio e soluzione. La ricerca esplicita può lanciare `ClassNotFoundException`, eccezione controllata. L'uso di un riferimento necessario nel bytecode può

produrre `NoClassDefFoundError`, un `Error`; il messaggio può contenere una `ClassNotFoundException` come causa. La compilazione precedente di `B` non carica `A` nella JVM che eseguirà il programma.

C23-A01 – Il resto del vettore

Consegna. Compila SommaVector con `javac --release 25 --add-modules jdk.incubator.vector`, eseguilo con la stessa opzione di modulo e varia la lunghezza degli array. Spiega la funzione di `loopBound` e del ciclo scalare finale.

Criterio e soluzione. I gruppi completi vengono elaborati dal ciclo vettoriale; gli elementi rimanenti dal ciclo scalare. Il numero di corsie non è un output fisso da confrontare fra computer. L'avviso sul modulo incubator è atteso; la presenza dell'API non la rende stabile.

C23-P01 – La maschera dell'ultimo gruppo

Consegna. Esegui SommaVectorMascherata con cinque elementi. Se la specie preferita ha quattro corsie, disegna la maschera del secondo giro. Ripeti il ragionamento se la specie ne ha otto e confronta il risultato con il programma che usa una coda scalare.

Criterio e soluzione. Con quattro corsie il secondo giro ha una sola corsia valida; con otto corsie anche il primo giro ha cinque corsie valide e tre escluse. Il risultato numerico è uguale in entrambi gli algoritmi. Il numero di corsie è una proprietà della piattaforma osservata, non un valore garantito dal libro.

C24-A01 – Metadati disponibili a runtime

Consegna. Nel programma Reflection, cambia la retention dell'annotazione da `RUNTIME` a `CLASS` in una copia. Prima di eseguire, prevedi il risultato di `getAnnotation(Etichetta.class)` e la prima istruzione che non potrà proseguire.

Criterio e soluzione. La ricerca restituisce `null`; la successiva chiamata a `etichetta.value()` produce una `NullPointerException`. La soluzione robusta controlla `null` oppure conserva `RUNTIME` se il contratto richiede lettura a esecuzione.

C24-P01 – Pubblico e dichiarato

Consegna. Aggiungi un metodo privato al record `Libro`. Confronta le domande poste da `getMethod` e `getDeclaredMethod`, senza assumere che ottenere il descrittore dia automaticamente il diritto di invocare il metodo. Scrivi come cambierebbe il problema in un modulo nominato non aperto.

Criterio e soluzione. `getMethod` riguarda i metodi pubblici visibili; `getDeclaredMethod` trova un metodo dichiarato direttamente, anche se privato. L'accesso all'invocazione segue regole distinte e l'incapsulamento del modulo può impedire l'accesso profondo. La prova va fatta in un progetto di test, senza usare un'apertura indiscriminata dei pacchetti.

Attività C20A — Operazioni atomiche

Queste attività accompagnano la capitolo C20A. I programmi usano soltanto API stabili del JDK 25. Ogni risposta deve dichiarare l'invariante o la proprietà da garantire prima di scegliere una classe: un'uscita fortunata non sostituisce la spiegazione.

C20A-A01 — Due richieste, una copia

Problema. Nel programma delle prenotazioni, immagina che A e B leggano entrambi 1. Scrivi i risultati possibili dei due `compareAndSet(1, 0)` e il valore finale. Ripeti il ragionamento se B arriva soltanto dopo il CAS riuscito di A. Poi compila ed esegui il programma più volte. Non tentare di prevedere quale lettore otterrà la copia.

Criterio di accettazione. La risposta deve mostrare che al massimo uno dei due CAS può restituire `true` per quella transizione, che un fallimento lascia il valore invariato e che il chiamante fallito rilegge prima di decidere. L'output del programma completo è cinque prenotazioni accettate e zero copie disponibili.

Suggerimento. Cerca il punto esatto in cui lo stato passa da 1 a 0. La lettura precedente non è ancora una prenotazione.

Soluzione ragionata. Se entrambi osservano 1, il primo CAS che effettivamente sostituisce 1 con 0 restituisce `true`. L'altro trova 0 e restituisce `false`; al giro successivo legge 0 e il metodo risponde `false`. Se B arriva dopo il successo di A, legge direttamente 0 e non chiama CAS. In nessuno dei due ordini il numero diventa negativo. Il conteggio esterno delle risposte `true` serve a controllare l'esito della prova, non a decidere la prenotazione.

C20A-P01 — Un nome non basta a garantire un ordine

Problema. Modifica una copia di `PrenotazioniAtomiche` affinché `prenota()` restituisca un numero da 1 a 5 per le richieste accettate e `0` per quelle respinte. Il numero deve indicare la posizione ottenuta nell'insieme delle cinque copie: quando un CAS sostituisce `osservate` con `osservate - 1`, la posizione è `6 - osservate`. Raccogli i numeri ottenuti da venti lavoratori e verifica, dopo le `join`, che l'insieme sia esattamente `{1, 2, 3, 4, 5}`. Spiega perché questa posizione non è un identificatore persistente.

Criterio di accettazione. Nessun numero accettato si ripete; non compare alcun valore fuori dall'intervallo 1–5; quindici richieste ricevono 0. La risposta distingue questa numerazione in memoria da una chiave che debba restare unica dopo un riavvio o fra più processi.

Suggerimento. Restituisci la posizione soltanto nel ramo in cui il CAS riesce. Un numero calcolato prima può diventare obsoleto.

Soluzione ragionata. Nel ramo `if (compareAndSet(...))` si restituisce `6 - osservate`; quando `osservate` vale 5, 4, 3, 2, 1, i numeri prodotti sono rispettivamente 1, 2, 3, 4, 5. Un CAS fallito non assegna una posizione e riparte. La

raccolta concorrente può usare una coda thread safe o un array con indice ottenuto atomicamente, ma la verifica si fa dopo le `join`. Un nuovo processo ricomincerebbe da cinque copie iniziali: l'archivio persistente deve definire un'altra garanzia per l'identità durevole.

C20A-E01 — L'effetto che può ripetersi

Problema. Nel programma di aggiornamento, qualcuno propone questa lambda: `valore -> { inviaEmail(); return Math.min(10, valore + 2); }`. Spiega perché non è corretta anche se il valore finale rimane entro dieci. Descrivi una soluzione per cui la conferma venga inviata dopo la decisione, indicando quale fallimento rimane possibile fra aggiornamento e invio.

Criterio di accettazione. La risposta identifica il possibile ricalcolo della funzione, l'invio duplicato e il nuovo problema «stato cambiato, email non inviata». Non deve affermare che spostare semplicemente la chiamata dopo `updateAndGet` garantisca una transazione fra memoria e servizio di posta.

Suggerimento. Una funzione passata a un aggiornamento atomico calcola un candidato; non rappresenta necessariamente una sola esecuzione dell'intera richiesta.

Soluzione ragionata. Se un altro thread aggiorna il valore prima che il tentativo termini, la funzione può essere invocata di nuovo. L'email della prima invocazione non può essere ritirata dal CAS fallito. Inviare dopo un aggiornamento riuscito evita l'invio per ogni tentativo fallito, ma resta la finestra in cui il programma può fermarsi prima dell'invio. Per una conferma affidabile serve un protocollo applicativo che registri decisione e messaggio da inviare nello stesso sistema persistente, oppure un meccanismo equivalente con recupero e deduplicazione.

C20A-T01 — Dalla prenotazione alla statistica

Problema. Una pagina registra molte visualizzazioni. Il rapporto finale viene letto solo dopo che tutti i lavoratori sono terminati; un pannello può mostrare durante il lavoro una stima aggiornata. Confronta `AtomicLong` e `LongAdder`. Poi cambia il requisito: ogni visualizzazione deve ricevere un numero progressivo distinto nel momento in cui arriva. Spiega perché la scelta può cambiare. Esegui StatisticheAccessi e indica che cosa rende esatto il suo output finale.

Criterio di accettazione. La soluzione distingue throughput sotto contesa e precisione della lettura concorrente. Riconosce che `LongAdder.sum()` durante gli aggiornamenti non è uno snapshot atomico, mentre dopo le `join` dell'esempio il totale è 8.000. Per assegnare un numero distinto al chiamante propone un'operazione atomica che restituisce il valore precedente o nuovo, con gestione dell'intervallo e della persistenza se richiesta.

Suggerimento. Chiediti se il chiamante debba conoscere il numero assegnato a *questa* richiesta oppure solo un totale collettivo.

Soluzione ragionata. Per il solo rapporto finale `LongAdder` è una scelta da misurare se gli aggiornamenti concorrenti sono numerosi; `AtomicLong` resta corretto e può essere più semplice. Nel programma, ogni lavoratore termina prima del `sum()`, quindi nessuno aggiorna mentre si forma il risultato letto da `main`. Quando invece ciascun chiamante deve ricevere un numero progressivo distinto, `AtomicLong.getAndIncrement()` esprime direttamente l'assegnazione locale. Né

questa classe né `LongAdder` garantiscono da soli un numero unico fra più JVM o dopo il riavvio.

C20A-T02 — Qual è il confine dello stato?

Problema. Nel programma con stato immutabile la somma di disponibili e prenotati resta cinque. Disegna la sequenza di un lettore che prende un riferimento prima di un CAS riuscito e di un altro che lo prende dopo. Ora aggiungi un requisito: salvare su database il nome del lettore e la copia assegnata prima di confermare la richiesta. Quale parte può ancora essere gestita da `AtomicReference` e quale richiede un'altra garanzia?

Criterio di accettazione. La risposta indica le due fotografie valide `(5, 0)` e `(4, 1)` e spiega perché non compare `(4, 0)` se tutti leggono un solo `Stato` immutabile. Deve riconoscere che `AtomicReference` governa soltanto la fotografia nella JVM; la conferma persistente richiede una transazione o un protocollo equivalente nell'archivio.

Suggerimento. Un record è immutabile soltanto se anche le parti mutabili a cui rimanda sono gestite in modo coerente. Nel programma proposto i due componenti sono `int`, quindi non nascondono una lista modificabile.

Soluzione ragionata. Un lettore che chiama `fotografia()` prima della sostituzione riceve l'oggetto `(5, 0)`; uno che lo chiama dopo riceve `(4, 1)`. La variabile atomica pubblica un riferimento alla volta, e nessuno dei due record cambia. Aggiungere un nome a un database porta l'invariante oltre quella variabile: il CAS può riuscire e la scrittura fallire. La conferma deve seguire il contratto dell'archivio e trattare concorrenza, errori e ripetizioni a quel livello.

Attività C25–C28: scegliere, provare, trasferire

Le attività si svolgono su copie dei programmi. `A` chiede di prevedere e spiegare un comportamento, `C` di correggere o completare un caso, `G` di riconoscere il confine tecnico, `D` di decidere in una variante. Le soluzioni indicano il ragionamento osservabile; l'esecuzione positiva dei programmi non sostituisce le prove sui casi di errore.

C25-A01 – Byte o testo?

Consegna. Un file contiene la sequenza di byte di una fotografia e un altro contiene titoli in UTF-8. Scegli per ciascuno fra `InputStream` e `BufferedReader`, spiegando il ruolo del charset. Nel programma di importazione prevedi cosa accade se una riga contiene `libro;errore`.

Soluzione ragionata. La fotografia rimane una sequenza di byte; tentare di interpretarla come testo può cambiarla o far fallire la decodifica. I titoli richiedono la conversione UTF-8 dichiarata. La riga con separatore produce `IOException` prima della pubblicazione finale. Per dimostrare il requisito, il test deve partire da una destinazione già esistente e verificare che il suo contenuto non cambi.

C25-C01 – Una prova del fallimento

Consegna. Modifica una copia del programma affinché il messaggio indichi il numero *fisico* della riga, comprese quelle vuote. Prepara un input con due righe vuote prima del dato errato. Verifica anche che non rimangano file provvisori se la validazione fallisce.

Soluzione ragionata. Serve un contatore incrementato a ogni `readLine`, distinto dal conteggio dei titoli importati. Il `finally` rimuove il provvisorio sul fallimento; il test deve controllare la directory, non soltanto l'eccezione. Se anche la pulizia fallisce, occorre conservare la causa principale e segnalare il problema di pulizia separatamente.

C25-D01 – Milioni di righe e duplicati

Consegna. L'archivio cresce oltre la memoria disponibile e non può contenere titoli duplicati. Disegna una soluzione che mantenga la pubblicazione completa o nulla. Specifica come scopri i duplicati e qual è il limite della tua scelta. Ripeti la decisione se la sorgente arriva da una risorsa dentro un JAR.

Soluzione ragionata. La lettura e la scrittura restano progressive. Una `Set` in memoria funziona solo se gli elementi distinti ci stanno; altrimenti serve una strategia esterna, per esempio partizionamento o ordinamento su disco. Il file nel JAR è una sorgente impacchettata, non una destinazione aggiornabile con `Files.move`; il risultato va pubblicato in un percorso esterno. La soluzione deve nominare il limite del provider per lo spostamento atomico.

C26-A01 – Due volte le 02:30

Consegna. Nel programma della fattura spiega perché `2026-10-25T02:30` a Roma produce due offset. Indica quali due istanti sono rappresentati e perché `LocalDateTime` da solo non basta per una prenotazione irrevocabile.

Soluzione ragionata. Durante il ritorno all'ora standard l'ora civile viene ripetuta. Gli offset `+02:00` e `+01:00` portano a istanti separati da sessanta minuti. Una prenotazione deve conservare l'offset scelto o l'istante, oltre alla zona necessaria per la presentazione e le regole future.

C26-C01 – Arrotondare nel posto giusto

Consegna. Calcola l'imposta del 22% su tre righe da `0.03` euro. Confronta arrotondamento `HALF_UP` a due decimali per ciascuna riga e arrotondamento una sola volta dopo la somma delle basi. Non usare `double` come sorgente dei valori.

Soluzione ragionata. Ogni riga produce `0.0066`, quindi `0.01` dopo arrotondamento; la somma delle imposte arrotondate è `0.03`. La base complessiva è `0.09`, la sua imposta è `0.0198`, arrotondata a `0.02`. La differenza dimostra che il punto dell'arrotondamento è una regola di dominio, non un dettaglio di sintassi.

C26-D01 – Trenta giorni o 720 ore?

Consegna. Una biblioteca concede un prestito «per trenta giorni civili»; un servizio concede una finestra di «720 ore effettive». Scegli tipi e operazioni per entrambe le regole. Spiega che cosa cambierebbe attraversando un cambio d'ora legale e come rendi ripetibile un test.

Soluzione ragionata. La prima regola usa `LocalDate.plusDays(30)` nel contesto civile della biblioteca. La seconda usa un `Instant` e `Duration.ofHours(720)`. Le due scadenze possono mostrare ore civili diverse perché le regole di zona cambiano l'offset. Un `Clock.fixed` elimina la dipendenza dall'ora corrente nei test.

C27-A01 – Assenza o guasto?

Consegna. Nel client locale prevedi cosa il chiamante ottiene se il server risponde `404` con un corpo di testo, e cosa accade se nessun processo ascolta sulla porta. Quale dei due casi può diventare `Optional.empty()` per una ricerca di libro?

Soluzione ragionata. Un `404` è una risposta HTTP con status e corpo; il client la riceve e il contratto del servizio può definirla come assenza normale. Una connessione rifiutata è un fallimento di comunicazione e non dimostra che il libro manchi. La distinzione va mantenuta anche se l'interfaccia utente mostra una frase semplice.

C27-C01 – Un retry controllato

Consegna. Cambia il server locale affinché la prima `GET /libri` restituisca `503` e la seconda `200`. Implementa al massimo un retry per questa sola richiesta e registra il numero di tentativi. Verifica separatamente che un `POST` che crea una prenotazione non venga ripetuto senza un contratto di idempotenza.

Soluzione ragionata. Il test deve osservare due chiamate nel caso temporaneo e una nel caso positivo immediato. Dopo due fallimenti la logica restituisce un errore esplicito; non entra in un ciclo indefinito. Per il `POST` il timeout lascia incerto se il server abbia già creato la prenotazione: serve una regola applicativa, per esempio una chiave di idempotenza supportata dal server.

C27-D01 – Dieci richieste, tre posti

Consegna. Dieci richieste HTTP partono da thread virtuali, ma il servizio remoto consente tre lavori simultanei. Descrivi dove applicare il limite e come controllarlo. Confronta la decisione con `sendAsync` se il processo deve comporre i risultati senza occupare un thread per ogni attesa.

Soluzione ragionata. Un `Semaphore(3)` attorno all'accesso al servizio limita le richieste in corso, indipendentemente dal numero di thread virtuali. Il test misura il massimo di richieste attive sul server locale, non soltanto il tempo totale. `sendAsync` restituisce future componibili; anche quel modello richiede un limite di capacità. La scelta riguarda la struttura del programma, non una promessa di maggiore velocità per la singola richiesta.

C28-A01 – Limite e durata

Consegna. Nel programma FFM alloca due interi e prova a scriverne tre, in una copia isolata. In un'altra copia conserva il segmento dopo la chiusura dell'arena e prova a leggerlo. Classifica i due errori.

Soluzione ragionata. Il primo accesso supera il limite spaziale del segmento; il secondo usa una regione la cui durata è terminata. La prova deve anticipare dove fallirà il programma. Non va proposta la `global` arena solo per evitare di ragionare sulla chiusura.

C28-G01 – Che cosa dice `strlen`?

Consegna. Sostituisci `"Java"` con `"città"` nel programma. Prevedi se `strlen` conta cinque caratteri o il numero dei byte della stringa C in UTF-8. Spiega perché il layout del risultato `size_t` va verificato sulla piattaforma.

Soluzione ragionata. `strlen` conta i byte prima dello zero finale; à richiede più di un byte in UTF-8, quindi il risultato supera cinque. `size_t` segue l'ABI della piattaforma, non il tipo Java `long` per definizione. Se cambiano architettura o libreria, un descrittore copiato senza verifica può essere errato anche se il codice Java compila.

C28-D01 – Un puntatore trattenuto dalla libreria

Consegna. Una libreria C conserva il puntatore ricevuto e lo usa in un callback successivo. Progetta durata dell'arena, condivisione fra thread e punto di chiusura. Specifica quale informazione manca se conosci soltanto il nome della funzione C.

Soluzione ragionata. L'arena deve restare viva almeno fino all'ultimo uso nativo del puntatore e dello stub di callback. Se thread diversi accedono alla regione, una confined arena non è sufficiente; serve una scelta di condivisione e sincronizzazione coerente. Senza firma C, layout, ABI e contratto di proprietà del

puntatore non è possibile costruire un binding corretto. La risposta valida sa fermarsi prima di inventare queste garanzie.

Prova integrata – Una ricevuta che arriva da fuori

Situazione. Un programma della biblioteca interroga un servizio HTTP locale per ottenere la ricevuta di un prestito. Il servizio restituisce un corpo testuale con codice del libro, data di scadenza, importo della cauzione e valuta. Il programma deve salvare una copia leggibile dal personale. Alcune richieste possono fallire; il file precedente non deve diventare una ricevuta parziale. Questa prova riunisce i confini dei capitoli 25, 26 e 27. Il capitolo 28 resta un approfondimento: la presenza di una libreria C nel sistema non è, da sola, un motivo per introdurre FFM.

Consegna. Disegna prima il contratto dei dati ricevuti: quale charset, quale formato per la data, quale unità monetaria, quali campi obbligatori e quale dimensione massima? Indica quali errori avvengono prima di avere una risposta HTTP, quali sono status HTTP da interpretare e quali sono errori di validazione del corpo. Scegli il tipo Java per la scadenza e per l'importo, poi descrivi come pubblichi il file finale. Scrivi almeno quattro prove: risposta valida, `404` secondo il contratto del servizio, corpo malformato e connessione interrotta dopo l'invio della richiesta. Per ogni prova specifica lo stato atteso del file precedente.

Soluzione ragionata. La scadenza civile è un `LocalDate` se il contratto dice «valido fino a quel giorno» e non fissa un istante preciso. L'importo viene interpretato da testo decimale con `BigDecimal`, conservando separatamente una `Currency` o un codice di valuta validato. Il formato di scambio dichiara UTF-8, una data ISO e un limite alla dimensione del corpo. Un parser controlla presenza e grammatica dei campi prima di trattare il dato come ricevuta. Non basta uno status `200`: il corpo può essere incompleto o non valido.

Il client imposta timeout coerenti con il servizio e distingue fallimento di trasporto, status e validazione. `404` può significare «ricevuta assente» soltanto se il contratto remoto lo dice; una connessione interrotta non equivale ad assenza. Il contenuto valido viene scritto in un file provvisorio nella directory della destinazione e chiuso prima del tentativo di pubblicazione. Se lo spostamento atomico richiesto non è disponibile, l'operazione fallisce lasciando il file precedente; il prodotto può scegliere un'altra politica, ma deve dichiararla e provarla separatamente. La prova positiva controlla contenuto e metadati essenziali; i tre casi negativi controllano che il file precedente sia identico byte per byte e che il provvisorio sia rimosso.

Variante di trasferimento. Il servizio comincia a fornire ricevute da molti megabyte. Una soluzione che usa `BodyHandlers.ofString` e poi costruisce un'altra stringa completa può consumare troppa memoria. Passiamo a una gestione progressiva del corpo e imponiamo un limite di byte mentre leggiamo; la validazione e la pubblicazione restano due passi distinti. Se il formato obbliga a vedere il documento intero prima di validarlo, si può conservare il provvisorio e rileggerlo, evitando comunque di esporlo come finale. La scelta non cambia la semantica degli errori HTTP.

Criterio di valutazione. La risposta collega ogni rischio al suo confine: trasporto, status, formato, tempo, numero o file system. Motiva la scelta delle API e prova che l'archivio precedente resta protetto. FFM non serve senza una necessità nativa.

Estensione avanzata: il calcolo esiste soltanto in una libreria C. In una seconda versione la biblioteca eredita una libreria nativa che calcola una firma di controllo del file. La funzione riceve puntatore ai byte e lunghezza, restituisce un codice numerico e non conserva il puntatore dopo il ritorno. Prima di scrivere un binding FFM, il lettore deve ottenere l'header C, conoscere il tipo esatto della lunghezza nell'ABI usata e sapere che cosa indicano i codici di ritorno. Il file è stato già ricevuto e validato; l'integrazione nativa interviene soltanto per calcolare la firma, prima della pubblicazione. Se la funzione restituisce un errore, il file precedente resta intatto.

Una soluzione motivata alloca un segmento abbastanza grande da contenere i byte da elaborare, vi copia il blocco secondo l'API scelta e lo mantiene valido per tutta la chiamata. Poiché la libreria non conserva il puntatore, un'arena chiudibile attorno alla chiamata può essere sufficiente; se il contratto cambiasse e il puntatore venisse trattenuto, la durata andrebbe riprogettata. Il `FunctionDescriptor` riflette la firma della libreria su quella piattaforma, non una supposizione ricavata da un esempio trovato altrove. Il codice di ritorno viene tradotto in un risultato di dominio che distingue firma calcolata e fallimento nativo. Il valore della firma, il nome della libreria e la sua versione entrano nelle evidenze di verifica, perché un aggiornamento del binario può cambiare il risultato anche se il sorgente Java è identico.

Le prove dell'estensione eseguono almeno tre casi: un file noto con firma attesa, un file vuoto e una chiamata che restituisce un codice di errore. La prova del file vuoto chiarisce se la funzione accetta un puntatore a zero byte o richiede comunque un indirizzo valido; la risposta non può essere inventata dal programma Java, deve provenire dal contratto C. Il caso d'errore conferma che il provvisorio non viene pubblicato. Queste prove si eseguono nella piattaforma nativa dichiarata, idealmente in un processo separato dalle altre verifiche, perché un binding errato può terminare la JVM. La scelta di FFM è giustificata solo dal nuovo vincolo della libreria C: il percorso precedente con API Java rimane la soluzione più semplice finché quel vincolo non esiste.

Attività guidate per C29 e C30

Queste attività usano il catalogo del capitolo 29 e i programmi del capitolo 30. Ogni modifica parte da un difetto osservabile: prima si formula una previsione, poi si cambia una cosa e si controlla il risultato. I comandi che producono file scrivono in `target/` o in una cartella `build/` locale. Per C29 servono JDK 25 e Maven con accesso alle dipendenze dichiarate nel POM; per C30 basta il JDK 25.

C29.1 – Il JAR esiste, ma non parte

Un collega riceve `catalogo-1.0.0.jar`. `jar --list --file target/catalogo-1.0.0.jar` mostra le classi, mentre `java -jar` risponde che manca l'attributo principale del manifest. Prima di modificare il codice Java, individua il confine che ha fallito. Ricostruisci una versione del JAR senza la configurazione `mainClass`, osserva il problema, ripristina il POM ed esegui `mvn clean verify` e il JAR con `dati/titoli.txt`.

Criterio di riuscita. Il lettore sa indicare il file `META-INF/MANIFEST.MF`, mostra la voce `Main-Class: esempio.catalogo.CatalogoCli` nel JAR corretto e ottiene due titoli in output. Sa spiegare perché la presenza di `CatalogoCli.class` non basta a `java -jar` per scegliere il punto d'ingresso.

Indizio. Confronta `jar --describe-module --file ...` solo se stai studiando i moduli; per questo esercizio apri il manifest con `unzip -p target/catalogo-1.0.0.jar META-INF/MANIFEST.MF` oppure con un programma per archivi. L'applicazione usa il class path.

Soluzione ragionata. `java -jar` legge il manifest. Il plugin JAR del POM scrive `Main-Class` perché la configurazione `<mainClass>` lo richiede. Senza quella voce il JAR può contenere classi compilate ed essere comunque privo di un ingresso scelto per questa modalità di avvio. La riparazione è nella build; cambiare `main` non risolve il difetto dell'artefatto.

C29.2 – Un test verde non copre il limite

Il catalogo accetta un titolo di ottanta unità `char` e rifiuta quello di ottantuno. Aggiungi un test per il primo caso e verifica il secondo. Prima di eseguirlo, scrivi quale risultato attendi per `"x".repeat(80)` e `"x".repeat(81)`. Cambia temporaneamente la condizione di `Titoli.normalizza` da `> 80` a `>= 80`: il test nuovo deve diventare rosso. Ripristina il codice e ripeti `mvn clean verify`.

Criterio di riuscita. Il rapporto Surefire mostra il nuovo test; la mutazione `>= 80` viene scoperta; il codice ripristinato torna verde. Il lettore sa dire che una percentuale di copertura, anche alta, non avrebbe espresso da sola questo confine.

Indizio. Usa `assertEquals(80, Titoli.normalizza("x".repeat(80)).length())` e conserva `assertThrows` per l'input da 81. Non modificare il test per adattarlo al difetto introdotto.

Soluzione ragionata. La regola è inclusiva sul massimo: `length() > 80` rifiuta soltanto lunghezze superiori. Il test a 80 verifica il lato ammesso, quello a 81 il lato rifiutato. Una mutazione che sposta il confine deve far fallire almeno il primo. Questo è più informativo di un test che esegue semplicemente il metodo.

C29.3 – Chi usa JUnit quando parte l'applicazione?

Esamina `pom.xml`, poi esegui `mvn dependency:tree` se il plugin è disponibile nel tuo ambiente. Individua JUnit e le eventuali sue dipendenze transitive. Produci il JAR, controlla che non contenga classi `org/junit`, ed esegui l'applicazione. Formula la conseguenza di sostituire `<scope>test</scope>` con il comportamento predefinito, senza sostenere che Maven copierà automaticamente tutte le librerie dentro il JAR ordinario.

Criterio di riuscita. Il lettore distingue dipendenza di test, dipendenza di runtime e contenuto fisico del JAR. Riconosce che il plugin JAR usato qui non crea un *fat JAR*.

Indizio. `jar --list --file target/catalogo-1.0.0.jar` mostra il contenuto effettivo. `dependency:tree` mostra invece il grafo risolto dalla build: sono osservazioni diverse.

Soluzione ragionata. JUnit è necessario per compilare ed eseguire i test. Lo `scope` di test lo esclude dal normale class path dell'applicazione. Cambiare lo scope può allargare il grafo delle dipendenze dell'applicazione, ma non trasforma da solo il JAR ordinario in un archivio che incorpora i JAR esterni. Il programma funziona perché le tre classi di produzione usano soltanto API del JDK.

C29.4 – Dal JAR a un altro ambiente

Esegui `jdeps --print-module-deps target/catalogo-1.0.0.jar`; crea una runtime image con `jlink --add-modules java.base --output target/runtime-catalogo` e avvia il JAR con `target/runtime-catalogo/bin/java -jar ...`. Poi aggiungi, su un ramo o in una copia di prova, una classe che usa direttamente `jdk.jfr.Recording`. Ripeti l'analisi e spiega perché un runtime costruito solo con `java.base` non soddisfa più il nuovo programma. Ripristina il progetto del capitolo.

Criterio di riuscita. Il lettore mostra l'output di `jdeps` prima e dopo la modifica, collega `jdk.jfr` alla nuova API e non scambia la dimensione del runtime per una prova funzionale. Il trasferimento può essere spiegato anche senza produrre il secondo pacchetto.

Indizio. `jdeps` analizza dipendenze statiche. Una riflessione o un caricamento dinamico può richiedere controlli ulteriori durante l'esecuzione.

Soluzione ragionata. L'applicazione iniziale usa API di `java.base`; quindi quel modulo basta al runtime provato. La classe aggiunta importa `jdk.jfr.Recording`, e il grafo statico deve includere `jdk.jfr`. L'analisi suggerisce i moduli per `jlink`, ma l'avvio e i casi d'uso restano prove necessarie, specialmente quando il programma carica componenti in modo dinamico.

C30.1 – Il limite è un comportamento, non un commento

Compila `ImportazioneControllata.java` e `ProvaImportazioneControllata.java` con `javac --release 25 -Xlint:all -d build ...` nella cartella C30. Aggiungi due prove: una riga di esattamente 80 unità `char` deve essere accettata; un archivio con esattamente 100 righe non vuote deve essere accettato. Mantieni i casi da 81 e 101 già presenti, che devono essere rifiutati. Spiega che cosa succede se il carattere finale è `\r\n`.

Criterio di riuscita. Quattro lati dei due confini hanno un risultato atteso esplicito. Il lettore sa localizzare il controllo prima dell'accumulo della riga e prima dell'aggiunta del titolo numero 101.

Indizio. `"x".repeat(80)` e `"x\n".repeat(100)` costruiscono input senza file esterni. Il parser ignora `\r` e usa `\n` come fine riga; questa è una scelta del formato didattico, da documentare se i dati hanno altre convenzioni.

Soluzione ragionata. Quando `riga.length()` vale 80, il carattere successivo diverso da fine riga causa `IOException`; una riga di 80 entra, quella di 81 no. Quando la lista contiene 100 titoli, la successiva chiamata ad `aggiungi` fallisce. La lettura di `\r\n` non aggiunge `\r` al titolo. Il test verifica il contratto concreto, non una vaga promessa di «input controllato».

Variante. Prova a inviare molte righe vuote. Il limite di cento titoli non interrompe questa lettura: decidi un massimo complessivo di caratteri o righe, aggiungi un errore distinto e una prova che lo attraversi. Il nuovo controllo va fatto durante la lettura, non dopo averla completata.

C30.2 – Un percorso fornito dall'utente

Immagina di esporre il catalogo come servizio: il client invia il nome del file da importare. Progetta una directory autorizzata e scrivi due casi di prova, uno per un file interno e uno per un percorso che tenta di uscire con `..`. Aggiungi un terzo caso con un collegamento simbolico che punta fuori dalla directory. Descrivi dove va collocato il controllo e quale rischio resta se il filesystem cambia tra controllo e apertura.

Criterio di riuscita. La soluzione non si limita a cercare la sottostringa `..`; considera normalizzazione, link simbolici, autorizzazione e apertura del file. Distingue inoltre il limite di righe del parser dal permesso di accedere a quel file.

Indizio. Parti dalla directory radice decisa dal server, non dalla directory corrente del processo. Le API `Path`, `Files` e gli attributi del filesystem aiutano a ragionare sul percorso effettivo; in alcuni sistemi serve una strategia di apertura che riduca la finestra tra controllo e uso.

Soluzione ragionata. Il client non deve scegliere liberamente qualunque percorso del processo. La soluzione definisce una radice, risolve e verifica il candidato rispetto a essa e tratta i link simbolici secondo una politica esplicita. Il controllo del nome e quello dell'oggetto effettivamente aperto non sono identici: un attaccante con accesso concorrente al filesystem può cambiare l'oggetto dopo una verifica separata. Il progetto del servizio deve scegliere API e permessi adeguati al sistema operativo, poi provare il percorso consentito e quelli rifiutati.

C30.3 – Log e registrazione raccontano fatti diversi

Esegui `DiagnosiImportazione` e stampa gli eventi con `jfr print --events esempio.catalogo.Importazione build/catalogo.jfr`. Confronta il messaggio `Importati 2 titoli` del logger con un evento JFR. Per ciascuno annota la domanda a cui risponde e una domanda a cui non risponde. Proponi un identificatore di richiesta che colleghi un errore di importazione ai log del servizio senza scrivere i titoli ricevuti.

Criterio di riuscita. Sono visibili tre eventi, ciascuno con `titoli = 2`; le durate vengono trattate come osservazioni locali. Il lettore evita di mettere input, token o dati personali nei campi diagnostici per comodità.

Indizio. Il log dice che una chiamata è terminata con un conteggio. L'evento JFR, se abilitato nel periodo giusto, conserva anche un intervallo temporale. Nessuno dei due prova che l'input di una richiesta diversa sarebbe accettato.

Soluzione ragionata. Il log è un messaggio applicativo su un risultato. L'evento JFR collega il tipo di operazione, il conteggio, il thread e una durata osservata. Un identificatore opaco generato al confine della richiesta può apparire nei log di ingresso, errore e uscita; deve avere una politica di conservazione e non deve rivelare il contenuto del file. Per investigare lentezze intermittenti servono misure nel periodo del problema, non tre numeri dell'esempio.

C30.4 – Trasferire il modello a HTTP e database

Ora il catalogo scarica titoli da una risposta HTTP e li salva in un database. Scrivi un contratto operativo per il caso in cui la risposta si interrompa a metà e per quello in cui il database fallisca dopo 40 inserimenti. Indica dove limiti dimensione e tempo della risposta, quando convalidi i record, come eviti uno stato parziale e che cosa osservi durante un nuovo tentativo. Non occorre scegliere un framework: la prova riguarda le decisioni e i confini.

Criterio di riuscita. La soluzione collega un problema a ogni misura: limiti di rete contro risposte senza fine; validazione contro dati malformati; transazione o strategia equivalente contro importazioni parziali; idempotenza contro duplicati nei retry; log e metriche per conoscere l'esito senza esporre dati. Il lettore indica almeno un test che provoca ciascun guasto.

Indizio. La lista del parser può essere valida e il salvataggio fallire comunque. Chiediti quando il nuovo catalogo debba diventare visibile agli utenti.

Soluzione ragionata. Si delimita la risposta con timeout e massimo di byte o record. Si verifica il formato mentre si legge e si rifiuta l'intera importazione se la politica richiede un insieme completo. Il salvataggio avviene in una transazione, oppure in una nuova versione sostituita solo al termine: così il fallimento dopo 40 righe non espone un catalogo incompleto. Un identificatore dell'importazione o una chiave naturale permette di decidere l'effetto di un retry. Test con risposta troncata, record errato e fallimento del database dimostrano il comportamento; i log mostrano fase, esito e identificatore senza copiare i titoli.

Attività conclusive: scegliere il prossimo mattone

Le due attività non chiedono di ripetere un elenco di API. Chiedono di mostrare come si passa da un problema a una scelta verificabile e come si continua a imparare quando il libro non può più accompagnare ogni passo. È utile conservare le risposte e rileggerle dopo aver lavorato a un progetto reale: il criterio non è indovinare la risposta «giusta» del libro, ma rendere espliciti contesto, prova e limite.

C22.1 – Dalla prima diagnosi a un problema nuovo

Riprendi l'autodiagnosi fatta all'inizio del volume, oppure descrivi ora un problema che allora avresti risolto copiando un esempio. Può essere un archivio che cresce, una richiesta HTTP lenta, una prenotazione concorrente o un programma difficile da distribuire. Scrivi in prosa quale comportamento osservabile vuoi ottenere e quale guasto temi. Indica un mattone del libro che aiuta, uno che in questo caso sarebbe fuori posto e un esperimento minimo che potrebbe smentire la tua scelta. Infine cambia un vincolo, per esempio la sorgente da file a rete o l'uso da un solo thread a molti thread, e spiega che cosa devi riprogettare.

Criterio di riuscita. La risposta distingue il problema dalla prima API venuta in mente; descrive almeno un'invariante o limite misurabile, una prova positiva e una negativa, e riconosce una condizione in cui la soluzione iniziale smette di bastare. Il lettore non riceve punti per il numero di tecnologie nominate.

Indizio. Nel catalogo C29 la build risponde alla domanda «un collega può ricostruire e avviare il programma?». In C30 il parser risponde a un'altra domanda: «quali input e costi accettiamo?». Una build verde non risponde alla seconda.

Soluzione ragionata, un esempio. Voglio importare titoli da un file caricato da un utente e renderli visibili soltanto quando sono tutti validi. Il limite di ottanta unità per riga e quello di cento titoli riducono due costi; un file con milioni di righe vuote mostra che manca un limite complessivo. Provo cento titoli validi, il centunesimo e molte righe vuote; poi verifico che un fallimento non sostituisca il catalogo precedente. `AtomicInteger` non risolve la pubblicazione completa di più record: serve una transazione o una sostituzione di versione. Se la sorgente diventa HTTP, aggiungo timeout, massimo della risposta e gestione dell'interruzione della rete. La scelta è trasferibile perché ogni nuova misura risponde a un rischio dichiarato.

C22.2 – Leggere una novità del JDK senza confondere progetto e release

Scegli una funzionalità menzionata nelle Conclusioni, per esempio i thread virtuali, la Foreign Function and Memory API o una proposta di Valhalla. Parti dalla pagina del progetto o da una notizia, poi raggiungi il JEP e la documentazione ufficiale della release che stai usando. Registra data della consultazione, stato nella tua release, eventuali opzioni di compilazione ed esecuzione e problema che la funzionalità intende affrontare. Scrivi un esperimento di poche righe o una prova di compilazione che distingua «disponibile» da «annunciata». Concludi dicendo se la useresti nel tuo progetto e quale informazione ti manca ancora.

Criterio di riuscita. La risposta cita fonti primarie con la release esatta; distingue stabile, preview e incubator; separa una motivazione del progetto dal contratto dell'API disponibile. La decisione d'uso contiene un caso concreto e un limite, non soltanto entusiasmo o diffidenza.

Indizio. Una pagina di progetto può contenere lavori per versioni future. Un JEP può essere draft, candidate, targeted o delivered. Per il comportamento in Java 25 controlla la documentazione Java 25 e compila con quel JDK.

Soluzione ragionata, un esempio. Il JEP 444 registra i thread virtuali come consegnati in Java 21. Nel JDK 25 `Thread.ofVirtual()` è disponibile senza `--enable-preview`. Il problema affrontato è il costo di mantenere un thread di piattaforma dedicato a molte operazioni che attendono I/O; non è una promessa di accelerare un calcolo CPU-bound. Un programma minimo crea un thread virtuale, lo avvia e ne attende la fine; una prova di carico rappresentativa serve prima di sostituire l'architettura di un servizio. Per una proposta di Valhalla la risposta sarebbe diversa: verifico l'eventuale build sperimentale, ma non tratto la proposta come API stabile di Java 25.

Conclusioni. Continuare a costruire, mattone dopo mattone

Per concludere

Se siete arrivati fino a qui, *Java Mattone dopo Mattone* è servito almeno a tenervi compagnia per un bel tratto di strada. Mi auguro anche che vi abbia lasciato qualcosa di più duraturo della sintassi di un costrutto o del nome di una classe del JDK. Le API cambiano; il modo di affrontare un problema, di spiegare una scelta e di lasciare il lavoro comprensibile a chi viene dopo resta.

Vorrei che di tutte le cose incontrate nel libro rimanessero l'amore per il buon codice e il desiderio di migliorare. Scrivere bene non è soltanto una soddisfazione personale. È un modo di rispettare i colleghi, l'azienda e i clienti: rende più facile correggere un difetto, adattare un requisito, capire se una modifica ha fatto danni. Un programma che funziona oggi e che domani nessuno osa toccare ha risolto soltanto una parte del problema.

In queste pagine abbiamo incontrato oggetti, tipi, collezioni, funzioni, thread, file, rete, strumenti di build e diagnosi. Non vorrei che ne usciste con una lista di nomi da ricordare. Ogni mattone ha avuto un motivo: riconoscere una difficoltà, vedere perché la soluzione precedente non bastava, costruire la successiva e verificarla. Se domani vi troverete davanti a una biblioteca, a un servizio HTTP o a un programma che si blocca, la domanda utile non sarà «quale capitolo devo copiare?». Sarà «qual è il confine del problema, quale proprietà devo conservare e quale prova mi dirà che ho scelto bene?».

> **Il metodo che portiamo fuori dal libro.** Davanti a un compito nuovo, descriviamo prima il risultato osservabile e il caso che può fallire. Scegliamo poi la soluzione più piccola capace di rispettare quel contratto. La proviamo, ne scopriamo il limite e la adattiamo al contesto reale. Questo percorso non garantisce che non sbaglieremo; rende gli errori visibili e correggibili.

Dare un nome alla cosa giusta

Quando apriamo un file scritto da altri, le prime domande sono semplici: che cosa rappresenta questa classe? Che cosa promette il metodo? Da dove arriva questo dato? Se i nomi sono opachi, dobbiamo eseguire il programma nella testa prima ancora di capire quale problema affronta. `var1` e `var2` fanno risparmiare qualche battuta a chi scrive e ne chiedono molte di più a chi legge. Un nome come `quantitaDisponibile` non elimina la necessità di verificare la regola, ma ci dice quale domanda porre. Allo stesso modo, `tang()` può essere sufficiente dentro un contesto matematico strettamente dichiarato; `calcolaAngoloTangente()` racconta di più quando quel contesto non è evidente. Il nome migliore dipende dalla responsabilità e dal pubblico del codice, non dalla sua lunghezza.

Abbiamo visto questa scelta nel catalogo degli ultimi capitoli. `MAX_RIGA` e `MAX_TITOLI` dichiarano limiti diversi. Se li chiamassimo entrambi `LIMITE`, un lettore dovrebbe risalire ogni volta all'uso per capire quale risorsa stiamo controllando. Se

invece un metodo si chiama `leggi` ma salva su database o invia richieste di rete, il nome mente. O cambiamo il nome per rappresentare l'effetto, o separiamo le responsabilità. Un buon nome è una piccola promessa verificabile.

Non trasformiamo però la regola in una gara al nome più lungo. `variabileCheFaQuestoEQuello` resta vago, anche se occupa mezza riga. Un nome utile indica il concetto del dominio o l'intenzione dell'operazione. Se non riusciamo a trovarlo, forse non abbiamo ancora capito abbastanza bene il compito. Fermarsi a discutere un nome può far emergere due significati che avevamo confuso nel codice.

È tutto un leggi e scrivi, ma bisogna sapere che cosa

Ricordo Gaetano Barbagallo, il mio primo capo progetto. Quando un compito mi sembrava enorme, mi riportava a una domanda molto concreta: che cosa leggiamo, che cosa trasformiamo e che cosa scriviamo? Negli anni quella semplicità mi è rimasta accanto. Non significa che ogni sistema sia banale. Significa che possiamo cercare i passaggi osservabili invece di lasciare tutto in un metodo di mille righe sperando che il debugger ci salvi.

Prendiamo ancora l'importazione del catalogo. Leggiamo caratteri da una sorgente; li separiamo in righe; normalizziamo i titoli; verifichiamo due limiti; infine restituiamo un risultato. Quando la sorgente diventa una risposta HTTP, aggiungiamo timeout e dimensione massima. Quando la destinazione diventa un database, decidiamo se il nuovo archivio debba apparire tutto insieme. Il percorso «leggi e scrivi» non è uno slogan che cancella i rischi: è una traccia per scoprire *dove* nascono e a chi spetta gestirli.

Una classe dovrebbe rappresentare una responsabilità che sappiamo raccontare. Un metodo dovrebbe offrire un'operazione con un contratto comprensibile. Se una parte del problema è troppo grande, dividerla può renderla verificabile. Ma dividere non vuol dire creare decine di classi minuscole senza una ragione. La composizione, le interfacce e talvolta l'ereditarietà sono strumenti, non premi per un diagramma complicato. Le usiamo quando rendono chiaro un confine o un comportamento da sostituire; altrimenti scegliamo la forma più diretta.

> **Dal problema alla struttura.** Se una modifica a una regola dei titoli costringe a cambiare il client HTTP e il codice che disegna una schermata, abbiamo accoppiato responsabilità che cambiano per motivi diversi. Se invece dividiamo un algoritmo in dieci metodi che si chiamano l'un l'altro senza offrire un concetto riconoscibile, abbiamo soltanto spostato la complessità. Il criterio è chiedere quale decisione appartiene a ogni parte e quale prova la difende.

Lasciare nel sorgente ciò che serve

Una pratica che ho incontrato spesso è conservare la vecchia versione di un metodo come blocco commentato accanto a quella nuova. All'inizio sembra prudenza. Dopo qualche modifica, nessuno sa più se quel blocco rappresenti un'alternativa valida, un esperimento fallito o un pezzo dimenticato. Il lettore deve attraversare due programmi sovrapposti per capire quale dei due esegua la JVM. Il versionamento conserva la storia in modo più affidabile: possiamo ritrovare una versione, leggere perché fu cambiata e, se serve, recuperarla con una decisione esplicita. Nel sorgente corrente lasciamo il codice che serve al comportamento corrente.

Questo non vuol dire eliminare ogni commento. Un commento utile spiega ciò che il codice da solo non può mostrare bene: perché abbiamo imposto un limite di cento titoli, quale formato esterno dobbiamo rispettare, quale anomalia di una libreria abbiamo aggirato e quando potremo rimuovere quell'aggiramento. Un commento come `// ciclo su tutto l'array` accanto a `for (int i = 0; i < array.length; i++)` ripete l'ovvio. Peggio ancora, può diventare falso se il codice cambia e il commento resta. Se un'operazione è difficile da spiegare, proviamo prima a dare nomi migliori e a separare i passaggi; poi documentiamo il motivo che non si vede dalla sintassi.

Anche la Javadoc ha una responsabilità precisa. Nel capitolo sulla build abbiamo documentato ingressi, risultati ed errori del catalogo. Una pagina HTML generata con successo non prova che il contratto sia corretto: se il metodo ora chiude il `Reader` passato e la documentazione tace, abbiamo nascosto un effetto importante. La documentazione utile si legge insieme ai test e cambia insieme al comportamento.

Convenzioni e leggibilità sono un patto con gli altri

Le convenzioni di un progetto aiutano a prevedere dove trovare una classe, come si chiamerà un metodo e dove sono i test. Java ha forme riconoscibili per package, classi, metodi e costanti. Possiamo avere ragioni per discostarci da una convenzione, ma una deviazione casuale costringe ogni collega a ricominciare da capo. Restare coerenti nel progetto riduce il lavoro di interpretazione. È una forma di collaborazione, non una questione di gusto personale imposta dall'alto.

La leggibilità non coincide con il minor numero di righe. Comprimere dieci passaggi in una sola espressione può essere divertente; la domanda è se il prossimo lettore saprà individuare l'errore quando il risultato sarà sbagliato. D'altra parte, spezzare una regola semplice in un labirinto di helper può peggiorarla. Nel capitolo sulle Stream abbiamo visto che una pipeline funziona bene quando ogni passaggio ha un significato e l'ordine resta visibile. Se un ciclo esplicito rende più chiara la chiusura di una risorsa o il punto in cui si ferma la lettura, quel ciclo è una scelta migliore per quel problema.

Pensiamo anche a chi userà il codice senza aver letto questo libro. Un test dal nome preciso, un messaggio d'errore che distingue input malformato da file inaccessibile, un log che non espone titoli riservati e un JAR che parte con le istruzioni dichiarate sono tutti atti di comunicazione. La qualità del sorgente non finisce alla parentesi graffa: comprende il modo in cui il programma si presenta a chi deve eseguirlo, diagnosticarlo e modificarlo.

Scegliere lo strumento dopo aver riconosciuto il problema

Capita di sentirsi dire: «Usiamo questo framework, così facciamo prima». A volte sarà davvero la scelta giusta. Ma prima dobbiamo sapere che cosa deve fare il programma, quali vincoli ha e che cosa introduciamo adottando quello strumento. Se il requisito è leggere un file di due righe, aggiungere una piattaforma intera può complicare distribuzione e manutenzione. Se invece il programma deve gestire molti utenti, persistenza, sicurezza e osservabilità, fingere che basti la stessa classe didattica sarebbe altrettanto ingenuo.

Lo abbiamo visto con gli strumenti della concorrenza. Un `AtomicInteger` aiuta quando più thread aggiornano in modo atomico *una* variabile. Non trasforma automaticamente una prenotazione che modifica disponibilità, ordine e pagamento in una transazione indivisibile. Un thread virtuale permette di mantenere uno stile diretto per molte attività che attendono I/O, senza riservare un thread di piattaforma per tutta l'attesa; non accelera per magia un calcolo che satura la CPU. In entrambi i casi il nome dell'API arriva dopo aver descritto l'invariante o il collo di bottiglia. La soluzione giusta è quella che risponde alla domanda concreta e supera una prova pertinente.

Lo stesso vale per la Vector API, ancora incubator nel percorso Java 25 di questo volume, e per l'accesso alla memoria nativa: sono argomenti affascinanti, ma chiedono condizioni specifiche, costi e rischi. La scelta di studiarli è utile; la scelta di usarli in produzione richiede un problema che li giustifichi. Un benchmark senza carico rappresentativo o una build verde senza test sui confini non sono prove sufficienti.

Imparare dagli altri, poi rifattorizzare con una prova

Per crescere conviene leggere codice scritto da persone più esperte, ma anche codice diverso dal nostro. Guardiamo come un progetto gestisce gli errori, dove colloca i test, quali decisioni spiega nei commenti e quali lascia alla struttura. Chiediamo perché una soluzione è stata scelta, invece di copiarne soltanto la forma. Il codice aperto può insegnare molto; anche una revisione fra colleghi, fatta con rispetto, rende visibili ipotesi che da soli non avremmo notato. Chi riceve una domanda spesso è felice di spiegare una scelta: è una delle parti migliori del nostro mestiere.

Rifattorizzare significa migliorare la struttura mantenendo il comportamento osservabile concordato. Prima ci serve una prova che quel comportamento sia riconoscibile. Poi possiamo cambiare un passo alla volta: rinominare una variabile, estrarre una regola, sostituire una duplicazione reale, ricompilare e rieseguire le verifiche. Se introduciamo un limite nuovo, cambiamo invece il contratto: lo chiamiamo per quello che è e aggiungiamo test sui casi ammessi e rifiutati. Questa distinzione evita che una modifica funzionale si nasconda dietro l'etichetta rassicurante di «pulizia».

Il suggerimento che avevo scritto anni fa, prima farlo funzionare, poi ottimizzarlo, infine renderlo più bello, va letto con una precisazione: *funzionare* significa rispettare il requisito e avere una prova adeguata, non mostrare una schermata verde una volta sola. Misuriamo prima di ottimizzare; rileggiamo prima di rendere elegante; ripetiamo i test dopo. Quando un collega apre il cambiamento, deve poter

capire quale problema abbiamo risolto e perché la nuova forma è più semplice da mantenere.

Continuare a seguire Java senza inseguire ogni novità

Amber, Loom, Panama e Valhalla rappresentano percorsi diversi di evoluzione della piattaforma. Questo volume usa Java 25 come riferimento. Alcune idee sono ormai parte stabile della piattaforma: per esempio i thread virtuali, descritti nel capitolo 20, sono stati finalizzati in Java 21; l'API Foreign Function and Memory, approfondita nel capitolo 28, è stata finalizzata in Java 22. Amber ha accompagnato funzionalità che oggi usiamo normalmente, come l'inferenza locale con `var`, i text block, i record e le gerarchie `sealed`. Valhalla continua a esplorare i value object; in Java 25 non trattiamo le value class come una funzionalità stabile da usare nei programmi del libro.

> **Come leggere una novità.** Una pagina di progetto racconta una direzione; un JEP descrive obiettivi, compromessi e stato di una proposta; la specifica e la documentazione della release dicono che cosa è disponibile nel JDK che stiamo usando. Prima di adottare una novità controlliamo release, stato stabile/preview/incubator, opzioni necessarie, compatibilità e casi d'uso. Poi scriviamo un piccolo programma e lo proviamo nel nostro ambiente. «Annunciato» e «disponibile in Java 25» non sono sinonimi.

Non serve prevedere oggi quale sarà la prossima API indispensabile. Serve conservare la capacità di imparare. Quando una nuova release arriverà, partite dal problema che avete, leggete le note ufficiali e chiedetevi quale parte del vostro modello cambia davvero. Una sintassi più breve può rendere espressivo un caso; non rende automaticamente più chiara una decisione di dominio. Un nuovo strumento di diagnosi può mostrare più dati; non decide da solo quale comportamento è corretto. Il lavoro del programmatore rimane collegare quei dati alle domande giuste.

Spero che questo libro vi abbia dato il coraggio di fare quelle domande, di provare una soluzione, di scoprire dove fallisce e di costruire il mattone successivo. Non smettete di cercare di capire il codice che scrivete e quello che vi viene affidato. Il buon codice non è un punto d'arrivo: è una pratica condivisa, fatta di problemi riconosciuti bene, scelte spiegabili e verifiche che consentono agli altri di continuare.

Per verificare il percorso

Le attività conclusive chiedono di partire da un requisito, riconoscere i confini che può attraversare e giustificare una soluzione con prove. Scegli un programma del libro che hai modificato: racconta il problema iniziale, l'invariante che hai protetto, una prova positiva e un caso che potrebbe smentire la soluzione. Poi immagina che un collega debba estenderlo a una nuova fonte di dati. Quale parte del modello può riusare e quale contratto deve verificare di nuovo? Se riesci a rispondere senza limitarti al nome di una API, hai acquisito il metodo che tiene insieme i capitoli.

Riferimenti per seguire il linguaggio

Lo stato tecnico richiamato in queste Conclusioni è riferito a Java 25, verificato il 2 ottobre 2026. Il JEP 444 sui thread virtuali li registra come funzionalità consegnata in Java 21; la documentazione Oracle della Foreign Function and Memory API documenta la finalizzazione in Java 22 e rimanda al JEP 454. La proposta istitutiva di Project Amber ne chiarisce l'obiettivo; le specifiche Java 25 dei capitoli precedenti documentano le singole funzionalità. Il sito ufficiale di Project Valhalla distingue gli esperimenti successivi da ciò che è incluso in Java 25. Per gli aggiornamenti futuri occorre ricontrollare queste fonti e le note della release usata, non riutilizzare questa fotografia come se fosse senza data.