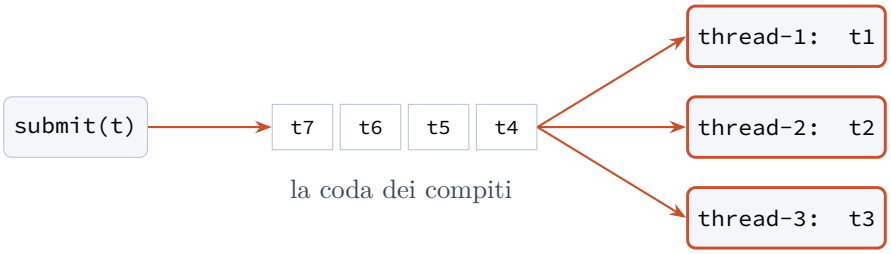
PAJC · Programmazione Avanzata in Java e C · 2026/27

Dispensa 4

Thread e concorrenza

Dal processo alla JVM, executor, stato condiviso, la coda degli eventi di Swing e i thread virtuali



Tanti compiti, pochi thread: l'immagine che torna in tutta la dispensa.

prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

massimiliano.redolfi@unibs.it

Versione 0.2

Quest'opera è distribuita con licenza Creative Commons "Attribuzione – Non commerciale – Condividi allo stesso modo 4.0 Internazionale".

Indice

1 Introduzione: una finestra che congela **1**
- 1.1 Il problema . . . 1
- 1.2 L'immagine guida: una coda e chi la svuota . . . 3
- 1.3 Che cosa serve . . . 4
- 1.4 La strada che percorreremo . . . 4
- Esercizi . . . 5
- Risposte . . . 5

2 Processi, thread e la JVM **7**
- 2.1 Il processo . . . 7
- 2.2 Il thread . . . 7
- 2.3 Scheduler e context switch . . . 8
- 2.4 Come fanno i sistemi moderni . . . 9
- 2.5 L'astrazione di Java . . . 10
- Esercizi . . . 13
- Risposte . . . 14

3 Il primo thread **16**
- 3.1 Compito e lavoratore . . . 16
- 3.2 Un compito con dei dati . . . 17
- 3.3 Aspettare: `sleep` e `join` . . . 18
- 3.4 Fermare un thread: l'interruzione . . . 19
- 3.5 Lambda e stato . . . 20
- 3.6 L'esempio guida: i primi su due thread . . . 20
- Esercizi . . . 22
- Risposte . . . 24

4 Executor, Callable e Future **26**
- 4.1 Perché non un thread per ogni compito . . . 26
- 4.2 `ExecutorService` e le factory di `Executors` . . . 26
- 4.3 Spegnere un executor . . . 28
- 4.4 Dentro un pool . . . 29
- 4.5 Valori di ritorno: `Callable` e `Future` . . . 29
- 4.6 L'esempio guida: i primi su tutti i core . . . 30
- Esercizi . . . 32
- Risposte . . . 34

5 Quando i thread condividono **35**
- 5.1 Il contatore che perde . . . 35
- 5.2 Visibilità: non è solo questione di atomicità . . . 36
- 5.3 `synchronized` e il monitor . . . 38
- 5.4 Coordinarsi: `wait` e `notifyAll` . . . 39
- 5.5 Deadlock . . . 40

5.6 Thread safety per costruzione . . . 41
5.7 L'esempio guida: sommare in un contatore condiviso . . . 42
Esercizi . . . 43
Risposte . . . 45

6 Gli strumenti di `java.util.concurrent` 47
6.1 Lock espliciti: `ReentrantLock` . . . 47
6.2 Lettori e scrittori: `ReadWriteLock` . . . 48
6.3 Risorse a posti limitati: `Semaphore` . . . 49
6.4 Variabili atomiche . . . 49
6.5 Collezioni concorrenti . . . 50
6.6 Quale strumento . . . 52
6.7 L'esempio guida: `LongAdder` e una mappa dei primi . . . 52
Esercizi . . . 53
Risposte . . . 55

7 Il tempo e l'attesa 56
7.1 Compiti differiti e periodici . . . 56
7.2 Stream paralleli, visti da sotto . . . 57
7.3 Thread virtuali . . . 58
7.4 L'esempio guida in tre modi, su calcolo e su attesa . . . 59
Esercizi . . . 59
Risposte . . . 61

8 Thread e Swing: la coda degli eventi 62
8.1 Un'interfaccia è un ciclo . . . 62
8.2 Come funziona `EventQueue` . . . 63
8.3 Chi avvia che cosa . . . 63
8.4 Il blocco . . . 64
8.5 Lavoro fuori, rientro dentro . . . 64
8.6 `SwingWorker` . . . 65
8.7 `javax.swing.Timer` . . . 67
Esercizi . . . 68
Risposte . . . 70

9 Palestra 71
Problemi . . . 71

10 Dove andare adesso 76
10.1 `CompletableFuture`: comporre invece di aspettare . . . 76
10.2 `ForkJoinPool` e `RecursiveTask`: dividi e conquista . . . 76
10.3 Il modello di memoria e `VarHandle` . . . 77
10.4 Structured concurrency e scoped values . . . 77
10.5 Nelle altre dispense . . . 78
10.6 Per approfondire . . . 78

Capitolo 1

Introduzione: una finestra che congela

1.1 Il problema

Il programma da cui parte tutta la dispensa è piccolo e, a prima vista, innocuo. Una finestra con un campo di testo, un pulsante e un'etichetta: si scrive un numero, si preme *Conta*, e l'etichetta dice quanti numeri primi ci sono fino a quel numero. Il test di primalità è quello che tutti scrivono la prima volta, per divisioni successive:

```
public class PrimeTester {
    public static boolean isPrime(long n) {
        if (n < 2) return false;
        if (n % 2 == 0) return n == 2;
        for (long d = 3; d * d <= n; d += 2)
            if (n % d == 0) return false;
        return true;
    }

    public static long count(long from, long to) {
        long n = 0;
        for (long i = from; i <= to; i++)
            if (isPrime(i)) n++;
        return n;
    }
}
```

La finestra è altrettanto diretta. Il metodo `conta()` è il *listener* del pulsante: legge il campo, chiama `count`, scrive il risultato.

```
public class PrimeFinderApp extends JFrame {
    private final JTextField campoMax = new JTextField("100000000", 10);
    private final JButton btnConta = new JButton("Conta");
    private final JLabel lblRisultato = new JLabel("-", SwingConstants.CENTER);

    public PrimeFinderApp() {
        super("Numeri primi");
        JPanel sopra = new JPanel();
        sopra.add(new JLabel("Fino a:"));
        sopra.add(campoMax);
        sopra.add(btnConta);
        add(sopra, BorderLayout.NORTH);
        add(lblRisultato, BorderLayout.SOUTH);
        btnConta.addActionListener(e -> conta());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
    }

    private void conta() {
        long max = Long.parseLong(campoMax.getText());
        lblRisultato.setText("conto...");
        long n = PrimeTester.count(1, max);
        lblRisultato.setText("primi: " + n);
    }

    public static void main(String[] args) {
        new PrimeFinderApp().setVisible(true);
    }
}
```

Con `max` uguale a cento milioni succedono tre cose, e nessuna è quella che l'autore si aspettava. La prima: l'etichetta non mostra mai "conto…". La seconda: per quattordici secondi la finestra è morta. Non si ridisegna se la si copre e la si scopre, non risponde al mouse, non si chiude; dopo qualche secondo il sistema operativo la segnala come "non risponde". Poi, tutto insieme, compare il risultato: `primi: 5761455`. La terza cosa si vede solo aprendo il monitor di sistema: per tutto il tempo *un* core lavora al cento per cento e gli altri quindici (su un portatile recente) stanno fermi.

Da questo programma nascono le due domande della dispensa, che sono le due facce della stessa idea.

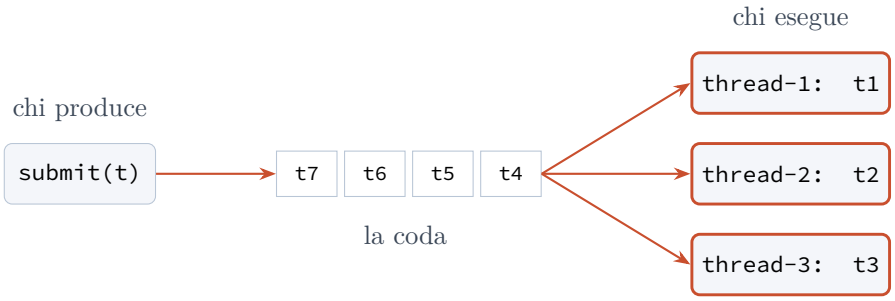
- **Come si usa tutta la macchina?** Il conteggio dei primi si divide in pezzi indipendenti: chi conta da 1 a 50 milioni non ha bisogno di chi conta da 50 a 100. Sedici core potrebbero farlo in un sedicesimo del tempo. Serve un modo per dire "questi due lavori, insieme".
- **Come si tiene viva la finestra?** Quattordici secondi, o anche uno, sono troppi per un'interfaccia che deve ridisegnarsi e rispondere. Serve un modo per dire "questo lavoro, altrove, e avvisami quando hai finito".

La risposta a entrambe è il *thread*: un secondo flusso di esecuzione dentro lo stesso programma. Con più thread il conteggio si divide fra i core; con un thread in più la finestra resta libera mentre il conteggio procede. Il capitolo 9 riprende `PrimeFinderApp` e la porta dove deve arrivare: il risultato in poco più di un secondo, una barra di avanzamento che si muove,

un pulsante per annullare. In mezzo c'è tutto ciò che rende l'idea facile da dire e delicata da realizzare, perché due flussi che lavorano insieme devono anche, prima o poi, *parlarsi*; e quando lo fanno male il programma non si rompe: dà risultati sbagliati una volta su mille.

1.2 L'immagine guida: una coda e chi la svuota

Un'immagine tornerà in ogni capitolo, e vale la pena fissarla subito. Da una parte dei *compiti*: unità di lavoro, piccole o grandi, che qualcuno produce. Dall'altra dei *lavoratori*: i thread, pochi, che vivono a lungo. In mezzo una *coda*: i compiti vi entrano nell'ordine in cui arrivano, e ogni lavoratore, quando è libero, prende il primo e lo esegue.



È l'*executor* del capitolo 4, con la sua coda di compiti e il suo gruppo di thread. È la coda bloccante del capitolo 6, con un produttore e un consumatore. È, con un solo lavoratore, la coda degli eventi di Swing del capitolo 8: ogni click, ogni ridisegno, ogni richiesta del programma è un compito in fila, e c'è un unico thread che li svuota uno alla volta. Ed è, rovesciata, il modo in cui Java 21 esegue i thread virtuali del capitolo 7: moltissimi thread in coda, pochi thread veri che li portano avanti. Chi tiene a mente questa figura ha già capito perché la finestra congela: il suo unico lavoratore è occupato da un compito di quattordici secondi, e dietro di lui la coda si allunga.

Due parole che il linguaggio comune confonde, e che qui vanno tenute distinte. *Concorrenza* è avere più flussi in corso nello stesso intervallo di tempo, anche su un solo core che li alterna: riguarda la *struttura* del programma, ed è ciò che rende la finestra reattiva. *Parallelismo* è eseguirli davvero nello stesso istante su core diversi: riguarda la *velocità*, ed è ciò che divide per dieci il tempo del conteggio. Un programma concorrente è corretto o sbagliato indipendentemente dal numero di core; il parallelismo lo ottiene solo chi ha già risolto la concorrenza.

Osservazione 1.1

Il salto non è nel codice, che resta quasi tutto uguale, ma nel modo di ragionare. Con un solo flusso ogni istruzione segue la precedente e il programma è una storia che si legge dall'alto in basso. Con due flussi fra due istruzioni qualsiasi di uno può passare un tempo arbitrario in cui l'altro fa quello che vuole: il programma è un insieme di storie intrecciate, e l'intreccio cambia a ogni esecuzione. Tutto ciò che segue serve a scrivere programmi corretti per *ogni* intreccio, non per quello che si è visto nella prova di ieri.

1.3 Che cosa serve

Poco, ma preciso. Le lambda e le interfacce funzionali (*Dal nome alla funzione*): un compito, qui, è quasi sempre una lambda, e la regola sulle variabili catturate spiega da sola metà del capitolo 3. Interfacce e classi (*Le interfacce in Java moderno*): `Runnable`, `Callable`, `Future` sono interfacce, e le si usa come tali. Gli stream (*Stream, per intero*) solo nel capitolo 7, dove si chiude un discorso lasciato aperto lì: che cosa fa davvero `parallel()`. Per il capitolo 8 basta saper costruire una finestra con un pulsante e un listener, come la `PrimeFinderApp` di questo capitolo; tutto ciò che riguarda i thread di Swing viene spiegato da zero. Il capitolo 2 contiene un programma C di dieci righe: va letto, non scritto, e chi non conosce il C può fidarsi del commento accanto a ogni riga.

Restano fuori, di proposito, la programmazione asincrona con `CompletableFuture`, il `ForkJoinPool` scritto a mano, il modello di memoria di Java nella sua forma completa e la rete: ne parla, con un assaggio ciascuno, il capitolo 10.

1.4 La strada che percorreremo

Il capitolo 2 scende sotto Java: che cosa sono un processo e un thread per il sistema operativo, che cosa ha di suo un thread e che cosa condivide, come lo scheduler li alterna sui core, e come `java.lang.Thread` si appoggia a tutto questo. È il capitolo che spiega *perché* valgono le regole dei capitoli seguenti.

Il capitolo 3 scrive il primo thread: compito e lavoratore, `Runnable` e `Thread`, `start` e non `run`, aspettare con `join`, fermare con l'interruzione. In chiusura il conteggio dei primi su due thread scritti a mano: quattordici secondi diventano sette.

Il capitolo 4 toglie di mezzo i thread scritti a mano: l'executor, la coda, i compiti che restituiscono un valore (`Callable` e `Future`), e il conteggio dei primi distribuito su tutti i core, con le misure, i limiti e il perché dei limiti.

Il capitolo 5 è quello delle cose che vanno storte: due thread che toccano lo stesso dato, il contatore che perde incrementi, `synchronized` e il monitor, `wait` e `notifyAll`, il deadlock. È il capitolo più importante, e l'unico in cui gli errori non danno nessun segnale.

Il capitolo 6 apre la cassetta degli attrezzi di `java.util.concurrent`: lock espliciti, semafori, variabili atomiche, collezioni concorrenti, code bloccanti; e una tabella per scegliere.

Il capitolo 7 aggiunge il tempo e l'attesa: compiti differiti e periodici, gli stream paralleli visti da sotto, e i thread virtuali di Java 21, che cambiano la risposta alla domanda "quanti thread posso avere?".

Il capitolo 8 torna alla finestra: la coda degli eventi, l'unico thread che la svuota, la regola che ne discende, e i tre modi di fare un lavoro lungo senza congelare niente: un thread con il rientro, `SwingWorker`, `javax.swing.Timer`.

Il capitolo 9 è la palestra: nove programmi completi, in difficoltà crescente, dall'esercizio da compito in classe alla `PrimeFinderApp` finita. Il capitolo 10 indica dove continuare.

Esercizi

Esercizio 1.1: Che cosa fa il tempo (⋆)

Nella `PrimeFinderApp` di questo capitolo `count(1, 100_000_000)` dura quattordici secondi. Senza eseguire nulla, stimare quanto dura con un milione e con diecimila, sapendo che `isPrime(n)` prova i divisori fino a $\sqrt{n}$. Con quale dei tre valori la finestra "congela"? Dove sta il confine?

Esercizio 1.2: Un core su sedici (⋆)

Il monitor di sistema mostra un core al cento per cento e quindici fermi. Il programma è lento perché la macchina è lenta, o perché il programma usa una parte della macchina? Che cosa cambierebbe eseguendolo su un computer con due core? E con uno solo?

Esercizio 1.3: Concorrenza o parallelismo (⋆)

Per ciascuna situazione dire se ciò che serve è concorrenza, parallelismo, o entrambi: (a) un editor che salva il file in automatico mentre si scrive; (b) la compressione di un video di due ore; (c) un server che serve mille connessioni che per lo più aspettano; (d) la `PrimeFinderApp` del capitolo 9, con la barra di avanzamento.

Risposte

Risposta 1.1. Il costo di `isPrime(n)` cresce come $\sqrt{n}$ e i numeri da provare sono n, quindi il conteggio fino a n costa circa $n\sqrt{n}$. Da 10^8 a 10^6 il rapporto è $100 \cdot 10 = 1000$: circa 14 millisecondi. Fino a diecimila, un altro fattore mille: microsecondi. Con un milione la finestra non congela in modo visibile, ma il listener dura comunque millisecondi in cui non succede nient'altro; con cento milioni congela per quattordici secondi. Il confine non è un numero: è la regola del capitolo 8, un listener non deve mai durare più di qualche millisecondo, perché durante la sua esecuzione la coda degli eventi è ferma. Un lavoro il cui tempo dipende da ciò che l'utente scrive non può stare in un listener.

Risposta 1.2. Il programma usa un sedicesimo della macchina: un solo flusso di esecuzione occupa un solo core, qualunque sia il numero di core disponibili. Su due core durerebbe gli stessi quattordici secondi (e un core resterebbe fermo); su uno solo, gli stessi quattordici secondi più il tempo che il sistema dedica agli altri programmi. La macchina non è lenta: il programma è scritto per una macchina che non esiste più da vent'anni. Il capitolo 4 divide il lavoro fra i core e misura che cosa si ottiene davvero: non sedici volte meno, per ragioni che valgono la pena di capire.

Risposta 1.3. (a) Concorrenza: due attività in corso insieme, e la seconda non deve fermare la prima; un solo core basta, perché il salvataggio passa quasi tutto il tempo ad aspettare il disco. (b) Parallelismo: un unico lavoro di puro calcolo che si vuole più veloce, diviso fra i core; senza più core non si guadagna niente. (c) Concorrenza: mille flussi che aspettano quasi sempre, e il problema è tenerli tutti in corso senza pagarli, non farli calcolare più in fretta (è il caso dei thread virtuali del capitolo 7). (d) Entrambi: parallelismo per il conteggio, concorrenza fra il conteggio e l'interfaccia, che deve muovere la barra e ascoltare "Annulla".

Capitolo 2

Processi, thread e la JVM

Il capitolo 1 ha promesso un secondo flusso di esecuzione. Prima di scrivere `new Thread(...)` conviene sapere che cosa si sta chiedendo, perché il thread di Java è un'astrazione sottile sopra un oggetto che esiste già nel kernel del sistema operativo, con i suoi costi e le sue regole. Alla fine del capitolo sapremo che cosa un thread ha di suo e che cosa condivide con gli altri, perché lo scheduler può sospenderlo fra due istruzioni qualsiasi, in quali stati può trovarsi e come guardare dentro una JVM in esecuzione. Ognuna di queste cose tornerà, come spiegazione, nei capitoli seguenti: la coda dei compiti del capitolo 4 esiste perché creare un thread costa, e il contatore che perde del capitolo 5 perde perché lo scheduler non chiede il permesso.

2.1 Il processo

Quando si lancia un programma, il sistema operativo crea un *processo*: un programma in esecuzione, con tutto ciò che gli serve per esistere. Gli assegna uno *spazio di indirizzamento* proprio, cioè un insieme di indirizzi di memoria che contengono il codice, i dati, l'*heap* (gli oggetti creati con `new`) e lo *stack* (le variabili locali e le chiamate in corso); gli assegna una tabella dei file aperti, un identificatore, dei diritti. Il `java` che esegue `PrimeFinderApp` è un processo; l'editor con cui l'abbiamo scritta è un altro.

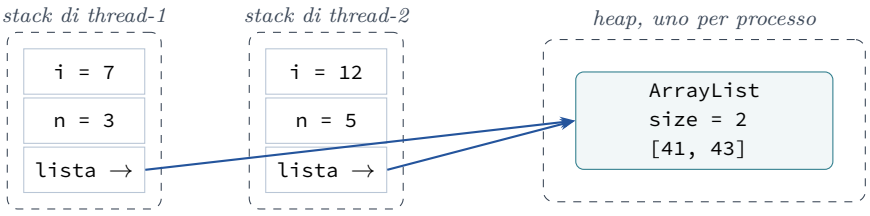
L'*isolamento* è la proprietà fondamentale del processo: non può leggere né scrivere la memoria di un altro, e se va in errore il sistema lo termina senza che gli altri se ne accorgano. È ciò che permette a un editor di continuare a funzionare mentre un altro programma va in crash. Il prezzo di questa protezione è doppio. Creare un processo costa: il sistema deve preparare uno spazio di indirizzamento intero, con le sue tabelle. E due processi, per scambiarsi anche un solo numero, devono passare da un canale esterno che il sistema controlla: un file, una *pipe*, un socket di rete. Per il nostro problema, dividere il conteggio dei primi fra due processi, vorrebbe dire avviare due JVM e far loro comunicare il risultato attraverso un file. Si può fare; non è il modo giusto.

2.2 Il thread

Un *thread* (filo) è un flusso di esecuzione *dentro* un processo. Ogni processo ne ha almeno uno, quello che esegue `main`; può crearne altri, e tutti condividono lo stesso spazio di indirizzamento. Ciò che ogni thread ha di suo è il minimo indispensabile per avanzare in modo indipendente dagli altri:

Privato del thread	**Condiviso dal processo**
stack (variabili locali, chiamate in corso)	heap (tutti gli oggetti)
program counter (l'istruzione a cui è arrivato)	codice del programma
registri del processore	variabili globali e statiche
stato (pronto, in esecuzione, bloccato)	file aperti, socket, finestre

La figura mostra la stessa cosa dal punto di vista della memoria: due thread, due stack, un solo heap. Le variabili locali di ciascuno (`i`, `n`) stanno nel suo stack e nessun altro le vede; l'oggetto a cui entrambi hanno un riferimento sta nell'heap, ed è uno solo.



variabili locali private, oggetti condivisi

La colonna di destra della tabella, l'heap condiviso, è la ragione per cui i thread esistono e, insieme, la fonte di ogni loro problema. Esistono perché creare un thread costa molto meno di un processo (uno stack e qualche registro, non uno spazio di indirizzamento) e perché due thread si scambiano dati semplicemente leggendo e scrivendo gli stessi oggetti: il thread che conta i primi da 1 a 50 milioni può scrivere il risultato in un campo, e chi lo aspetta lo legge. Sono un problema per lo stesso motivo: se due thread scrivono lo stesso oggetto nello stesso momento, nessuno li protegge, e l'oggetto resta a metà. Il capitolo 5 è dedicato a questo.

Definizione 2.1: Processo e thread

Un *processo* è un programma in esecuzione con uno spazio di indirizzamento proprio e isolato. Un *thread* è un flusso di esecuzione dentro un processo, con stack, registri e stato propri, che condivide con gli altri thread del processo l'heap, il codice e le risorse aperte.

2.3 Scheduler e context switch

I thread pronti a eseguire sono quasi sempre più dei core disponibili: su un portatile con sedici core girano centinaia di thread, fra il sistema e i programmi aperti. Lo *scheduler* del sistema operativo decide quale thread gira su quale core e per quanto. A ogni thread tocca una *fetta di tempo* (qualche millisecondo); scaduta la fetta, il sistema sospende il thread, ne salva i registri e il program counter, e carica quelli di un altro. Questo scambio si chiama *context switch*: costa poco, qualche microsecondo, ma non niente, e con migliaia di thread diventa una parte visibile del lavoro.

La sospensione è *preemptive*: il thread non deve collaborare e non può impedirla. Il sistema lo ferma dove si trova, anche a metà di ciò che per il programmatore è una sola istruzione. Da qui la prima regola mentale della programmazione con i thread, quella che spiega da sola metà del capitolo 5:

Osservazione 2.1

Fra due istruzioni qualsiasi di un thread può passare un tempo arbitrario, durante il quale gli altri thread fanno quello che vogliono. Non esiste "abbastanza veloce da non essere interrotti": un programma è corretto solo se lo è per ogni punto in cui lo scheduler potrebbe fermarlo.

Dal punto di vista dello scheduler un thread è in uno di tre stati. *In esecuzione*: sta girando su un core. *Pronto*: vorrebbe girare e aspetta che un core si liberi. *Bloccato*: aspetta qualcosa che non è un core (un dato dal disco, un timer, un lock tenuto da altri) e fino ad allora non viene nemmeno considerato. La distinzione fra gli ultimi due conta più di quanto sembri: un thread bloccato non consuma processore. Per questo un programma con cento thread in attesa di rete costa quasi nulla, mentre dieci thread che calcolano saturano dieci core; e per questo, nel capitolo 4, il numero giusto di thread dipende da che cosa fanno.

2.4 Come fanno i sistemi moderni

I tre sistemi su cui gira il materiale del corso realizzano i thread in modi diversi nei dettagli e uguali nella sostanza: il kernel conosce i thread e li schedula direttamente, un thread del linguaggio corrisponde a un thread del kernel (modello *uno a uno*), e ogni thread ha uno stack riservato in anticipo.

	Linux	**macOS**	**Windows**
che cos'è un thread	un *task* come il processo, creato con `clone` condividendo memoria e file; il processo è il gruppo dei suoi task	un thread Mach del kernel XNU; sopra, la libreria `pthread`	l'unità di esecuzione del kernel; il processo è il contenitore di risorse
chiamata di sistema	`clone`, tramite `pthread_create` (NPTL)	`pthread_create`, che chiama il kernel Mach	`CreateThread`
scheduler	CFS fino alla 6.5, EEVDF dalla 6.6: ogni thread riceve una quota equa di tempo	a priorità con *classi di qualità del servizio* (interattivo, utente, utility, sfondo) che pesano anche sulla scelta fra core veloci ed efficienti	a priorità (32 livelli) con innalzamento temporaneo per i thread interattivi e quelli appena sbloccati
pool di sistema	nessuno: le librerie se lo fanno	Grand Central Dispatch: code di compiti servite da un pool del sistema	thread pool nativo e *fiber* cooperativi
stack per thread	8 MB virtuali per default (`ulimit -s`)	512 KB (8 MB il principale)	1 MB

Il punto comune è che oggi tutti schedulano thread, non processi: il processo è un contenitore di risorse, il thread è ciò che gira. L'altra cosa che la tabella mostra è l'ultima riga: uno stack si riserva per ogni thread. È memoria *virtuale*, consumata davvero solo quando il thread la usa, ma è il motivo per cui "un thread per ogni cosa" smette di funzionare attorno a qualche migliaio di thread, e per cui il capitolo 7 introdurrà un tipo di thread che quello stack non ce l'ha.

Per vedere il livello sotto Java basta un programma C, visto che anche il C fa parte del corso; chi non lo conosce legga i commenti. La libreria `pthread` è la stessa su Linux e macOS:

```c
/* thread.c: compilare con  gcc thread.c -o thread -lpthread */
#include <pthread.h>
#include <stdio.h>

void *lavoro(void *arg) {                    /* il compito: una funzione */
    printf("ciao da un altro thread (%s)\n", (char *) arg);
    return NULL;
}

int main(void) {
    pthread_t t;
    pthread_create(&t, NULL, lavoro, "T1");   /* avvia: clone() sotto */
    printf("ciao dal thread principale\n");
    pthread_join(t, NULL);                    /* aspetta che finisca */
    return 0;
}
```

Tre cose da notare, perché le ritroveremo identiche in Java nel capitolo 3. Il compito è una funzione passata come argomento: il thread è *chi* esegue, la funzione è *che cosa*. `pthread_create` ritorna subito, senza aspettare che il nuovo thread parta. E le due stampe possono uscire in un ordine o nell'altro: dopo la creazione ci sono due thread pronti, e chi gira per primo lo decide lo scheduler, non il programma. Eseguendo `./thread` dieci volte si vedono, prima o poi, entrambi gli ordini.

2.5 L'astrazione di Java

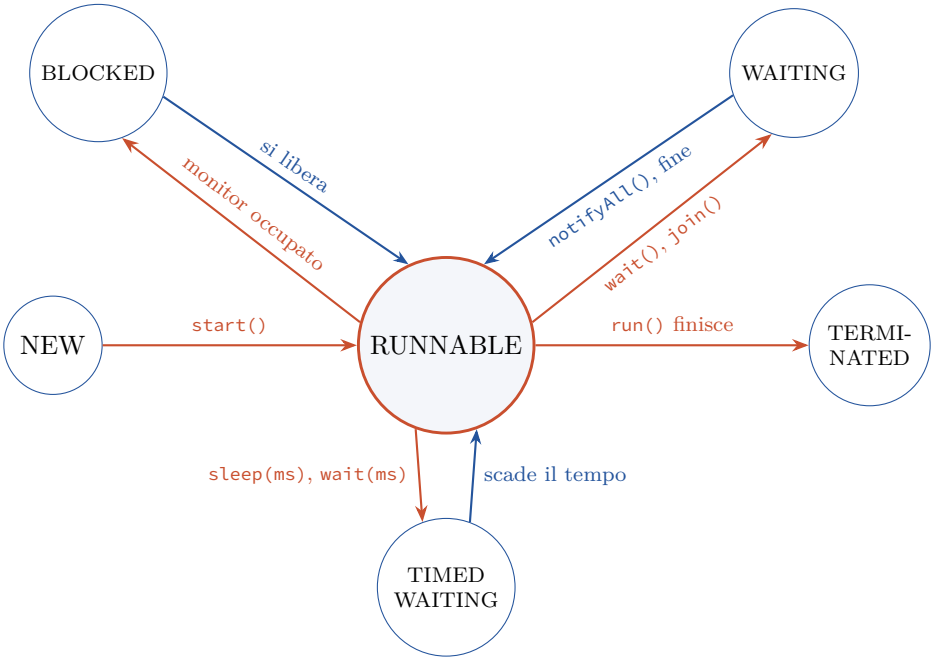
In Java un thread è un oggetto della classe `java.lang.Thread`. Quando si chiama `start()` la JVM chiede al sistema operativo un thread nativo (con `pthread_create` o `CreateThread`) e vi lega l'oggetto: da quel momento il thread Java e il thread del kernel sono la stessa cosa, con lo stesso ciclo di vita, lo stesso stack (un megabyte per default, regolabile con `-Xss`) e lo stesso scheduler. È il *thread di piattaforma*, e l'uno a uno della tabella vale per tutti i thread che incontreremo fino al capitolo 7, dove Java 21 aggiunge un secondo tipo.

Il programma non è mai solo. Già un `main` che stampa una riga gira in una JVM con una ventina di thread: `main`, i thread del garbage collector, quelli del compilatore JIT, `Reference Handler`, `Finalizer`, `Signal Dispatcher`. Li mostra tutti lo strumento `jstack` del JDK, che stampa lo stack di ogni thread di una JVM in esecuzione; dal programma stesso, `Thread.getAllStackTraces()` ne vede solo una parte, i cinque o sei che sono oggetti `Thread` (quelli del garbage collector e del JIT sono thread nativi, senza un oggetto Java); il debugger di Eclipse (vista *Debug*, sospendendo il programma) sta nel mezzo.

2.5.1 Gli stati di un thread Java

La classe `Thread` descrive lo stato con l'enumerazione `Thread.State`, che raffina i tre stati del sistema operativo e si legge con `t.getState()`:

Stato Java	Che cosa significa	Per il SO
`NEW`	creato, `start()` non ancora chiamato	non esiste
`RUNNABLE`	in esecuzione, pronto, o in attesa di I/O	in esecuzione / pronto / bloccato
`BLOCKED`	aspetta di entrare in un blocco `synchronized`	bloccato
`WAITING`	`wait()`, `join()`, `park()` senza scadenza	bloccato
`TIMED_WAITING`	`sleep(ms)`, `wait(ms)`, `join(ms)`	bloccato
`TERMINATED`	`run()` è finito, bene o male	non esiste più



in rosso le transizioni che il programma chiede, in blu i risvegli

Una sorpresa della tabella: un thread che aspetta un file o la rete è `RUNNABLE` per Java, anche se per il sistema è bloccato. La JVM non sa che cosa fa una chiamata nativa, e lo stato riflette solo ciò che la JVM controlla: i monitor, le attese esplicite, i timer. `BLOCKED` e `WAITING` sono i due stati del capitolo 5, e vale la pena memorizzare la differenza fin d'ora: `BLOCKED` è "vorrei entrare e c'è qualcuno", `WAITING` è "sono entrato e aspetto che mi chiamino".

2.5.2 Daemon, priorità, e come si guarda dentro

Due dettagli che tornano utili. La JVM termina quando finisce l'ultimo thread *non daemon*: un thread marcato `setDaemon(true)` prima di `start()` è di servizio e non tiene vivo il programma (così sono i thread del garbage collector). Nel capitolo 4 questa regola spiegherà un programma che non termina mai. Le priorità (`setPriority`, da 1 a 10) sono un suggerimento allo scheduler che su Linux per default viene ignorato e altrove conta poco: un programma corretto non dipende mai dalle priorità.

Quando un programma con i thread si ferma senza errori, l'unico modo di capire perché è guardare che cosa sta facendo ciascun thread. `jstack <pid>` lo fa dall'esterno, senza toccare il programma; `jps` dà il `pid` delle JVM in esecuzione.

```
$ jps
41230 PrimeFinderApp
$ jstack 41230
"main" #1 prio=5 os_prio=31 cpu=14210ms tid=0x... nid=0x1703 runnable
   java.lang.Thread.State: RUNNABLE
        at PrimeTester.isPrime(PrimeTester.java:6)
        at PrimeTester.count(PrimeTester.java:13)
        at PrimeFinderApp.conta(PrimeFinderApp.java:23)
        ...
"AWT-EventQueue-0" #15 prio=6 os_prio=31 cpu=120ms tid=0x... waiting on condition
   java.lang.Thread.State: WAITING (parking)
        at java.base@21/jdk.internal.misc.Unsafe.park(Native Method)
        ...
"Reference Handler" #9 daemon prio=10 os_prio=31 cpu=0ms tid=0x... waiting on
    condition
   java.lang.Thread.State: RUNNABLE
        ...
```

Ricetta: leggere un `jstack`

1. Ogni blocco è un thread: nome, se è daemon, priorità, tempo di CPU consumato (`cpu=`), stato per il sistema (`runnable`, `waiting on condition`, `waiting for monitor entry`).
2. La seconda riga è lo stato Java della tabella; sotto, lo stack dalla chiamata più recente alla più vecchia: la prima riga "propria" del programma (non `java.base`) dice dove si trova.
3. Cercare i thread `BLOCKED`: ciascuno dice "waiting to lock <X>" e, poco sotto, chi ha "locked <X>". Se due thread si aspettano a vicenda, `jstack` lo scrive in fondo: "Found one Java-level deadlock".
4. Un thread con `cpu=` alto e `RUNNABLE` sta calcolando: nel `jstack` sopra è `main`, fermo dentro `isPrime`, mentre il thread degli eventi di Swing aspetta. È la fotografia della finestra congelata del capitolo 1: ci torneremo nel capitolo 8 sapendo chi è `AWT-EventQueue-0`.

Attenzione 2.1: "Il mio thread è lento perché ha priorità bassa"

Alzare la priorità non rende un thread più veloce, e abbassarla non lo rende educato: sui sistemi di oggi le priorità di Java sono quasi sempre ignorate, e un programma che *funziona* solo con certe priorità ha un errore di sincronizzazione che le priorità nascondono. Se due thread devono eseguire in un certo ordine, l'ordine si impone con gli strumenti del capitolo 3 e del capitolo 5, mai con `setPriority`.

Esercizi

Esercizio 2.1: Che cosa è di chi (⋆)

Due thread eseguono lo stesso metodo `count(from, to)` di `PrimeTester` su intervalli diversi e, alla fine, aggiungono il risultato a una stessa `ArrayList<Long>`. Elencare che cosa, in questo scenario, è privato di ciascun thread e che cosa è condiviso. Dove sta `n`? Dove sta la lista? Il codice di `isPrime` esiste in due copie?

Esercizio 2.2: Dieci gigabyte (⋆)

Su Linux ogni thread riserva per default 8 MB di stack. Un programma crea diecimila thread che dormono: dieci gigabyte, su un portatile con otto. Perché funziona lo stesso? E che cosa succederebbe se ogni thread, invece di dormire, chiamasse una funzione ricorsiva molto profonda?

Esercizio 2.3: In che stato (⋆)

Dire lo stato `Thread.State` di un thread che: (a) sta leggendo un file di un gigabyte da un disco lento; (b) ha chiamato `Thread.sleep(1000)`; (c) è stato creato con `new Thread(r)` e nient'altro; (d) sta eseguendo `isPrime` su un core; (e) sta eseguendo `isPrime`, ma lo scheduler lo ha sospeso per dare il core a un altro.

Esercizio 2.4: Preemptive (⋆⋆)

Un collega sostiene che `count++` non può dare problemi "perché è una sola istruzione e dura un nanosecondo, nessuno fa in tempo a interrompere". Con quanto visto in questo capitolo, spiegare in tre frasi perché il ragionamento è sbagliato, anche senza sapere ancora come è fatto `count++`.

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 2.5: I thread di una JVM vuota (⋆)

Scrivere un programma il cui `main` non fa altro che stampare, uno per riga, nome, stato e flag daemon di tutti i thread della JVM, ottenuti con `Thread.getAllStackTraces().keySet()`. Eseguirlo tre volte: il numero di thread è sempre lo stesso? Quali sono daemon e quali no?

Soluzione: progetto `sol-d04-c02`, package `it.unibs.pajc.redolfi.e05`.

Attenzione 2.2: I thread non sono i vostri

Chi vede per la prima volta una manciata di thread in un programma che ne ha scritto uno tende a cercarli nel proprio codice. Non ci sono: sono della JVM (gestione dei riferimenti, segnali, finalizzazione) e sono tutti daemon, per questo il programma termina appena `main` finisce. L'unico non daemon è `main`. Con `jstack` se ne vedono molti di più, perché mostra anche i thread nativi.

Esercizio 2.6: Mille thread che dormono (⋆⋆)

Scrivere un programma che crea e avvia N thread, ciascuno dei quali dorme per dieci secondi, misurando con `System.nanoTime()` quanto tempo passa fra il primo e l'ultimo `start()`. Provare con N uguale a 100, 1 000 e 10 000 e riportare i tempi; mentre dormono, osservare con `jstack` (o con il monitor di sistema) la memoria e lo stato dei thread. Il tempo di creazione cresce in modo lineare?

Soluzione: progetto `sol-d04-c02`, package `it.unibs.pajc.redolfi.e06`.

Esercizio 2.7: Tre stati in una fotografia (⋆⋆)

Scrivere un programma che avvia tre thread e li porta in tre stati diversi: `TIMED_WAITING` (con `sleep`), `WAITING` (con `join` su un thread che non finisce mai) e `BLOCKED` (provando a entrare in un blocco `synchronized` su un oggetto il cui monitor è tenuto da `main`). Dopo un secondo il `main` stampa `getState()` di ciascuno e termina il programma. Verificare anche con `jstack` che gli stati corrispondano.
Suggerimento: per tenere il monitor, `main` entra in `synchronized (lock)` e dentro al blocco avvia il terzo thread e aspetta.

Soluzione: progetto `sol-d04-c02`, package `it.unibs.pajc.redolfi.e07`.

Risposte

Risposta 2.1. Privati: lo stack di ciascun thread, quindi le variabili locali `n`, `i`, `d` e i parametri `from` e `to` di ogni chiamata in corso, più il program counter e i registri. Condivisi: la lista (un solo oggetto nell'heap, raggiunto da due riferimenti), il codice di `isPrime` e `count` (esiste in una copia sola, nell'area del codice: due thread possono eseguire *lo stesso metodo* nello stesso momento, ciascuno con il proprio stack), le variabili statiche. `n` è privato finché resta locale; nel momento in cui viene aggiunto alla lista, il suo valore passa in un oggetto condiviso, e l'`add` di due thread sulla stessa `ArrayList` è esattamente il problema del capitolo 5.

Risposta 2.2. Gli 8 MB sono *riservati*, non allocati: il sistema promette al thread un intervallo di indirizzi, e assegna memoria fisica solo alle pagine che il thread tocca davvero. Un thread che dorme ha usato poche decine di kilobyte del suo stack, e diecimila thread costano qualche centinaio di megabyte reali. La memoria virtuale riservata è però un limite anch'essa (lo spazio di indirizzamento e i parametri del sistema), ed è una delle ragioni per cui attorno a qualche migliaio di thread le cose smettono di funzionare. Con una ricorsione profonda ogni thread toccherebbe davvero il suo stack, fino agli 8 MB oppure fino a `StackOverflowError`; diecimila thread così esaurirebbero la memoria fisica. I thread virtuali del capitolo 7 nascono per il primo caso: non riservano niente, e lo stack cresce nell'heap quanto serve.

Risposta 2.3. (a) `RUNNABLE`: per Java è in esecuzione dentro una chiamata nativa, anche se il sistema lo tiene bloccato in attesa del disco. (b) `TIMED_WAITING`. (c) `NEW`: per il sistema operativo non esiste ancora. (d) `RUNNABLE`. (e) Ancora `RUNNABLE`: "pronto" e "in esecuzione" sono lo stesso stato per Java, perché la distinzione la fa lo scheduler e cambia ogni millisecondo. Gli stati Java descrivono ciò che il *programma* ha chiesto (aspettare un monitor, un tempo, un altro thread), non ciò che il sistema sta facendo.

Risposta 2.4. Primo: la sospensione è preemptive, il sistema ferma un thread dove si trova senza chiedere, e "un nanosecondo" è solo una probabilità bassa, non zero: su un programma che esegue `count++` dieci milioni di volte una probabilità bassa diventa una certezza. Secondo: con più core non serve nemmeno la sospensione, perché due thread eseguono *davvero* nello stesso istante e i loro accessi alla stessa variabile si sovrappongono senza che nessuno venga interrotto. Terzo: "una sola istruzione" nel sorgente non vuol dire una sola istruzione per il processore, e il capitolo 5 mostrerà che `count++` sono tre; ma anche se fosse una, le prime due ragioni basterebbero a non fidarsi.

Capitolo 3

Il primo thread

Nel capitolo 2 abbiamo visto che cosa è un thread per il sistema operativo e come Java vi si appoggia. Ora lo scriviamo. Il problema di questo capitolo è il primo dei due del capitolo 1: far lavorare due flussi insieme, e alla fine mettere d'accordo i loro risultati. Alla fine sapremo avviare un thread, aspettarlo, fermarlo in modo pulito, e avremo diviso il conteggio dei primi fra due thread scritti a mano, dimezzando il tempo. Il capitolo 4 toglierà poi di mezzo i thread scritti a mano; i *compiti* che scriviamo qui resteranno identici.

3.1 Compito e lavoratore

Il programma C del capitolo 2 separava due cose: la funzione da eseguire e il thread che la esegue. Java fa la stessa separazione con due tipi. Il *compito* è un `Runnable`, l'interfaccia funzionale con il solo metodo `void run()`; il *lavoratore* è un oggetto `Thread`, che riceve il compito nel costruttore. Il primo programma esegue lo stesso compito due volte, prima da sé e poi attraverso un thread, per vedere la differenza:

```
Runnable task = () -> {
    String nome = Thread.currentThread().getName();
    System.out.println("Ciao, sono " + nome);
};

task.run();                          // 1: lo eseguo io

Thread thread = new Thread(task);
thread.start();                      // 2: lo esegue un altro thread

System.out.println("Fatto!");        // 3
```

```
Ciao, sono main
Fatto!
Ciao, sono Thread-0
```

La prima riga non sorprende: `task.run()` è una chiamata di metodo come le altre, eseguita dal thread `main`, che infatti si presenta con il suo nome. Le due successive sì: "Fatto!" esce *prima* del saluto del secondo thread. `start()` chiede al sistema un nuovo thread (è il `pthread_create` del capitolo 2) e ritorna subito, senza aspettare che quello parta; `main` prosegue e stampa; nel frattempo `Thread-0` viene schedulato, esegue `run()` e stampa a sua volta. L'ordine delle ultime due righe non è garantito: su una macchina carica, o con un'altra JVM, potrebbero invertirsi. È il primo insegnamento pratico, e vale per tutta la dispensa: due thread non hanno un ordine reciproco, a meno che il programma non lo imponga.

Attenzione 3.1: `run()` al posto di `start()`

`thread.run()` compila, gira e non crea nessun thread: `run` è un metodo normale, e lo esegue chi lo chiama. Il programma resta sequenziale e non dà nessun segnale, se non che è lento quanto prima. È l'errore più frequente del principiante, e la domanda più frequente degli esami: dato un frammento, quanti thread crea? Solo gli `start()` contano. E uno `start()` chiamato due volte sullo stesso oggetto non crea un secondo thread: lancia `IllegalThreadStateException`, perché un `Thread` si avvia una volta sola.

3.2 Un compito con dei dati

Un compito di una riga sta bene in una lambda. Un compito con dei dati (un nome, quante volte ripetere, quanto aspettare) ha bisogno di un posto dove tenerli: una classe che implementa `Runnable`, con i dati nei campi. Questa, che stampa `nStep` righe a intervalli di `delay` millisecondi, servirà per tutto il capitolo:

```
public class MyTask implements Runnable {
    private final String nome;
    private final int nStep, delay;

    public MyTask(String nome, int nStep, int delay) {
        this.nome = nome; this.nStep = nStep; this.delay = delay;
    }

    public void run() {
        for (int i = 0; i < nStep; i++) {
            System.out.printf("%s, passo %2d [%s]%n",
                    nome, i, Thread.currentThread().getName());
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                return;                          // ci hanno chiesto di smettere
            }
        }
    }
}
```

Due compiti, due thread, e `main` che dopo averli avviati non ha più niente da fare:

```
new Thread(new MyTask("A", 5, 30)).start();
new Thread(new MyTask("B", 5, 20)).start();
System.out.println("main ha finito");
```

```
main ha finito
A, passo  0 [Thread-0]
B, passo  0 [Thread-1]
B, passo  1 [Thread-1]
A, passo  1 [Thread-0]
B, passo  2 [Thread-1]
...
```

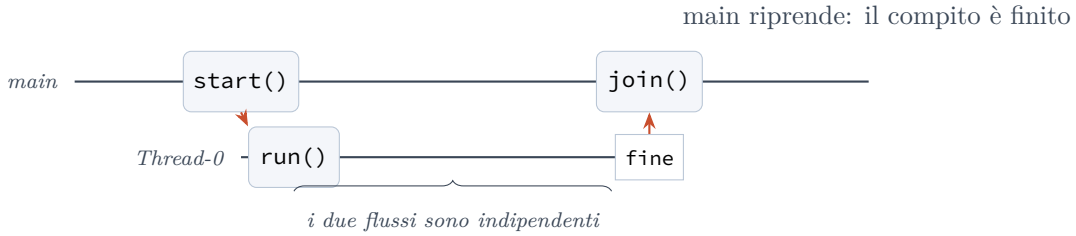
`main` finisce subito, ma il programma no: la JVM resta viva finché `A` e `B`, che non sono daemon (capitolo 2), non hanno completato il loro `run`. Le righe si intrecciano secondo i tempi dei due `sleep` e le decisioni dello scheduler; eseguendo due volte si ottengono due intrecci diversi. Il `try/catch` attorno a `sleep` è obbligatorio e non è un dettaglio: è l'argomento della sezione 3.4.

3.3 Aspettare: `sleep` e `join`

Due modi di aspettare, diversi per natura. `Thread.sleep(ms)` mette il thread *corrente* in `TIMED_WAITING` per almeno `ms` millisecondi: non consuma processore, e *almeno* vuol dire che il risveglio dipende dallo scheduler, non dall'orologio. `t.join()` mette il thread corrente in `WAITING` finché `t` non termina: è il modo per imporre un ordine fra due thread, quando serve.

```
Thread a = new Thread(new MyTask("A", 5, 30));
a.start();
a.join();                                    // main aspetta A
System.out.println("A ha finito, ora tocca a main");
```

La figura mostra i due thread nel tempo: `start()` crea il secondo flusso, `join()` lo riunisce al primo. Fra i due punti i flussi sono indipendenti; dopo il `join`, `main` sa che `run()` è finito, e può leggere ciò che il compito ha scritto nei suoi campi. Questa garanzia, "ciò che `A` ha fatto prima di finire è visibile a chi fa `join` su di lui", è la prima delle regole di visibilità del capitolo 5, e è quella su cui si regge l'esempio guida di questo capitolo.



Entrambi i metodi dichiarano `InterruptedException`, un'eccezione *checked* che il compilatore obbliga a gestire. Non è pedanteria: è l'intero meccanismo con cui in Java si chiede a un thread di smettere.

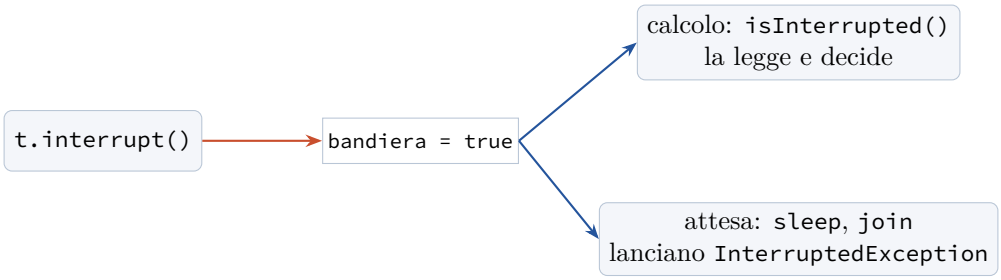
3.4 Fermare un thread: l'interruzione

Un thread non si può uccidere dall'esterno. C'era un `stop()`, deprecato da decenni e oggi reso inoperante, perché fermava il thread dove si trovava lasciando gli oggetti a metà: esattamente ciò che la regola "fra due istruzioni può passare un tempo arbitrario" insegna a temere, ma per sempre. Si può solo *chiedere* a un thread di fermarsi, e il thread deve collaborare. La richiesta è `t.interrupt()`, che alza una *bandiera* nel thread `t`; il thread la vede in due modi.

- Se sta *calcolando*, controllando `Thread.currentThread().isInterrupted()` a intervalli ragionevoli, per esempio a ogni giro di ciclo.
- Se è in *attesa* (`sleep`, `join`, `wait`, una coda bloccante), l'attesa viene interrotta e lancia `InterruptedException`, abbassando la bandiera.

Ecco perché `MyTask` esce dal ciclo con `return` nel `catch`: un `interrupt()` ricevuto durante lo `sleep` è la richiesta di smettere, e il compito la rispetta. Un ciclo di solo calcolo, senza attese, deve invece controllare la bandiera da sé:

```
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        faiUnPassoDiCalcolo();
    }
}
```



chi chiede alza la bandiera; il thread la vede e sceglie

Attenzione 3.2: Il `catch` vuoto

```
try {
    Thread.sleep(delay);
} catch (InterruptedException e) { }      // "non succede mai"
```

È il codice che Eclipse genera e che si vede in troppi programmi, e ingoia la richiesta: l'eccezione ha *abbassato* la bandiera, il ciclo riprende e il thread non si fermerà mai. Nel `catch` si fa una di due cose: si esce (`return`, come in `MyTask`), oppure, se non si è i proprietari del ciclo, si rialza la bandiera con `Thread.currentThread().interrupt()` e si lascia che sia chi sta sopra a decidere. Un parente stretto: `Thread.interrupted()` (statico) legge la bandiera *e la azzera*, quindi un secondo controllo subito dopo legge `false`; per guardare senza toccare si usa `isInterrupted()`.

Negli esempi che seguono una pausa serve spesso solo a rendere visibile l'intreccio dei thread; per non ripetere il `try` ogni volta useremo un metodo di comodo che applica la seconda regola, e che vale la pena copiare in ogni progetto:

```
static void pausa(long ms) {
    try {
        Thread.sleep(ms);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();   // rialza la bandiera e ritorna
    }
}
```

Ricetta: un thread che si ferma bene

1. Chi vuole fermarlo chiama `t.interrupt()`, mai `stop()`.
2. Un ciclo di calcolo controlla `isInterrupted()` a ogni giro, o ogni n giri se un giro è brevissimo, ed esce.
3. Un'attesa lascia che `InterruptedException` la interrompa; nel `catch` si esce dal ciclo, oppure si rialza la bandiera.
4. Se serve, chi ha chiesto lo stop fa `t.join()` per sapere che il thread ha davvero finito.

3.5 Lambda e stato

Un `Runnable` è un'interfaccia funzionale, quindi un compito può essere una lambda, `new Thread(() -> ...)`, e lo sarà quasi sempre. Vale la regola delle variabili catturate: la lambda vede le variabili locali del contesto solo se sono *effectively final*. La regola, che altrove sembra una pignoleria del compilatore, qui mostra il suo perché: la lambda verrà eseguita da un altro thread, su un altro stack, forse quando il metodo che l'ha creata è già finito e le sue variabili locali non esistono più. Catturare una copia del valore è l'unica cosa sensata, e una copia è sicura solo se l'originale non cambia.

Un compito che deve *modificare* uno stato (un contatore, un risultato) non può quindi tenerlo in una variabile locale del metodo che lo crea:

```
long totale = 0;
Thread t = new Thread(() -> totale += 1);   // errore: totale non e' effectively
    final
```

Lo stato va in un oggetto, come i campi di `MyTask`, oppure in un contenitore pensato per essere condiviso, come quelli del capitolo 6. Ed è qui che si vede la differenza fra estendere `Thread` e implementare `Runnable`: la seconda tiene insieme il compito e il suo stato in un oggetto che non sa, né deve sapere, da chi sarà eseguito. Nel capitolo 4 il lavoratore smetterà di essere un `Thread` creato a mano, e i compiti scritti come `Runnable` resteranno identici; quelli scritti estendendo `Thread` andrebbero riscritti.

3.6 L'esempio guida: i primi su due thread

Torniamo al conteggio dei primi del capitolo 1. Il lavoro si divide in due metà indipendenti; ogni metà è un compito con dei dati (l'intervallo) e un risultato. Il risultato, per la sezione 3.5, va in un campo dell'oggetto, che il thread scrive e `main` legge dopo il `join`:

```
public class PrimeCounter implements Runnable {
    private final long from, to;
    private long result;                              // scritto da run, letto dopo
    join

    public PrimeCounter(long from, long to) { this.from = from; this.to = to; }

    public void run()    { result = PrimeTester.count(from, to); }
    public long result() { return result; }
}
```

Il tempo lo misura un cronometro di tre righe, che useremo in tutta la dispensa:

```
public class Chrono {
    private final long t0 = System.nanoTime();
    public long millis() { return (System.nanoTime() - t0) / 1_000_000; }
}
```

Il `main` crea i due compiti, li avvia, li aspetta entrambi, somma:

```
long max = 100_000_000L;
PrimeCounter a = new PrimeCounter(1, max / 2);
PrimeCounter b = new PrimeCounter(max / 2 + 1, max);
Thread ta = new Thread(a), tb = new Thread(b);
Chrono chrono = new Chrono();

ta.start(); tb.start();                     // i due flussi partono
ta.join();  tb.join();                      // main aspetta entrambi
long nPrimi = a.result() + b.result();      // ora i risultati ci sono

System.out.printf("%d primi in %d ms%n", nPrimi, chrono.millis());
```

```
5761455 primi in 8120 ms
```

Da quattordici secondi a otto. Non a sette: la seconda metà dell'intervallo contiene i numeri più grandi, per cui `isPrime` prova più divisori, e il thread `b` finisce dopo `a`, che intanto è fermo. Il tempo totale è quello del più lento. Con sedici core a disposizione si vorrebbe dividere in sedici, o meglio in molti più pezzi di dimensione diversa, e assegnarli man mano a chi è libero; ma scrivere a mano sedici `Thread`, sedici `join` e sedici somme è già scomodo, e per mille compiti diventa impossibile. È il problema che l'executor del capitolo 4 risolve, con l'immagine della coda del capitolo 1.

Osservazione 3.1

Due cose di questo programma restano vere in tutta la dispensa. La prima: il thread scrive `result` e `main` lo legge *dopo* il `join`, mai prima; senza il `join`, `main` leggerebbe zero, o un valore a metà, o quello giusto, a seconda del momento. La seconda: i due compiti non condividono niente, ciascuno conta nel suo campo, e la condivisione avviene una volta sola, alla fine, in un punto in cui l'ordine è garantito. Un programma concorrente scritto così non ha bisogno di nessuno degli strumenti del capitolo 5: è il modo migliore di usare i thread, quando il problema lo permette.

Esercizi

Esercizio 3.1: Quanti thread (⋆)

Per ciascun frammento dire quanti thread *nuovi* vengono creati e che cosa succede:

```
Runnable r = () -> System.out.println("x");
// (a)
new Thread(r).start(); new Thread(r).start(); r.run();
// (b)
Thread t = new Thread(r); t.run(); t.run();
// (c)
Thread u = new Thread(r); u.start(); u.start();
// (d)
r.start();
```

Esercizio 3.2: Prevedere l'ordine (⋆)

Senza eseguirlo, dire che cosa stampa il programma e quali righe potrebbero scambiarsi di posto da un'esecuzione all'altra. Poi dire come modificarlo, con `join`, perché l'ordine sia sempre A, B, fine, e che cosa si perde facendolo.

```
Thread a = new Thread(() -> { pausa(50); System.out.println("A"); });
Thread b = new Thread(() -> { pausa(10); System.out.println("B"); });
a.start(); b.start();
System.out.println("fine");
```

Esercizio 3.3: Non si ferma (⋆)

Questo compito viene avviato in un thread, e dopo un secondo `main` chiama `t.interrupt()`. Il thread continua a stampare per sempre. Perché? Correggerlo in due modi diversi.

```
Runnable r = () -> {
    while (true) {
        System.out.println("lavoro");
        try { Thread.sleep(200); } catch (InterruptedException e) { }
    }
};
```

Esercizio 3.4: Catturata (⋆)

Dire quali di questi frammenti compilano e, per quelli che non compilano, perché la regola ha senso proprio con i thread.

```
int n = 5;
new Thread(() -> System.out.println(n)).start();            // (a)
int k = 0;
new Thread(() -> System.out.println(k)).start(); k++;       // (b)
int[] box = {0};
new Thread(() -> box[0]++).start();                         // (c)
```

Esercizio 3.5: Prima del `join` (⋆⋆)

Nel programma della sezione 3.6 si sposta la riga `long nPrimi = a.result() + b.result();` *prima* dei due `join`. Che cosa stampa, e perché il programma è sbagliato anche se per caso stampasse il numero giusto?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 3.6: Un contatore di secondi (⋆)

Scrivere un programma in cui un thread stampa un numero crescente ogni secondo (`1`, `2`, `3`, ...) e `main`, dopo cinque secondi e mezzo, gli chiede di fermarsi con `interrupt()` e aspetta che abbia finito con `join()`, poi stampa "fermato". Il thread deve uscire *subito* alla richiesta, non alla fine del secondo in corso.

Soluzione: progetto `sol-d04-c03`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 3.3: `sleep` dentro, `interrupt` fuori

Se il `catch` dell'`InterruptedException` non esce dal ciclo, il contatore continua: l'eccezione ha già abbassato la bandiera e il `while` non se ne accorge. E se si controlla la bandiera con `Thread.interrupted()` due volte, la seconda legge `false`.

Esercizio 3.7: Avanzamento a puntini (⋆⋆)

Scrivere un programma che avvia un thread di lavoro lungo (per esempio `PrimeTester.count(1, 30_000_000)`) e, mentre quello lavora, `main` stampa un punto ogni 250 ms sulla stessa riga; quando il lavoro finisce, `main` va a capo e stampa il risultato. Usare `isAlive()` oppure `join(ms)` con un timeout, e spiegare nel commento quale dei due si è scelto e perché.

Soluzione: progetto `sol-d04-c03`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 3.8: `MyTask` educato (⋆⋆)

Avviare tre `MyTask` con passi e ritardi diversi e, da `main`, interromperli tutti dopo un tempo fissato; ogni compito, quando viene interrotto, deve stampare "`<nome>: interrotto al passo <i>`" e terminare. Modificare `MyTask` per farlo, senza cambiare l'interfaccia della classe.

Soluzione: progetto `sol-d04-c03`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 3.9: Quattro thread a mano (⋆⋆)

Estendere il programma della sezione 3.6 a quattro `PrimeCounter` su quattro thread. Prima con quattro intervalli di uguale ampiezza: misurare il tempo di ciascun thread (ognuno stampa il suo `Chrono` alla fine) e osservare lo sbilanciamento. Poi scegliere i quattro estremi in modo che i tempi siano simili (suggerimento: il costo fino a n cresce come $n\sqrt{n}$) e misurare di nuovo.
Suggerimento: tenere i thread in un array e usare due cicli, uno per gli `start` e uno per i `join`; se i `join` stanno nello stesso ciclo degli `start`, i thread girano uno alla volta.

Soluzione: progetto `sol-d04-c03`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 3.1. (a) Due thread nuovi: i due `start()`; `r.run()` è una chiamata normale, eseguita da `main`, e stampa la terza `x`. (b) Nessuno: `run()` due volte, due `x` stampate da `main`, il `Thread t` resta in `NEW`. (c) Un thread: il secondo `start()` lancia `IllegalThreadStateException`. (d) Non compila: `Runnable` non ha un metodo `start`; solo `Thread` ce l'ha.

Risposta 3.2. Quasi certamente `fine`, `B`, `A`: `main` non aspetta nessuno e stampa per primo; `B` dorme meno di `A`. Ma nessuna delle tre posizioni è garantita: `fine` potrebbe uscire dopo `B` se `main` venisse sospeso subito dopo i due `start`, e su una macchina molto carica anche `A` e `B` potrebbero invertirsi, perché `sleep` garantisce un'attesa minima, non un ordine. Per imporlo: `a.start(); a.join(); b.start(); b.join();` e poi la stampa. Così però i due thread non girano più insieme: ogni ordine imposto è parallelismo tolto. Se si vuole che `A` e `B` lavorino in parallelo e solo `fine` aspetti entrambi: `a.start(); b.start(); a.join(); b.join();`, e l'ordine fra `A` e `B` resta libero.

Risposta 3.3. La richiesta di `interrupt()` arriva quasi sempre durante lo `sleep`, che lancia `InterruptedException` e *abbassa* la bandiera; il `catch` vuoto la ignora, il `while (true)` riprende e non c'è più nessuna traccia della richiesta. Prima correzione: `return` nel `catch`. Seconda: la condizione del ciclo diventa `!isInterrupted()` sul thread corrente e, nel `catch`, si rialza la bandiera con `interrupt()`, così la condizione la vede al giro seguente. La seconda forma è quella giusta quando il `sleep` sta dentro un metodo chiamato dal ciclo, che non può decidere da solo di uscire.

Risposta 3.4. (a) Compila: `n` non cambia mai. (b) Non compila: `k` viene modificata dopo, quindi non è effectively final. Il perché è evidente con i thread: la lambda gira su un altro thread, forse dopo il `k++`, forse prima, forse quando il metodo è già terminato e `k` non esiste più; l'unico valore che ha senso catturare è una copia, e una copia di qualcosa che cambia sarebbe una bugia. (c) Compila: `box` è effectively final (il riferimento non cambia), e la lambda modifica l'*oggetto* a cui punta. È lecito, ma ora l'array è condiviso fra due thread, e se anche `main` lo toccasse si sarebbe nel capitolo 5.

Risposta 3.5. Quasi sempre stampa `0 primi`: `main` legge i due campi un istante dopo gli `start`, quando i thread non hanno ancora finito (né, probabilmente, cominciato). Potrebbe anche leggere un valore giusto, se per caso i thread fossero velocissimi; e qui sta l'errore: senza il `join` non c'è nessuna regola che dica a `main` che cosa vedrà, né *quando* vedrà la scrittura fatta dall'altro thread. Il `join` dà entrambe le cose: il compito è finito, e ciò che ha scritto è visibile. Un programma che funziona "per caso" è il tema del capitolo 5.

Capitolo 4

Executor, Callable e Future

Il capitolo 3 si è chiuso con un programma che funziona e non scala: due thread scritti a mano, due `join`, due somme, e un tempo deciso dal più lento dei due. Vorremmo molti compiti, di dimensione diversa, assegnati man mano a chi è libero, su un numero di thread scelto in base alla macchina e non al problema. È l'immagine della coda del capitolo 1, e in Java si chiama *executor*. Alla fine del capitolo sapremo consegnare compiti a un pool, spegnerlo, ricevere da ogni compito un risultato attraverso un `Future`, e avremo misurato quanto davvero si guadagna contando i primi su tutti i core: non sedici volte, e il perché è la parte più istruttiva.

4.1 Perché non un thread per ogni compito

Con `new Thread(task).start()` ogni compito costa un thread del sistema operativo: uno stack riservato, una chiamata al kernel, un context switch in più da gestire (capitolo 2). Per dieci compiti è irrilevante; per diecimila è un problema di memoria e di tempo; e per un flusso continuo di piccoli compiti (le richieste di un server, gli eventi di un'interfaccia, i pezzi di un calcolo) è il modo sbagliato di pensare, perché il numero di thread finisce per dipendere da quanti compiti arrivano e non da quanti core ci sono. Il modello giusto è quello della figura del capitolo 1: pochi thread che vivono a lungo e una coda di compiti da cui pescano. In Java si chiama *executor*, sta nel package `java.util.concurrent`, e separa definitivamente il *che cosa* dal *chi*: i compiti restano i `Runnable` del capitolo 3, i thread non li creiamo più noi.

4.2 `ExecutorService` e le factory di `Executors`

Il primo executor ha un solo thread. Gli consegniamo i tre `MyTask` del capitolo 3 con `submit`, e guardiamo chi li esegue:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(new MyTask("A", 3, 30));
executor.submit(new MyTask("B", 3, 20));
executor.submit(new MyTask("C", 3, 10));
System.out.println("Fatto!");
executor.shutdown();
```

```
Fatto!
A, passo  0 [pool-1-thread-1]
A, passo  1 [pool-1-thread-1]
A, passo  2 [pool-1-thread-1]
B, passo  0 [pool-1-thread-1]
B, passo  1 [pool-1-thread-1]
...
```

`submit` accoda il compito e ritorna subito, come `start()`; "Fatto!" esce per primo. Ma un unico thread, `pool-1-thread-1`, esegue i tre compiti *in sequenza*, nell'ordine di arrivo: `B` comincia quando `A` ha finito. È la coda con un solo lavoratore. Cambiando una riga cambia il lavoratore, ma non i compiti:

```
ExecutorService executor = Executors.newCachedThreadPool();
```

```
Fatto!
A, passo  0 [pool-1-thread-1]
B, passo  0 [pool-1-thread-2]
C, passo  0 [pool-1-thread-3]
C, passo  1 [pool-1-thread-3]
B, passo  1 [pool-1-thread-2]
...
```

Ora i tre compiti si intrecciano come nel capitolo 3, su tre thread che il pool ha creato perché c'era lavoro per tre. Le factory della classe `Executors` coprono i casi ricorrenti:

Factory	**Che cosa fa**
`newSingleThreadExecutor()`	un thread, compiti in sequenza: serializza senza bloccare chi li manda
`newFixedThreadPool(n)`	`n` thread fissi e una coda illimitata: il caso tipico per il calcolo
`newCachedThreadPool()`	crea thread quando servono, li riusa, li elimina dopo un minuto di inattività: per molti compiti brevi che aspettano (I/O)
`newScheduledThreadPool(n)`	come il pool fisso, ma i compiti possono essere ritardati o periodici (capitolo 7)
`newWorkStealingPool()`	un `ForkJoinPool` con tanti thread quanti i core, in cui ogni thread ha la sua coda e ruba agli altri quando ha finito: lo stesso usato dagli stream paralleli
`newVirtualThreadPerTaskExecutor()`	un thread virtuale per ogni compito, senza pool (capitolo 7)

`ExecutorService` ha due metodi per consegnare un compito, e la differenza riguarda gli errori. `execute(Runnable)` lo esegue e basta: un'eccezione non catturata nel compito termina il thread del pool (che viene rimpiazzato) e viene stampata dal gestore di default, come in un thread normale. `submit(Runnable)` e `submit(Callable)` restituiscono un `Future` (sezione 4.5), e con `submit` l'eccezione *non* viene stampata: viene conservata nel `Future`, e chi non lo interroga non la vedrà mai.

Attenzione 4.1: Il compito che fallisce in silenzio

```
executor.submit(() -> { throw new IllegalStateException("qualcosa non va");
    });
```

Niente sulla console, niente nel log: il programma continua come se il compito fosse riuscito. Con `execute` al posto di `submit` la stessa riga stampa lo stack trace. Regola: `execute` per i compiti di cui non si vuole il risultato, `submit` solo se poi si chiama `get()` sul `Future`, dove l'eccezione riemerge.

4.3 Spegnere un executor

Il primo programma che tutti scrivono con un executor finisce così:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> System.out.println("Ciao da "
        + Thread.currentThread().getName()));
```

```
Ciao da pool-1-thread-1
_                                   <- e il programma non termina
```

I thread di un pool non sono daemon (capitolo 2) e restano in attesa di altri compiti, per sempre: la JVM, di conseguenza, non esce. Un executor va spento. `shutdown()` chiude l'ingresso: i compiti già in coda vengono completati, quelli nuovi rifiutati con `RejectedExecutionException`, e i thread terminano quando la coda si svuota. `shutdownNow()` è la versione brusca: svuota la coda, interrompe con `interrupt()` (capitolo 3: i compiti devono collaborare) quelli in corso, e restituisce la lista di quelli mai partiti. `awaitTermination(t, unit)` blocca il chiamante finché tutto non è finito, o fino allo scadere del tempo, e dice quale dei due è successo.

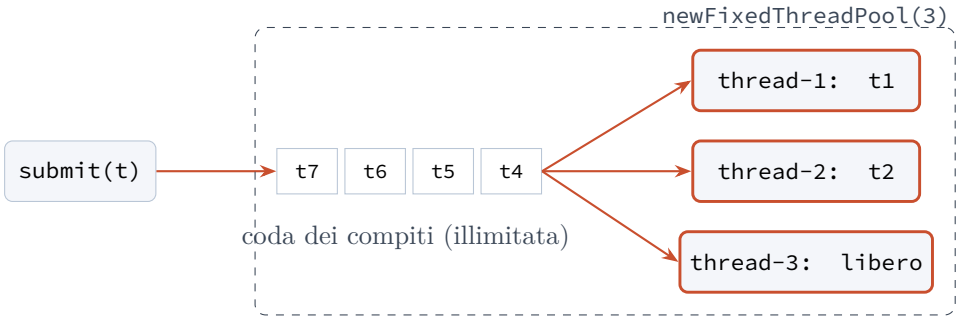
Ricetta: spegnere un executor

```
executor.shutdown();                        // 1. niente nuovi compiti
try {
    if (!executor.awaitTermination(5, TimeUnit.SECONDS)) {
        executor.shutdownNow();             // 2. troppo lenti: interrompi
    }
} catch (InterruptedException e) {
    executor.shutdownNow();                 // 3. interrotti noi: interrompi
    tutti
    Thread.currentThread().interrupt();
}
```

Da Java 19 `ExecutorService` è `AutoCloseable` e `close()` fa `shutdown` seguito da un'attesa illimitata: in un metodo che crea l'executor, lo usa e lo chiude, `try (ExecutorService ex = ...) { ... }` JAVA 19 è la forma più breve e corretta, e la useremo spesso.

4.4 Dentro un pool

Un pool è una coda più un gruppo di thread che eseguono tutti lo stesso ciclo: estrai un compito, eseguilo, ricomincia. Chi chiama `submit` non aspetta nessuno; chi esegue non sa chi ha mandato il compito.



Quanti thread? Dipende da che cosa fanno i compiti, e la risposta viene dai tre stati del capitolo 2. Se *calcolano* (CPU-bound), più thread dei core non servono: si alternano sugli stessi core pagando i context switch; `Runtime.getRuntime().availableProcessors()` dà il numero giusto. Se *aspettano* (I/O-bound: rete, disco, un altro servizio) la maggior parte del tempo sono bloccati e non occupano un core: possono essere molti di più, e il limite è la memoria degli stack. Il pool con `n` thread fissi e la coda illimitata è la scelta di default per il calcolo; il pool *cached* per i compiti brevi che aspettano; e per l'attesa in grande Java 21 offre un'alternativa migliore di entrambi, nel capitolo 7.

4.5 Valori di ritorno: `Callable` e `Future`

`Runnable.run()` non restituisce niente e non può lanciare eccezioni checked: va bene per un compito che agisce (stampa, aggiorna, scrive), non per uno che *calcola* un risultato. Nel capitolo 3 abbiamo aggirato il limite con un campo e un `join`. La libreria ha una soluzione più pulita: l'altra interfaccia funzionale dei compiti.

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

Un `Callable<Long>` che conta i primi è una lambda di una riga, senza campi e senza classe: `() -> PrimeTester.count(from, to)`. E `submit(Callable<V>)` restituisce subito un `Future<V>`: una promessa del risultato, che il compito riempirà quando avrà finito.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Future<Long> fa = executor.submit(() -> PrimeTester.count(1, 50_000_000));
Future<Long> fb = executor.submit(() -> PrimeTester.count(50_000_001,
    100_000_000));

System.out.println("compiti avviati, main e' libero");
long totale = fa.get() + fb.get();       // qui main aspetta
System.out.println("primi fino a 10^8: " + totale);
executor.shutdown();
```

```
compiti avviati, main e' libero
primi fino a 10^8: 5761455
```

È il programma del capitolo 3 senza `PrimeCounter`, senza `Thread` e senza `join`: `get()` fa il lavoro del `join` e in più consegna il valore. La figura segue un `Future` nel tempo:



il compito lavora; main è libero fino a `get()`, poi aspetta

`get()` è il punto di sincronizzazione: blocca il chiamante finché il risultato non c'è, poi lo restituisce; se il risultato c'è già, ritorna subito. Il resto dell'interfaccia serve a non bloccarsi alla cieca: `isDone()` risponde subito; `get(t, unit)` aspetta al massimo `t` e poi lancia `TimeoutException`; `cancel(true)` interrompe il compito se è in corso (con `interrupt()`, quindi il compito deve collaborare) e lo toglie dalla coda se non è ancora partito. Le eccezioni del compito seguono il risultato: se `call()` lancia, `get()` rilancia un'`ExecutionException` che *incarta* l'eccezione originale, recuperabile con `getCause()`; è qui che riemerge il fallimento silenzioso della sezione 4.2. `get()` dichiara anche `InterruptedException`, perché è un'attesa come `sleep` e `join`, e vale la stessa regola.

4.5.1 `invokeAll` e `invokeAny`

Per una collezione di compiti ci sono due scorciatoie. `invokeAll` li esegue tutti e restituisce la lista dei `Future`, *già completati*:

```
List<Callable<String>> compiti = List.of(
        () -> "alfa", () -> "beta", () -> "gamma");

List<String> risultati = new ArrayList<>();
for (Future<String> f : executor.invokeAll(compiti))
    risultati.add(f.get());              // non blocca: sono gia' finiti
```

`invokeAny` esegue i compiti e restituisce il risultato del *primo* che termina, annullando gli altri: utile quando più strade portano allo stesso risultato (tre server che rispondono alla stessa domanda, tre algoritmi per lo stesso problema) e si vuole la più veloce.

4.6 L'esempio guida: i primi su tutti i core

Abbiamo tutto per fare bene ciò che il capitolo 3 faceva a metà. Dividiamo l'intervallo in `nTask` pezzi, molti più dei thread, li consegniamo a un pool con tanti thread quanti i core e sommiamo i risultati nell'ordine in cui li abbiamo consegnati:

```
long max = 100_000_000L;
int nTask = 64;
int nThread = Runtime.getRuntime().availableProcessors();
Chrono chrono = new Chrono();
long nPrimi = 0;

try (ExecutorService executor = Executors.newFixedThreadPool(nThread)) {
    List<Future<Long>> risultati = new ArrayList<>();
    long perTask = max / nTask;
    for (int i = 0; i < nTask; i++) {
        long da = i * perTask + 1;
        long a  = (i == nTask - 1) ? max : (i + 1) * perTask;
        risultati.add(executor.submit(() -> PrimeTester.count(da, a)));
    }
    for (Future<Long> f : risultati)
        nPrimi += f.get();
}
System.out.printf("%d primi in %d ms con %d thread%n",
        nPrimi, chrono.millis(), nThread);
```

```
5761455 primi in 1402 ms con 16 thread
```

Da quattordici secondi a uno e quattro. Le variabili `da` e `a` sono dichiarate dentro il ciclo e mai modificate: effectively final, come vuole la lambda (capitolo 3). Il `try` con risorse chiude l'executor e aspetta la fine di tutti i compiti; i `get()` dentro al `try` aspettano ciascuno il suo. Su un portatile con 16 core (12 veloci e 4 efficienti), facendo variare `nThread`:

thread	tempo	accelerazione
1	14,4 s	1,0
2	7,1 s	2,0
4	3,9 s	3,7
8	2,0 s	7,1
12	1,5 s	9,9
16	1,4 s	10,6
32	1,3 s	11,3

Fino a 8 thread l'accelerazione è quasi lineare: ogni thread ha il suo core veloce. Oltre, i core efficienti rendono meno, e i thread in più dei core non aggiungono niente, perché si alternano sugli stessi core. Il limite non è Java: è la *legge di Amdahl*. Se una frazione p del lavoro è parallelizzabile e $1 - p$ no (qui: dividere l'intervallo, sommare i risultati, avviare i thread), con n core l'accelerazione è al massimo $1/((1 - p) + p/n)$, e per $n \to \infty$ tende a $1/(1 - p)$: la parte sequenziale, per piccola che sia, mette un tetto.

Perché 64 compiti e non uno per thread? Perché il lavoro non è uniforme, come abbiamo visto con due thread nel capitolo 3: l'ultimo pezzo dell'intervallo costa più del primo, e con 8 compiti su 8 thread sette thread finiscono e aspettano l'ottavo (2,6 s invece di 2,0 s nella stessa misura). Compiti *più numerosi dei thread* si bilanciano da soli: chi finisce prende il successivo dalla coda, e lo sbilanciamento si riduce a quello di un singolo compito. È la regola generale, ed è la ragione di essere della coda: tanti compiti piccoli, pochi thread.

Osservazione 4.1

`CompletableFuture` JAVA 8 è un `Future` che si può *comporre*: `thenApply` trasforma il risultato quando arriva, `thenCombine` ne unisce due, `thenAccept` lo consuma, tutto senza chiamare `get()` e senza bloccare nessuno. È il modello su cui sono costruite le librerie di rete asincrone, e ne parla il capitolo 10; in questa dispensa restiamo a `Future`, che basta per i casi di calcolo.

Esercizi

Esercizio 4.1: Non termina (⋆)

Questo programma stampa la somma e poi resta appeso. Perché? Correggerlo in due modi.

```
ExecutorService ex = Executors.newFixedThreadPool(4);
Future<Integer> f = ex.submit(() -> 2 + 2);
System.out.println(f.get());
```

Esercizio 4.2: Sparita (⋆)

Un compito consegnato con `submit` lancia `NullPointerException` dopo un secondo. Dire che cosa si vede sulla console (a) se nessuno chiama `get()` sul `Future`; (b) se `main` chiama `get()`: quale eccezione riceve, e come si arriva alla `NullPointerException`; (c) se al posto di `submit` c'era `execute`.

Esercizio 4.3: Un lavoratore (⋆)

Con `newSingleThreadExecutor()` si consegnano `MyTask("A", 2, 100)` e `MyTask("B", 2, 10)`. Scrivere l'output esatto. Che cosa cambia con `newFixedThreadPool(2)`? E l'ordine, in quel caso, è garantito?

Esercizio 4.4: Quanti thread (⋆⋆)

Scegliere la factory e il numero di thread, motivando: (a) mille immagini da ridimensionare, tutte in memoria; (b) mille file da scaricare da un server lento; (c) un registro di eventi in cui più thread scrivono e che deve restare in ordine di arrivo; (d) un compito ogni dieci minuti.

Esercizio 4.5: `get` dopo `invokeAll` (⋆⋆)

Nel frammento di `invokeAll` della sezione 4.5 il commento dice che `f.get()` non blocca. Perché? E se uno dei tre compiti lancia un'eccezione, che cosa fa `invokeAll`, e che cosa fa il `get()` corrispondente?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 4.6: Numeri di Armstrong (⋆)

Un numero è di Armstrong se è uguale alla somma delle sue cifre elevate al numero di cifre ($153 = 1^3 + 5^3 + 3^3$). Scrivere una classe `Armstrong` con un metodo `static List<Long> trova(long da, long a)` e un `main` che cerca i numeri di Armstrong fino a 10^8 dividendo l'intervallo in 32 compiti `Callable` su un pool con tanti thread quanti i core, raccoglie i `Future` e stampa i risultati in ordine crescente con il tempo impiegato. Confrontare con la versione a un thread.

Soluzione: progetto `sol-d04-c04`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 4.2: I risultati nell'ordine dei `Future`, non dell'arrivo

I compiti finiscono in un ordine qualsiasi; i `Future`, letti nell'ordine in cui sono stati creati, restituiscono i risultati nell'ordine degli intervalli. Chi raccoglie i risultati in una lista *dentro* i compiti, man mano che finiscono, ottiene un ordine casuale e, se la lista è una `ArrayList`, un errore ancora peggiore (capitolo 5).

Esercizio 4.7: Il programma che non muore (⋆)

Scrivere un programma che crea un `newFixedThreadPool(3)`, vi consegna cinque compiti brevi, e non lo spegne. Osservarlo con `jstack`: in che stato sono i tre thread, e dentro quale metodo si trovano? Poi aggiungere lo spegnimento secondo la ricetta della sezione 4.3, e infine riscriverlo con il `try` con risorse.

Soluzione: progetto `sol-d04-c04`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 4.8: `Matrix.mapMT` (⋆⋆)

Scrivere una classe `Matrix` con un array bidimensionale di `double`, un costruttore che la riempie di valori casuali, e due metodi: `Matrix map(DoubleUnaryOperator f)`, che restituisce una nuova matrice applicando `f` a ogni elemento, e `Matrix mapMT(DoubleUnaryOperator f, int nThread)`, che fa lo stesso dividendo le righe fra `nThread` compiti con `invokeAll`. Misurare i due metodi su una matrice 4000×4000 con una funzione costosa (per esempio `x -> Math.sin(x) * Math.exp(x)`) e con una banale (`x -> x + 1`): quando `mapMT` conviene?
Suggerimento: ogni compito scrive solo le proprie righe della matrice risultato; non c'è stato condiviso da proteggere, e il risultato è completo quando `invokeAll` ritorna.

Soluzione: progetto `sol-d04-c04`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 4.9: La strada più veloce (⋆⋆)

Scrivere tre `Callable<Long>` che calcolano la stessa cosa, il numero di primi fino a $5 \cdot 10^7$, con tre strategie: per divisioni successive (il nostro `PrimeTester`), con il crivello di Eratostene su un array di `boolean`, e con il crivello sui soli dispari. Consegnarli con `invokeAny` a un pool di tre thread, stampare il risultato e, in ciascun compito, stampare il proprio nome e il tempo quando finisce: chi vince? Che cosa succede agli altri due dopo che il primo ha finito, e perché un crivello non si accorge dell'annullamento?

Soluzione: progetto `sol-d04-c04`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 4.1. I quattro thread del pool non sono daemon e aspettano altri compiti; `main` è finito ma la JVM resta viva. Primo modo: `ex.shutdown()` dopo il `get()` (i compiti in coda, nessuno, vengono completati e i thread terminano). Secondo: `try (ExecutorService ex = Executors.newFixedThreadPool(4)) { ... }`, che spegne e aspetta alla fine del blocco. Un terzo, da conoscere e non usare: una `ThreadFactory` che crea thread daemon, così la JVM esce anche con i compiti a metà, troncandoli.

Risposta 4.2. (a) Niente: l'eccezione è conservata nel `Future` e il programma prosegue. (b) `get()` lancia `ExecutionException`; `getCause()` restituisce la `NullPointerException` originale, con lo stack trace del thread del pool. (c) Con `execute` il thread del pool termina con l'eccezione non catturata, il gestore di default stampa lo stack trace, e il pool crea un thread nuovo al posto di quello morto.

Risposta 4.3. Con un thread: `A, passo 0`, `A, passo 1`, `B, passo 0`, `B, passo 1`, tutti da `pool-1-thread-1`, perché `B` parte solo quando `A` ha finito; l'ordine è garantito. Con due thread i compiti si intrecciano: quasi certamente `A 0`, `B 0`, `B 1`, `A 1` (`B` dorme meno), ma le prime due righe possono invertirsi e l'ordine fra thread diversi non è mai garantito. Un `newSingleThreadExecutor` è proprio lo strumento per quando l'ordine serve.

Risposta 4.4. (a) Puro calcolo: `newFixedThreadPool(n)` con `n` pari ai core disponibili, mille compiti in coda. (b) Attesa: `newCachedThreadPool()` o un pool fisso largo (50–200), perché i thread passano il tempo bloccati; meglio ancora i thread virtuali del capitolo 7. (c) `newSingleThreadExecutor()`: un solo lavoratore serializza le scritture senza che chi scrive aspetti, e l'ordine è quello della coda. (d) `newScheduledThreadPool(1)`, capitolo 7.

Risposta 4.5. `invokeAll` ritorna solo quando *tutti* i compiti sono terminati, bene o male: i `Future` sono già completati e `get()` restituisce subito. Se un compito lancia, `invokeAll` non se ne accorge e ritorna lo stesso; è il `get()` di quel `Future` a lanciare `ExecutionException`, mentre gli altri danno il loro valore. Per questo il ciclo sui `get()` va comunque scritto con il suo `try`: non per l'attesa, ma per gli errori.

Capitolo 5

Quando i thread condividono

Finora i compiti erano indipendenti: ciascuno lavorava sui suoi dati e consegnava un risultato attraverso un campo letto dopo il `join` (capitolo 3) o un `Future` (capitolo 4). Il momento in cui due thread leggono e scrivono *lo stesso oggetto* è quello in cui i programmi iniziano a dare risultati sbagliati, raramente, in modo non riproducibile. Questo capitolo mostra il fenomeno su un contatore, ne spiega le due cause (l'atomicità e la visibilità), introduce lo strumento di base di Java per proteggere uno stato condiviso, il monitor, e le due cose che possono andare storte anche con il monitor: aspettare male, e aspettarsi a vicenda. Alla fine riprenderemo l'esempio guida chiedendoci che cosa succede se i compiti, invece di restituire un numero, sommano in un contatore comune: è la domanda che ogni studente si fa, e la risposta misurata vale più di una regola.

5.1 Il contatore che perde

La classe più semplice che si possa scrivere, e il programma che la mette alla prova: diecimila incrementi, consegnati a un pool di due thread.

```
class Counter {
    int count = 0;
    void inc() { count++; }
}
```

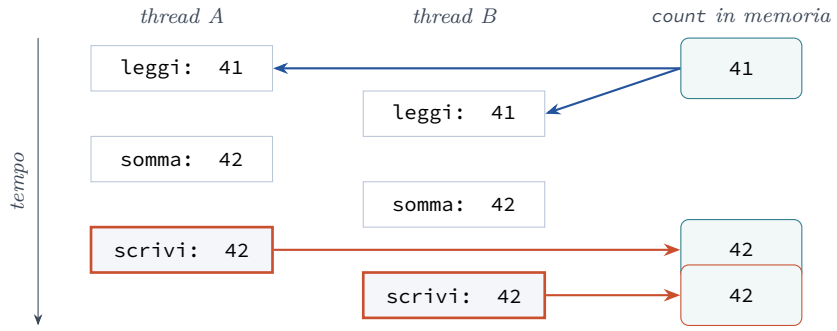
```
Counter counter = new Counter();
try (ExecutorService executor = Executors.newFixedThreadPool(2)) {
    for (int i = 0; i < 10_000; i++)
        executor.submit(counter::inc);
}
System.out.println(counter.count);
```

```
9994
```

Diecimila incrementi, 9994 nel contatore; alla prova successiva 9991, poi 9997, e ogni tanto 10000. Il programma è corretto quasi sempre, che è il modo peggiore di essere sbagliato: nei test passa, in produzione perde. Per capire dove, bisogna guardare `count++` come lo vede la macchina. Sembra un'istruzione ed è tre: leggi `count`, somma uno, scrivi `count`. Lo dice il bytecode:

```
$ javap -c Counter
  void inc();
       0: aload_0
       1: dup
       2: getfield      #7     // leggi count
       5: iconst_1
       6: iadd                 // somma 1
       7: putfield      #7     // scrivi count
      10: return
```

Fra la lettura e la scrittura lo scheduler può sospendere il thread, o un altro core può eseguire le stesse tre istruzioni nello stesso istante (capitolo 2). La figura mostra l'intreccio che perde: `A` e `B` leggono entrambi 41, entrambi sommano, entrambi scrivono 42. Due incrementi, uno solo registrato.



due incrementi, uno perso: B non ha visto la scrittura di A

Definizione 5.1: Conflitto e sezione critica

Si ha un *conflitto* (*race condition*) quando il risultato di un programma dipende dall'ordine, non controllato, in cui più thread accedono a uno stato condiviso. Il tratto di codice che accede allo stato condiviso e che deve essere eseguito da un thread alla volta per restare corretto è una *sezione critica*.

La soluzione, in tutte le sue forme, è sempre la stessa idea: rendere *atomica* la sezione critica, cioè far sì che nessun altro thread la veda a metà. Nel nostro caso: che nessuno legga `count` fra la lettura e la scrittura di un altro.

5.2 Visibilità: non è solo questione di atomicità

C'è un secondo problema, più sottile, che non riguarda le operazioni a metà ma le scritture che *non si vedono*. Ogni core ha le sue cache, e il compilatore JIT ottimizza come se il thread fosse solo: una variabile letta in un ciclo può essere tenuta in un registro. Questo programma, su un portatile qualsiasi, non termina:

```
class Lavoro {
    boolean fermati = false;

    void esegui() {
        long k = 0;
        while (!fermati) k++;              // la JIT legge fermati una volta sola
        System.out.println("fermato dopo " + k);
    }
}
```

```
Lavoro lavoro = new Lavoro();
new Thread(lavoro::esegui).start();
pausa(500);
lavoro.fermati = true;                     // main scrive, l'altro non vede
```

Il thread di lavoro non legge mai più `fermati` dalla memoria: nulla gli dice che un altro thread possa cambiarla, e la JIT, legittimamente, ha spostato la lettura fuori dal ciclo. La parola `volatile` è esattamente quel segnale: una variabile `volatile` viene sempre letta e scritta in memoria, mai tenuta in un registro, e una scrittura è visibile a tutte le letture successive di qualsiasi thread. Con `volatile boolean fermati` il ciclo termina.

Attenzione 5.1: `volatile` non rende atomico niente

`volatile int count` con `count++` perde incrementi esattamente come prima, perché leggi-somma-scrivi restano tre operazioni: `volatile` garantisce che ogni lettura veda l'ultima scrittura, non che fra la lettura e la scrittura non si infili nessuno. La regola pratica: `volatile` per una *bandiera* che un thread scrive e altri leggono; per tutto ciò che si legge e si modifica insieme servono gli strumenti che seguono.

Osservazione 5.1

Il *Java Memory Model* descrive con precisione quando una scrittura di un thread è garantita visibile a un altro, tramite la relazione *happens-before*. I casi che servono nella pratica, e che abbiamo già usato senza nominarli: tutto ciò che un thread fa prima di `start()` è visibile al thread avviato; tutto ciò che un thread fa prima di terminare è visibile a chi fa `join()` o `get()` su di lui (è la garanzia su cui si reggeva `PrimeCounter`); l'uscita da un blocco `synchronized` è visibile a chi entra poi nello stesso blocco; una scrittura `volatile` è visibile a chi legge dopo; `submit` su un executor è visibile al compito. Fuori da queste garanzie non c'è nessun ordine su cui contare, e "ha sempre funzionato" non è una garanzia.

5.3 `synchronized` e il monitor

Una parola risolve il contatore:

```
class Counter {
    int count = 0;
    synchronized void inc() { count++; }
}
```

```
10000
```

Ogni oggetto Java possiede un *monitor* (o *lock intrinseco*): una serratura che un solo thread alla volta può tenere. Un metodo `synchronized` acquisisce il monitor dell'oggetto su cui è chiamato (`this`) all'ingresso e lo rilascia all'uscita, anche per eccezione; un thread che trova il monitor occupato passa in `BLOCKED` (capitolo 2) e aspetta che si liberi. Le tre istruzioni di `count++` sono ora eseguite da un thread alla volta: la sezione critica è atomica. E l'uscita dal monitor rende visibile la scrittura a chi entrerà dopo: anche la visibilità è sistemata, senza `volatile`.

La forma a blocco permette di scegliere l'oggetto e di restringere la sezione critica al minimo:

```
void inc() {
    long nPrimi = PrimeTester.count(1, 1_000_000);   // calcolo lungo, fuori
    synchronized (this) {
        count += nPrimi;                               // accesso condiviso,
    dentro
    }
}
```

Tre regole discendono dal meccanismo, e valgono per ogni lock della dispensa.

- **Si protegge lo stato, non il codice.** Il monitor è dell'oggetto: due thread che chiamano `inc()` su due `Counter` diversi non si bloccano a vicenda, e due metodi diversi `synchronized` sullo stesso oggetto sì. *Tutti* gli accessi a un dato condiviso, anche le sole letture, devono passare dallo stesso monitor: una lettura non sincronizzata può vedere un valore vecchio (visibilità) o a metà (un `long` scritto in due pezzi su hardware a 32 bit).
- **Il monitor è rientrante.** Un thread che tiene il monitor può entrare in altri metodi `synchronized` dello stesso oggetto, anche ricorsivamente, senza bloccarsi da solo: il monitor conta quante volte è stato preso e lo rilascia all'ultima uscita. È ciò che permette a un metodo `synchronized` di chiamarne un altro dello stesso oggetto senza pensarci.
- **La sezione critica è corta.** Dentro il monitor ci sta ciò che tocca lo stato condiviso; il calcolo che non lo tocca sta fuori. Un `synchronized` attorno a un metodo lungo trasforma sedici thread in uno, con in più i costi del lock: lo misureremo nella sezione 5.7.

Un metodo `static synchronized` usa il monitor dell'oggetto `Classe.class`: protegge i campi statici, ed è indipendente dai monitor delle istanze. Sincronizzare su `this` ha un difetto: chiunque abbia un riferimento all'oggetto può fare `synchronized (counter)` e interferire, per esempio tenendo il monitor a lungo e bloccando tutti gli `inc()`. Una classe che espone il proprio stato condiviso a molti fa meglio a usare un lock privato, che nessuno fuori può toccare:

```
private final Object lock = new Object();

void inc() {
    synchronized (lock) { count++; }
}
```

5.4 Coordinarsi: `wait` e `notifyAll`

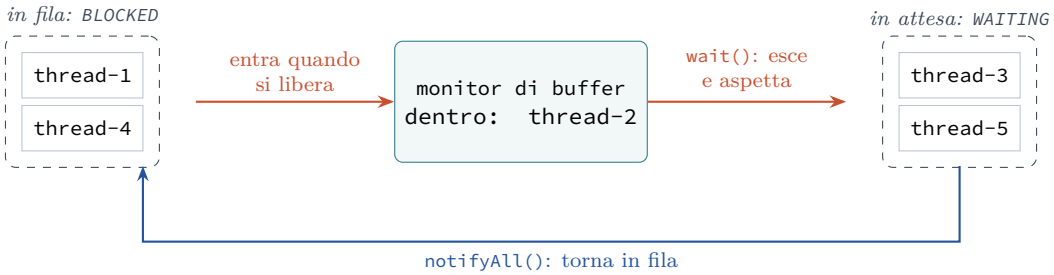
Il monitor non serve solo a escludere: serve anche ad *aspettare* che lo stato diventi quello giusto. Il caso classico è il produttore e il consumatore, la coda dell'immagine guida scritta a mano: un produttore mette elementi in un buffer di capienza fissa, un consumatore li prende; il produttore deve aspettare quando il buffer è pieno, il consumatore quando è vuoto. Aspettare *tenendo* il monitor sarebbe un controsenso, perché l'altro non potrebbe mai entrare a cambiare lo stato. Servono due metodi che ogni oggetto ha, proprio perché ogni oggetto ha un monitor.

```
class Buffer {
    private final Deque<String> coda = new ArrayDeque<>();
    private final int capienza;
    Buffer(int capienza) { this.capienza = capienza; }

    synchronized void metti(String s) throws InterruptedException {
        while (coda.size() == capienza) wait();    // pieno: rilascia e dormi
        coda.addLast(s);
        notifyAll();                               // forse qualcuno aspettava
    }

    synchronized String prendi() throws InterruptedException {
        while (coda.isEmpty()) wait();
        String s = coda.removeFirst();
        notifyAll();
        return s;
    }
}
```

`wait()` si chiama tenendo il monitor: lo *rilascia*, mette il thread in `WAITING`, e quando il thread viene svegliato lo riacquisisce prima di ritornare. `notifyAll()` sveglia tutti i thread in attesa su quel monitor (`notify()` ne sveglia uno, a caso: quasi sempre è la scelta sbagliata, perché potrebbe svegliare un produttore quando serviva un consumatore). La figura mette insieme i due stati del capitolo 2: chi vuole entrare e trova occupato è `BLOCKED`; chi è entrato e ha chiamato `wait()` è `WAITING`, fuori dal monitor, finché un `notifyAll` non lo rimette in fila.



un solo thread dentro; gli altri o bussano o aspettano di essere chiamati

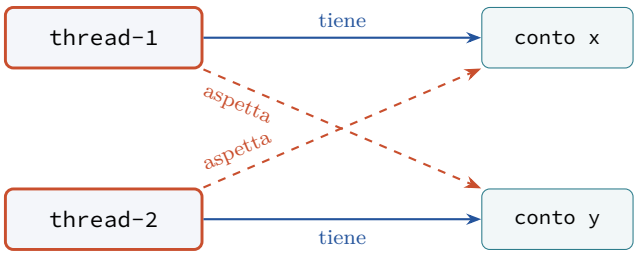
Il `while` attorno a `wait()` non è un `if`, per due ragioni. Fra la sveglia e il riacquisto del monitor un altro thread può aver cambiato di nuovo lo stato: due consumatori svegliati dallo stesso `notifyAll` troverebbero, il secondo, la coda di nuovo vuota. E la JVM ammette *risvegli spuri*, senza nessuna `notify`. Si ricontrolla sempre la condizione, e se non vale si torna a dormire. Nella pratica questo schema è già scritto, e meglio, nelle code bloccanti del capitolo 6; va conosciuto perché è ciò che sta sotto, e perché prima o poi serve una condizione che la libreria non offre.

5.5 Deadlock

Due conti e un trasferimento che blocca entrambi, per non lasciare mai la somma a metà:

```
void trasferisci(Conto da, Conto a, int importo) {
    synchronized (da) {
        synchronized (a) {
            da.preleva(importo);
            a.versa(importo);
        }
    }
}
```

Due thread chiamano nello stesso istante `trasferisci(x, y, 10)` e `trasferisci(y, x, 20)`. Il primo prende il monitor di `x` e aspetta quello di `y`; il secondo ha preso `y` e aspetta `x`. Nessuno dei due cederà mai: è un *deadlock*, e il programma resta fermo senza errori, senza eccezioni, senza consumare processore. `jstack` (capitolo 2) lo riconosce e lo stampa: "Found one Java-level deadlock".



un ciclo di attese: nessuno rilascia, nessuno avanza

Il deadlock si verifica quando quattro condizioni sono vere insieme: risorse esclusive (un monitor lo tiene uno solo), un thread che ne tiene una mentre ne chiede un'altra, nessuna prelazione (un monitor non si può togliere a chi lo ha), e un *ciclo* di attese. Basta negarne una. La soluzione più semplice rompe l'ultima: tutti i thread acquisiscono i lock *nello stesso ordine*, per esempio quello di un identificatore dei conti, così chi vuole entrambi prende sempre prima il minore, e un ciclo non può formarsi.

```
void trasferisci(Conto da, Conto a, int importo) {
    Conto primo   = da.id() < a.id() ? da : a;
    Conto secondo = da.id() < a.id() ? a : da;
    synchronized (primo) {
        synchronized (secondo) {
            da.preleva(importo);
            a.versa(importo);
        }
    }
}
```

L'altra via è non aspettare per sempre: i lock espliciti del capitolo 6 hanno `tryLock(t, unit)`, che rinuncia dopo un tempo e permette di rilasciare quello che si aveva e riprovare.

Osservazione 5.2

Due parenti del deadlock. *Starvation*: un thread che non ottiene mai la risorsa perché altri, più fortunati o più prioritari, la prendono sempre prima. *Livelock*: due thread che, per evitarsi, cambiano stato in continuazione senza avanzare, come due persone in un corridoio che si spostano sempre dallo stesso lato. Entrambi si vedono come un programma che gira ma non conclude, e in entrambi `jstack` non trova nessun ciclo.

5.6 Thread safety per costruzione

Il modo migliore di proteggere uno stato condiviso è non averne. Tre strategie, dalla più semplice, che insieme coprono la maggior parte dei programmi ben scritti.

- **Immutabilità.** Un oggetto che non cambia dopo la costruzione può essere letto da qualsiasi numero di thread senza nessun lock: `String`, gli involucri dei primitivi, `List.of(...)`, i `record` con campi immutabili. Un "cambiamento" produce un oggetto nuovo. È la ragione per cui un `record` è il modo giusto di passare un risultato fra thread.
- **Confinamento.** Un dato usato da un solo thread non è condiviso: le variabili locali lo sono per natura (ogni thread ha il suo stack, capitolo 2), e un oggetto creato dal compito e mai passato ad altri lo è di fatto. È il modello di tutti i conteggi dei primi visti finora: ogni compito conta nel suo campo o nella sua variabile, e la condivisione avviene una volta sola, attraverso un `join` o un `Future`, in un punto in cui l'ordine è garantito.
- **Oggetti thread-safe dichiarati.** Le classi della libreria che lo sono lo dicono nella documentazione; quelle che non lo dicono, non lo sono. Non lo sono, per esempio, `ArrayList`, `HashMap` e `StringBuilder`; lo sono `ConcurrentHashMap`, `AtomicInteger` e le code del package `java.util.concurrent`, che sono l'argomento del capitolo 6.

5.7 L'esempio guida: sommare in un contatore condiviso

Il programma del capitolo 4 restituiva da ogni compito un `Long`, e `main` sommava i `Future`. Molti, alla prima stesura, fanno un'altra cosa: un contatore condiviso, e ogni compito lo incrementa a ogni primo trovato. Sembra più semplice, e non richiede i `Future`. Vediamo che cosa succede, nelle stesse condizioni del capitolo 4 (sedici thread, 64 compiti, primi fino a 10^8), con tre versioni del compito.

```
// 1. un Counter senza synchronized, incrementato a ogni primo
executor.submit(() -> {
    for (long k = da; k <= a; k++)
        if (PrimeTester.isPrime(k)) counter.inc();      // perde
});
```

```
// 2. lo stesso, con inc() synchronized
executor.submit(() -> {
    for (long k = da; k <= a; k++)
        if (PrimeTester.isPrime(k)) counter.inc();      // corretto, ma 5,7
    milioni di lock
});
```

```
// 3. conta in locale, aggiungi una volta sola sotto lock
executor.submit(() -> {
    long n = 0;
    for (long k = da; k <= a; k++)
        if (PrimeTester.isPrime(k)) n++;                // variabile locale:
    confinata
    counter.add(n);                                      // synchronized, 64
    volte in tutto
});
```

versione	risultato	tempo
`Future` (capitolo 4)	5 761 455	1,37 s
1. contatore senza lock	4 435 156	1,43 s
2. `synchronized` a ogni primo	5 761 455	1,83 s
3. locale, poi `add` una volta	5 761 455	1,34 s

La versione 1 perde un milione e trecentomila incrementi su cinque milioni e mezzo: con sedici thread che martellano la stessa variabile, l'intreccio della sezione 5.1 non è un caso raro ma la norma. La versione 2 è corretta e costa mezzo secondo in più: cinque milioni e settecentomila acquisizioni del monitor, molte delle quali contese, con i thread che passano per `BLOCKED` e tornano. La versione 3 è corretta e veloce quanto quella con i `Future`: la sezione critica è corta (una somma) e rara (64 volte in tutto), e il lavoro vero avviene su una variabile locale, confinata nel thread. È la terza regola della sezione 5.3 misurata, ed è il modo di ragionare da portarsi dietro: condividere il meno possibile, il più tardi possibile, sotto un lock che si tiene per il tempo di una somma.

Esercizi

Esercizio 5.1: L'intreccio che perde (⋆)

Con `count` uguale a 41 e due thread che eseguono `count++`, scrivere un intreccio delle sei istruzioni (leggi, somma, scrivi per ciascuno) che porta a 43 e uno che porta a 42. Con tre thread, qual è il valore minimo che si può ottenere?

Esercizio 5.2: `volatile` e `count++` (⋆)

Un collega dichiara `volatile int count` e sostiene che ora `count++` è sicuro "perché ogni thread vede sempre l'ultimo valore". Che cosa vede davvero il thread `B` nell'intreccio della sezione 5.1, e perché l'incremento si perde lo stesso?

Esercizio 5.3: Chi blocca chi (⋆⋆)

Una classe ha due metodi `synchronized`, `a()` e `b()`, e un metodo `static synchronized c()`. Dire, per ciascuna coppia di chiamate da due thread diversi, se possono eseguire insieme: `x.a()` e `x.b()`; `x.a()` e `y.a()`; `x.a()` e `c()`; `x.a()` e, dentro `a()`, una chiamata a `this.b()` dallo stesso thread.

Esercizio 5.4: `if` al posto di `while` (⋆⋆)

Nel `Buffer` della sezione 5.4 si sostituisce `while (coda.isEmpty()) wait();` con `if`. Descrivere, con due consumatori e un produttore che mette un solo elemento, la sequenza che porta un consumatore a chiamare `removeFirst()` su una coda vuota.

Esercizio 5.5: Dov'è il ciclo (⋆⋆)

Tre thread: `T1` prende `A` poi `B`; `T2` prende `B` poi `C`; `T3` prende `C` poi `A`. C'è un deadlock possibile? Disegnare il ciclo. Poi `T3` viene cambiato in "prende `A` poi `C`": c'è ancora?

Esercizio 5.6: Immutabile davvero (⋆)

`record Risultato(long nPrimi, List<Long> trovati)` è thread-safe? Un thread lo crea e lo passa a un altro attraverso un `Future`; il secondo chiama `trovati().add(7)`. Che cosa può andare storto, e come si costruisce il record perché non possa?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 5.7: `Counter` in tre modi (⋆)

Scrivere tre versioni di `Counter`, ciascuna provata con diecimila `inc()` da un pool di quattro thread: con il metodo `synchronized`; con un blocco su un lock privato; e senza condivisione, cioè un contatore per thread e la somma alla fine (il confinamento della sezione 5.6). Ogni versione deve avere anche un `get()` corretto. Dire nel commento in quale situazione si sceglierebbe ciascuna.

Soluzione: progetto `sol-d04-c05`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 5.2: Anche la lettura

Un `get()` senza `synchronized` compila e quasi sempre restituisce il valore giusto; ma non è garantito che veda l'ultima scrittura, e per un `long` su hardware a 32 bit potrebbe vederne metà. Tutti gli accessi allo stato, non solo le scritture.

Esercizio 5.8: Il deadlock dal vivo (⋆⋆)

Scrivere una classe `Conto` con `id`, `saldo`, `preleva` e `versa`, e la prima versione di `trasferisci` della sezione 5.5. Avviare due thread che trasferiscono in continuazione da `x` a `y` e da `y` a `x`, osservare il blocco con `jstack` (copiare nel commento la riga "Found one Java-level deadlock" e le due "waiting to lock"). Poi risolverlo con l'ordine di acquisizione e verificare, con un terzo thread che legge ogni secondo, che la somma dei saldi resti costante.

Soluzione: progetto `sol-d04-c05`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 5.9: Produttore e consumatore (⋆⋆)

Un produttore genera i numeri da 1 a 1000 e un consumatore ne stampa la somma finale, attraverso un `Buffer<Integer>` di capienza 10 scritto con `wait` e `notifyAll` come nella sezione 5.4, reso generico. Il consumatore deve sapere quando non arriverà più niente: progettare il segnale di fine e spiegarlo nel commento. Poi provare con due consumatori: che cosa cambia nel segnale di fine?
Suggerimento: un valore sentinella *che non può essere un dato; con più consumatori, uno per ciascuno, oppure chi lo trova lo rimette in coda.*

Soluzione: progetto `sol-d04-c05`, package `it.unibs.pajc.redolfi.e09`.

Esercizio 5.10: Un registro a più scrittori (⋆⋆⋆)

Scrivere una classe `Registro` in cui più thread aggiungono eventi (`record Evento(long istante, String thread, String testo)`) e un thread lettore chiede ogni tanto un'*istantanea*: la lista degli eventi in ordine di inserimento, come era in quel momento, che non cambia anche se gli scrittori continuano. Proteggere lo stato con un lock privato, decidere che cosa è dentro la sezione critica e che cosa fuori, e restituire al lettore qualcosa che non possa essere modificato né dal lettore né, dopo, dagli scrittori. Misurare con otto scrittori e un lettore.
Suggerimento: l'istantanea è una copia, fatta sotto lock, resa immutabile con `List.copyOf`; la copia costa, e si paga solo quando il lettore la chiede.

Soluzione: progetto `sol-d04-c05`, package `it.unibs.pajc.redolfi.e10`.

Risposte

Risposta 5.1. Porta a 43: `A` legge 41, somma, scrive 42; poi `B` legge 42, somma, scrive 43 (le sue tre istruzioni dopo quelle di `A`, o viceversa). Porta a 42: `A` legge 41, `B` legge 41, poi entrambi sommano e scrivono 42, in qualsiasi ordine. Con tre thread il minimo è 42: tutti e tre leggono 41 prima che qualcuno scriva, e tutti scrivono 42. In generale con n incrementi concorrenti il risultato può essere qualsiasi valore fra 1 e n in più.

Risposta 5.2. `B` vede davvero l'ultimo valore *scritto*: ma quando `B` legge, `A` ha letto 41 e non ha ancora scritto 42, quindi l'ultimo valore scritto è 41, e `B` legge 41 correttamente. `volatile` ha fatto il suo dovere, e l'incremento si perde lo stesso, perché il problema non è una lettura vecchia ma due letture *prima* di due scritture. L'atomicità di leggi-somma-scrivi la dà solo un lock, o un'operazione atomica del processore (capitolo 6).

Risposta 5.3. `x.a()` e `x.b()`: no, stesso monitor (quello di `x`). `x.a()` e `y.a()`: sì, monitor diversi. `x.a()` e `c()`: sì, `c()` usa il monitor di `Classe.class`, indipendente da quello di `x`. `a()` che chiama `this.b()`: sì, lo stesso thread rientra nel monitor che già tiene, senza bloccarsi; un altro thread che volesse `x.b()` aspetterebbe l'uscita da entrambi.

Risposta 5.4. I consumatori `C1` e `C2` trovano la coda vuota e chiamano `wait()` (rilasciando il monitor). Il produttore mette un elemento e chiama `notifyAll()`: entrambi si svegliano e si mettono in fila per il monitor. `C1` entra, con l'`if` non ricontrolla, fa `removeFirst()` e prende l'unico elemento, esce. `C2` entra, con l'`if` non ricontrolla, fa `removeFirst()` su una coda vuota: `NoSuchElementException`. Con il `while`, `C2` ricontrolla, trova la coda vuota e torna a dormire.

Risposta 5.5. Sì: `T1` tiene `A` e aspetta `B`, `T2` tiene `B` e aspetta `C`, `T3` tiene `C` e aspetta `A`; il ciclo è $T1 \to T2 \to T3 \to T1$ e si chiude se i tre prendono il primo lock prima che qualcuno prenda il secondo. Con `T3` che prende `A` poi `C` l'ordine di tutti rispetta $A < B < C$ (`T1`: A, B; `T2`: B, C; `T3`: A, C), e un ciclo richiederebbe che qualcuno tenga un lock "maggiore" aspettandone uno "minore": impossibile. Nessun deadlock.

Risposta 5.6. Il record è immutabile nei suoi *campi*: i due riferimenti non cambiano. Ma `trovati` è un riferimento a una lista, e se la lista è una `ArrayList` il secondo thread può modificarla, mentre magari il primo la sta ancora leggendo: lo stato condiviso è rientrato dalla finestra. Si costruisce il record con una copia immutabile, `List.copyOf(trovati)` nel costruttore compatto, così `add` lancia `UnsupportedOperationException` e nessun thread può cambiare ciò che l'altro vede. Immutabile vuol dire fino in fondo, non solo al primo livello.

Capitolo 6

Gli strumenti di `java.util.concurrent`

Il capitolo 5 ha dato lo strumento di base, il monitor, e le regole per usarlo. Scrivere ogni volta un lock, un `while` con `wait` e un `notifyAll` è però faticoso e facile da sbagliare, e per i casi comuni (un contatore, una coda, una mappa condivisa, una risorsa a posti limitati) la libreria ha soluzioni già pronte, più veloci e più difficili da usare male. Questo capitolo le passa in rassegna con un criterio: per ciascuna, qual è il problema che risolve meglio di `synchronized`. Alla fine una tabella per scegliere, e l'esempio guida riscritto con due di questi strumenti.

6.1 Lock espliciti: `ReentrantLock`

Il primo strumento è il monitor stesso, in forma di oggetto. `ReentrantLock` fa ciò che fa `synchronized`, con il rilascio a carico del programmatore:

```
private final ReentrantLock lock = new ReentrantLock();
private int count = 0;

void inc() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();          // sempre, anche in caso di eccezione
    }
}
```

Il `finally` è obbligatorio: un'eccezione fra `lock()` e `unlock()` lascerebbe il lock preso per sempre, e con `synchronized` questo non può succedere. Perché allora usarlo? Per ciò che il monitor non sa fare. `tryLock()` prova senza bloccarsi e dice se ha avuto il lock; `tryLock(t, unit)` aspetta al massimo un tempo, ed è la seconda via d'uscita dal deadlock del capitolo 5: chi non ottiene il secondo lock entro il tempo rilascia il primo e riprova. `lockInterruptibly()` si lascia interrompere mentre aspetta, cosa che un thread `BLOCKED` su un monitor non fa. Il costruttore `new ReentrantLock(true)` rende l'ordine di accesso equo: chi aspetta da più tempo entra prima, contro la starvation. E `isHeldByCurrentThread()`, `isLocked()` aiutano a diagnosticare.

```
ReentrantLock lock = new ReentrantLock();
executor.submit(() -> {
    lock.lock();
    try { pausa(1000); } finally { lock.unlock(); }
});
executor.submit(() -> {
    System.out.println("occupato: " + lock.isLocked());             // true
    System.out.println("da me: " + lock.isHeldByCurrentThread());  // false
    System.out.println("preso: " + lock.tryLock());                 // false
});
```

Si usa `ReentrantLock` quando serve una di queste cose; altrimenti `synchronized` è più corto, altrettanto veloce, e non si può dimenticare di rilasciarlo.

6.2 Lettori e scrittori: `ReadWriteLock`

Molti thread possono *leggere* una struttura insieme senza danno; il problema nasce solo quando qualcuno *scrive*. Un monitor non lo sa, e fa aspettare anche i lettori fra loro. `ReentrantReadWriteLock` ha due lock coordinati: quello di lettura, che più thread possono tenere insieme, e quello di scrittura, esclusivo e incompatibile con le letture.

```
Map<String, String> mappa = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {
    lock.writeLock().lock();
    try {
        pausa(1000);
        mappa.put("alfa", "primo elemento");
    } finally { lock.writeLock().unlock(); }
});

Runnable lettura = () -> {
    lock.readLock().lock();
    try {
        System.out.println(mappa.get("alfa"));
        pausa(1000);
    } finally { lock.readLock().unlock(); }
};
executor.submit(lettura);
executor.submit(lettura);          // le due letture girano insieme, dopo la
    scrittura
```

Conviene quando le letture sono molte e lunghe e le scritture rare: una cache, una configurazione, un indice. Con letture brevissime il costo del lock di lettura supera il guadagno, e un monitor fa meglio. `StampedLock` JAVA 8 è una variante più veloce con una modalità in più, la *lettura ottimistica*: si legge senza prendere nessun lock, e `validate(stamp)` dice a posteriori se nel frattempo c'è stata una scrittura (nel qual caso si rilegge con il lock vero). Non è rientrante e richiede di passare lo `stamp` restituito da ogni acquisizione al rilascio: va riservato ai casi in cui le misure dicono che `ReadWriteLock` è il collo di bottiglia.

6.3 Risorse a posti limitati: `Semaphore`

Un lock ammette un thread; un semaforo ne ammette `n`. Nasce con un numero di *permessi*; `acquire()` ne prende uno o aspetta, `release()` lo restituisce. Il caso tipico è limitare l'accesso a una risorsa che regge solo un certo numero di utenti simultanei: connessioni a un database, licenze, memoria per immagini grandi.

```
Semaphore semaforo = new Semaphore(5);

Runnable compito = () -> {
    boolean permesso = false;
    try {
        permesso = semaforo.tryAcquire(1, TimeUnit.SECONDS);
        if (permesso) {
            System.out.println("dentro: " + Thread.currentThread().getName());
            pausa(5000);
        } else {
            System.out.println("rinuncio: " + Thread.currentThread().getName());
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } finally {
        if (permesso) semaforo.release();
    }
};
for (int i = 0; i < 10; i++) executor.submit(compito);
```

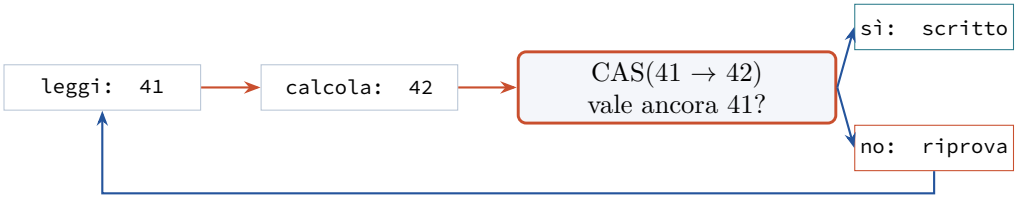
Dieci compiti, cinque permessi: cinque entrano, cinque aspettano un secondo e rinunciano.

Attenzione 6.1: `release` solo se si è acquisito

Il `finally` rilascia *solo* se il permesso era stato ottenuto. Un semaforo non sa chi ha preso che cosa: rilasciare senza aver acquisito *aumenta* i permessi, e da quel momento la risorsa ammette un utente in più di quanti ne regge. Un `release()` nel `finally` senza la guardia è l'errore classico, e non dà nessun segnale.

6.4 Variabili atomiche

Per un contatore, una bandiera, un riferimento da aggiornare in modo atomico, il lock è sproporzionato: un thread che trova il monitor occupato passa per `BLOCKED`, cioè per il sistema operativo, per proteggere una somma. Le classi di `java.util.concurrent.atomic` (`AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference<V>`) offrono operazioni atomiche senza lock, realizzate con un'istruzione del processore, il *compare-and-swap*: "scrivi il nuovo valore solo se quello attuale è ancora quello che ho letto, altrimenti dimmelo". Se il confronto fallisce la libreria rilegge e riprova; non c'è mai un thread bloccato.



compare-and-swap: una sola istruzione, nessun lock, nessun thread bloccato

```
AtomicInteger count = new AtomicInteger(0);
for (int i = 0; i < 10_000; i++)
    executor.submit(count::incrementAndGet);
...
System.out.println(count.get());                    // 10000
```

```
count.updateAndGet(n -> n + 2);                     // qualsiasi funzione,
    atomica
count.accumulateAndGet(5, Math::max);
boolean fatto = count.compareAndSet(42, 0);         // se vale 42, azzera
```

`updateAndGet` riceve una lambda che può essere eseguita *più volte*, se nel frattempo un altro thread ha cambiato il valore e il compare-and-swap è fallito: deve essere pura, senza effetti. Quando tanti thread incrementano *lo stesso* contatore in continuazione (il numero di richieste di un server) il compare-and-swap fallisce spesso e si riprova spesso: per questo caso esistono `LongAdder` e `LongAccumulator` JAVA 8, che tengono una cella per thread e sommano solo quando si chiede `sum()`. Costano più memoria, e la lettura è meno precisa durante gli aggiornamenti; in cambio scalano.

Le prestazioni dipendono dall'hardware più di quanto si creda, e la sola regola seria è misurare. Dieci milioni di incrementi da 16 thread sul portatile del capitolo 4: `synchronized` 74 ms, `AtomicInteger` 391 ms, `LongAdder` 66 ms. Il compare-and-swap, sotto contesa altissima, ha perso dal lock; su un altro processore il risultato può rovesciarsi. L'atomica resta la scelta giusta per il caso comune, la contesa moderata, dove non blocca nessuno e non costa una chiamata al sistema.

6.5 Collezioni concorrenti

Le collezioni di `java.util` non sono thread-safe (capitolo 5): due `add` insieme su una `ArrayList` possono perdere un elemento o corrompere la struttura. La scorciatoia storica, `Collections.synchronizedList`, avvolge ogni metodo in un `synchronized` e ha un limite che sorprende: l'iterazione non è protetta, perché è fatta di molte chiamate separate, e un `for` su una lista che un altro thread sta modificando finisce con `ConcurrentModificationException`. Le classi del package `java.util.concurrent` sono invece progettate per la concorrenza dall'interno.

6.5.1 `ConcurrentHashMap`

```
ConcurrentMap<String, Integer> conteggi = new ConcurrentHashMap<>();
conteggi.put("alfa", 1);
conteggi.putIfAbsent("alfa", 99);                    // resta 1
conteggi.merge("alfa", 1, Integer::sum);             // 2, atomico
conteggi.compute("beta", (k, v) -> v == null ? 1 : v + 1);
conteggi.forEach((k, v) -> System.out.println(k + " = " + v));
```

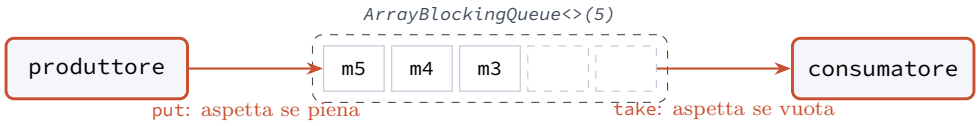
`ConcurrentHashMap` suddivide la mappa in segmenti con lock indipendenti, così scritture su chiavi diverse non si bloccano a vicenda, e le letture non prendono lock. I metodi `merge`, `compute`, `computeIfAbsent` sono atomici: la coppia "leggi, poi scrivi" che con una `HashMap` e un lock esterno sarebbe una sezione critica qui è una chiamata sola. È lo strumento per ogni conteggio o raggruppamento fatto da più thread: parole di più file, richieste per utente, primi per intervallo.

6.5.2 `CopyOnWriteArrayList`

Una lista che a ogni modifica *copia* l'array interno: le letture e le iterazioni non prendono mai lock e vedono un'istantanea coerente, le scritture costano una copia. È la struttura giusta per un elenco di *listener*: pochi cambiamenti, molte scansioni, e nessun rischio che un listener che si rimuove durante la notifica rompa il `for`. È anche l'istantanea del registro del capitolo 5, fatta dalla libreria.

6.5.3 `BlockingQueue`

È il produttore e consumatore del capitolo 5 già scritto, e la coda dell'immagine guida in forma di classe: `put` aspetta se la coda è piena, `take` se è vuota, con le varianti con timeout `offer` e `poll`. `ArrayBlockingQueue` ha capienza fissa, `LinkedBlockingQueue` opzionale, `PriorityBlockingQueue` estrae per priorità.



la coda fa da cuscinetto: chi produce e chi consuma non si vedono mai

```
BlockingQueue<String> coda = new ArrayBlockingQueue<>(10);

executor.submit(() -> {                              // produttore
    for (int i = 0; i < 100; i++) coda.put("msg " + i);
    coda.put("FINE");
    return null;
});
executor.submit(() -> {                              // consumatore
    String s;
    while (!(s = coda.take()).equals("FINE"))
        System.out.println(s);
    return null;
});
```

Le due lambda terminano con `return null`: così sono `Callable` e possono lanciare l'eccezione di `put` e `take` senza un `try`, un piccolo trucco che vale la pena conoscere. È lo schema con cui un thread di rete consegna i messaggi ricevuti a chi li elabora, e con cui l'interfaccia grafica, nel capitolo 8, riceve il lavoro da fare.

6.6 Quale strumento

Il problema	**Lo strumento**
una bandiera scritta da uno, letta da altri	`volatile boolean` o `AtomicBoolean`
un contatore, un riferimento da aggiornare	`AtomicInteger`/`Long`/`Reference`; `LongAdder` se la contesa è alta
più campi da aggiornare insieme	`synchronized` su un lock privato; `ReentrantLock` se servono timeout o equità
molte letture, poche scritture	`ReadWriteLock`; `CopyOnWriteArrayList` per gli elenchi
una risorsa a `n` posti	`Semaphore`
passare dati fra thread	`BlockingQueue`
una mappa condivisa	`ConcurrentHashMap` con `merge`/`compute`
aspettare che una condizione diventi vera	`wait`/`notifyAll`, `Condition` di `ReentrantLock`, `CountDownLatch`
nessuno stato condiviso	la scelta migliore: immutabilità e confinamento

6.7 L'esempio guida: `LongAdder` e una mappa dei primi

Il capitolo 5 si è chiuso con un contatore `synchronized` aggiornato una volta per compito. Con gli strumenti di questo capitolo si fa la stessa cosa senza scrivere un lock, e in più si raccoglie un'informazione che i `Future` non davano: quanti primi ci sono in ogni decina di milioni.

```
LongAdder totale = new LongAdder();
ConcurrentMap<Long, Long> perDecina = new ConcurrentHashMap<>();

try (ExecutorService executor = Executors.newFixedThreadPool(nThread)) {
    for (int i = 0; i < nTask; i++) {
        long da = i * perTask + 1, a = (i == nTask - 1) ? max : (i + 1) * perTask;
        executor.execute(() -> {
            long n = 0;
            for (long k = da; k <= a; k++)
                if (PrimeTester.isPrime(k)) n++;
            totale.add(n);                                    // una cella per
    thread
            perDecina.merge(da / 10_000_000, n, Long::sum);     // atomico per
    chiave
        });
    }
}
System.out.println(totale.sum());
new TreeMap<>(perDecina).forEach((d, n) -> System.out.printf("%2d: %d%n", d, n));
```

```
5761455
 0: 664579
 1: 586081
 2: 563955
 ...
 9: 548835
```

Nessun `Future`, nessun `synchronized`, nessun `get()`: i compiti sono `Runnable` consegnati con `execute`, il `try` con risorse aspetta che abbiano finito, e la visibilità dei risultati è garantita dalla chiusura dell'executor. Il `merge` fa in una chiamata ciò che con una `HashMap` e un lock sarebbero tre righe e una sezione critica. La mappa è ordinata alla fine con un `TreeMap`, perché `ConcurrentHashMap` non ha un ordine. E si vede, di passaggio, che i primi si diradano: 664 579 nella prima decina di milioni, 548 835 nell'ultima.

Esercizi

Esercizio 6.1: Senza `finally` (⋆)

In `inc()` della sezione 6.1 si toglie il `try/finally` e si scrive `lock.lock(); count++; lock.unlock();`. Funziona? Che cosa succede al programma se `count++` fosse un'operazione che può lanciare, e lanciasse una volta? Perché con `synchronized` la domanda non si pone?

Esercizio 6.2: Letture e scritture (⋆)

Con un `ReentrantReadWriteLock` e quattro thread, dire quali coppie possono essere dentro nello stesso istante: due lettori; un lettore e uno scrittore; due scrittori; un lettore che, tenendo il lock di lettura, chiede quello di scrittura (attenzione: la risposta è una brutta notizia).

Esercizio 6.3: La lambda eseguita due volte (⋆⋆)

`count.updateAndGet(n -> { System.out.println("x"); return n + 1; })` da sedici thread che lo chiamano insieme. Il numero di `x` stampate può essere maggiore del numero di chiamate? Perché? Che cosa deve valere per la lambda passata a `updateAndGet`?

Esercizio 6.4: `synchronizedList` e il `for` (⋆⋆)

`List<Integer> l = Collections.synchronizedList(new ArrayList<>());` un thread fa `l.add(i)` in ciclo, un altro `for (int x : l) somma += x;`. Che cosa può succedere al secondo, e perché il `synchronized` dentro `add` e `get` non lo evita? Come si scrive il `for` in modo corretto con la stessa lista, e quale classe lo evita per costruzione?

Esercizio 6.5: `merge` contro leggi-e-scrivi (⋆)

Con una `ConcurrentHashMap<String, Integer>` e due thread che contano la stessa parola, confrontare `m.put(w, m.getOrDefault(w, 0) + 1)` con `m.merge(w, 1, Integer::sum)`: la prima può perdere conteggi anche se la mappa è concorrente? Perché?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 6.6: Le parole di molti file (⋆)

Dati i file di testo di una cartella, contare quante volte compare ogni parola in tutti i file insieme, con un compito per file su un pool e una sola `ConcurrentHashMap<String, Integer>` aggiornata con `merge`. Stampare le dieci parole più frequenti. Poi scrivere la stessa cosa con una `HashMap` e un `synchronized` attorno all'aggiornamento, e confrontare i tempi su una cartella con qualche decina di file.

Soluzione: progetto `sol-d04-c06`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 6.2: Il `merge` è atomico, la lettura del file no

Ogni compito legge il suo file per intero prima di toccare la mappa: la sezione critica è il solo `merge`, e il lavoro (leggere, dividere in parole) sta fuori. Chi mette la lettura dentro il `synchronized` della seconda versione trasforma il pool in un thread solo.

Esercizio 6.7: Un pool di connessioni (⋆⋆)

Simulare un servizio che regge al massimo tre connessioni: una classe `Connessioni` con `apri()` e `chiudi()` che usa un `Semaphore(3)`, e venti compiti che aprono, lavorano per un tempo casuale fra 100 e 500 ms, chiudono. Ogni compito stampa quando entra e quando esce con il numero di permessi disponibili; verificare che non siano mai più di tre dentro. Poi introdurre un timeout con `tryAcquire`: i compiti che aspettano più di 300 ms rinunciano e lo dicono.

Soluzione: progetto `sol-d04-c06`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 6.8: Una cache con lettori e scrittori (⋆⋆)

Scrivere una `Cache<K, V>` con `get(K)` e `put(K, V)` su una `HashMap`, protetta da un `ReentrantReadWriteLock`; quindici thread che fanno solo `get` e uno che fa `put` ogni 10 ms, per tre secondi. Misurare il numero di `get` completati; poi sostituire il `ReadWriteLock` con un monitor e rimisurare; infine con una `ConcurrentHashMap` senza nessun lock. Commentare i tre numeri.

Soluzione: progetto `sol-d04-c06`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 6.9: Una pipeline a tre stadi (⋆⋆⋆)

Tre thread collegati da due `BlockingQueue`: il primo legge le righe di un file di testo e le mette nella prima coda; il secondo prende ogni riga, la trasforma (per esempio in maiuscolo con il numero di riga davanti) e la mette nella seconda; il terzo scrive le righe trasformate su un file di uscita. Progettare il segnale di fine che attraversa i tre stadi, gestire l'interruzione, e misurare con code di capienza 1, 10 e 1000: cambia qualcosa? *Suggerimento: la sentinella passa di stadio in stadio; ogni stadio, quando la riceve, la inoltra e termina.*

Soluzione: progetto `sol-d04-c06`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 6.1. Funziona finché non succede niente. Se `count++` lanciasse, `unlock()` non verrebbe eseguito e il lock resterebbe preso per sempre: ogni altro thread che chiama `inc()` si blocca al `lock()` e il programma si ferma, senza deadlock visibile a `jstack` (un solo thread, morto, tiene il lock). Con `synchronized` il monitor viene rilasciato all'uscita dal blocco *comunque*, anche per eccezione: è il compilatore a mettere il `finally`.

Risposta 6.2. Due lettori: sì. Lettore e scrittore: no. Due scrittori: no. Un lettore che chiede il lock di scrittura tenendo quello di lettura: *deadlock con se stesso*: la scrittura aspetta che tutti i lettori escano, compreso lui, che non uscirà finché non ottiene la scrittura. `ReentrantReadWriteLock` non ammette la promozione da lettura a scrittura; ammette la retrocessione (prendere la lettura tenendo la scrittura, poi rilasciare la scrittura).

Risposta 6.3. Sì: ogni volta che il compare-and-swap fallisce perché un altro thread ha cambiato il valore nel frattempo, la libreria rilegge e *riesegue* la lambda. Con sedici thread in contesa le `x` possono essere molte più di sedici. La lambda deve essere pura: calcolare il nuovo valore dal vecchio e nient'altro, niente stampe, niente contatori esterni, niente I/O.

Risposta 6.4. Il `for` usa un iteratore, che a ogni passo chiama `hasNext` e `next`: ciascuna chiamata è sincronizzata, ma fra una e l'altra il primo thread può fare `add`, e l'iteratore se ne accorge e lancia `ConcurrentModificationException`. Il modo corretto è sincronizzare l'intera iterazione sulla lista stessa, `synchronized (l) { for (...) }`, come dice la documentazione di `synchronizedList`; oppure usare `CopyOnWriteArrayList`, il cui iteratore lavora su un'istantanea e non può essere disturbato.

Risposta 6.5. Sì: `getOrDefault` e `put` sono due chiamate, ciascuna atomica, ma fra le due l'altro thread può fare le sue, e si torna all'intreccio del contatore: entrambi leggono 3, entrambi scrivono 4. Una collezione concorrente rende atomica ogni singola operazione, non una sequenza di operazioni. `merge` è una sola operazione, e la mappa la esegue tenendo il lock del segmento: è per questo che esiste.

Capitolo 7

Il tempo e l'attesa

I capitoli precedenti hanno distribuito un *calcolo* fra i thread. Questo capitolo aggiunge due dimensioni che un programma reale ha sempre: il *tempo* (un compito che deve partire fra tre secondi, o ripetersi ogni minuto) e l'*attesa* (compiti che non calcolano quasi nulla e passano la vita bloccati su una rete o un disco). Per la prima la libreria ha un executor che conosce l'orologio; per la seconda Java 21 ha cambiato la risposta alla domanda "quanti thread posso avere?" con i thread virtuali. In mezzo chiudiamo un conto lasciato aperto nella dispensa sugli stream: che cosa fa davvero `parallel()`, ora che sappiamo che cosa c'è sotto. Alla fine l'esempio guida viene eseguito in tre modi, su un lavoro di calcolo e su uno di attesa, e i numeri dicono quale strumento serve a che cosa.

7.1 Compiti differiti e periodici

`ScheduledExecutorService`, creato con `Executors.newScheduledThreadPool(n)`, è un pool che aggiunge il tempo: un compito può partire dopo un ritardo, oppure ripetersi.

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
Runnable task = () -> System.out.println("eseguito alle " + LocalTime.now());

ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);
pausa(1400);
System.out.printf("mancano %d ms%n", future.getDelay(TimeUnit.MILLISECONDS));
```

```
mancano 1598 ms
eseguito alle 10:42:17.003
```

`schedule` restituisce uno `ScheduledFuture`, un `Future` che sa anche quanto manca; `cancel` lo toglie dalla coda prima che parta. Per i compiti periodici ci sono due metodi, e la differenza conta quando il compito è lento.

```
executor.scheduleAtFixedRate(task, 0, 1, TimeUnit.SECONDS);
```

`scheduleAtFixedRate` avvia il compito ogni secondo, *contato dall'inizio dell'esecuzione precedente*. Se il compito dura tre secondi, le esecuzioni si susseguono senza pausa: ogni inizio alla prima scadenza utile, 0, 3, 6, 9. Non si accumulano esecuzioni arretrate, e con un solo thread non si sovrappongono.

```
executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SECONDS);
```

`scheduleWithFixedDelay` aspetta un secondo *dalla fine* dell'esecuzione precedente: con lo stesso compito da tre secondi, gli inizi sono a 0, 4, 8, 12.



Il primo serve a un orologio o a un campionamento a frequenza fissa, che non deve *derivare* se un'esecuzione è lenta; il secondo a un lavoro di manutenzione che deve lasciare respiro fra una corsa e l'altra.

Attenzione 7.1: Il periodico che muore in silenzio

Un'eccezione non catturata in un compito periodico *ferma le ripetizioni*: lo `ScheduledFuture` la conserva, come ogni `Future` (capitolo 4), ma nessuno di solito lo interroga, e il compito smette di girare senza che appaia niente. Un compito periodico avvolge il suo corpo in un `try/catch` che registra l'errore e lascia continuare le ripetizioni.

7.2 Stream paralleli, visti da sotto

Con quanto visto si può chiudere un discorso lasciato aperto nella dispensa *Stream, per intero*: che cosa succede con `parallel()`. L'esempio guida in quattro righe:

```
long n = LongStream.rangeClosed(1, 100_000_000)
        .parallel()
        .filter(PrimeTester::isPrime)
        .count();
```

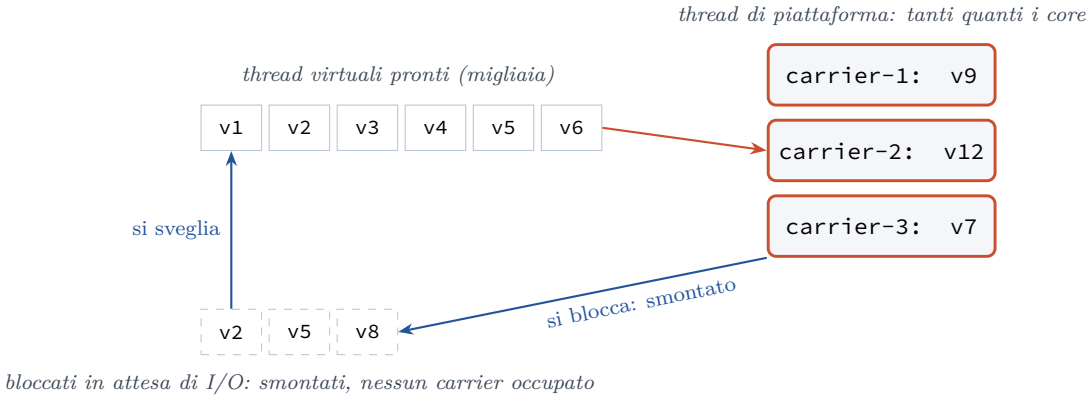
La libreria spacca la sorgente in pezzi (con `trySplit`, finché i pezzi sono abbastanza piccoli), consegna ogni pezzo come compito al `ForkJoinPool` *comune* della JVM, un pool a *work stealing* con tanti thread quanti i core meno uno (il chiamante partecipa), e fonde i risultati parziali. In pratica è il programma del capitolo 4 con i 64 compiti, scritto dalla libreria, con lo stesso tempo di esecuzione; e vale per questa versione tutto ciò che valeva per quella. Il lavoro dev'essere puro calcolo su elementi indipendenti; le lambda non devono toccare stato condiviso, perché girano su thread diversi in un ordine che dipende dalla divisione (una `ArrayList` riempita dentro un `forEach` parallelo è il contatore che perde del capitolo 5, in forma di lista); e il guadagno c'è solo se gli elementi sono molti e il lavoro per elemento non è banale.

Due cose in più, ora che sappiamo che cosa c'è sotto. Il pool comune è *uno per JVM*: tutti gli stream paralleli del programma, e i `CompletableFuture` senza executor esplicito, condividono gli stessi thread. Un compito che vi *aspetta* (una lettura di rete dentro una lambda) tiene fermo un thread del pool, e con pochi compiti così tutto il parallelismo della JVM si ferma: gli stream paralleli sono per il calcolo, non per l'I/O. E non sono per i casi in cui serve controllare i thread: chi vuole un numero diverso di thread, o isolare un lavoro dagli altri, usa `Executors.newWorkStealingPool(n)`, che è un `ForkJoinPool` privato, e vi consegna i compiti a mano.

7.3 Thread virtuali

JAVA 21 Chiudiamo il discorso aperto nel capitolo 2. I thread di piattaforma sono uno a uno con quelli del sistema operativo, e questo mette un tetto: un server che dedica un thread a ogni connessione, con diecimila connessioni aperte che per lo più *aspettano* dati, tiene diecimila stack e diecimila thread del kernel per fare, in ogni istante, pochissimo. Il pool limita il numero di thread, ma allora le connessioni in più aspettano in coda. Nessuna delle due è la risposta giusta, perché il problema non è il calcolo: è l'attesa.

Java 21 aggiunge i *thread virtuali*: thread gestiti dalla JVM e non dal kernel. Sono oggetti `Thread` a tutti gli effetti, con lo stesso codice, ma leggeri (lo stack vive nell'heap e cresce quanto serve, niente megabyte riservato) e numerosi (milioni, non migliaia). La JVM li esegue *montandoli* su un piccolo pool di thread di piattaforma, i *carrier*; quando un thread virtuale si blocca (`sleep`, una lettura di rete, una coda, un lock della libreria) viene *smontato*, il carrier passa a un altro, e al risveglio viene rimontato dove c'è posto. È l'immagine guida rovesciata: la coda contiene thread, non compiti, e i lavoratori sono i carrier. Ed è il punto in cui l'astrazione di Java smette di coincidere con quella del sistema operativo: un modello *molti a molti*.




```
Thread t = Thread.ofVirtual().start(() -> System.out.println("virtuale"));
t.join();

try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 10_000; i++)
        executor.submit(() -> { pausa(1000); return null; });
}                                          // close(): aspetta tutti
```

Diecimila compiti che dormono un secondo: con un pool di 200 thread di piattaforma ci vogliono 50 secondi (cinquanta turni da 200), con un thread virtuale per compito poco più di un secondo. Non è magia: i compiti *aspettano*, e aspettare in diecimila non costa niente se nessuno occupa un thread del kernel per farlo. Le regole d'uso derivano dal meccanismo.

- **Per l'attesa, non per il calcolo.** Un thread virtuale che calcola occupa il suo carrier esattamente come un thread di piattaforma: per i numeri primi non cambia niente, come mostra la sezione 7.4. Il guadagno c'è con molti compiti bloccati su I/O.
- **Non si mettono in pool.** Costano così poco che si crea un thread per compito e lo si butta: `newVirtualThreadPerTaskExecutor` fa esattamente questo, e un pool di thread virtuali è un controsenso.
- **Attenzione a `synchronized`, in Java 21.** Un thread virtuale bloccato *dentro* un blocco `synchronized` resta *inchiodato* al carrier, che non può servire altri: nelle sezioni critiche

lunghe dei thread virtuali conviene `ReentrantLock` (capitolo 6). Le versioni successive di Java hanno tolto il limite.

- **Tutto il resto è uguale.** Lock, code, variabili atomiche, executor, interruzione: il codice scritto per i thread di piattaforma gira sui virtuali senza modifiche, ed è la ragione per cui vale la pena averli imparati prima.

7.4 L'esempio guida in tre modi, su calcolo e su attesa

Chiudiamo con una misura che mette in fila tutto il capitolo. Due lavori: il conteggio dei primi fino a 10^8 in 64 compiti (*calcolo*), e diecimila compiti che dormono un secondo ciascuno, come una richiesta di rete (*attesa*). Tre modi di eseguirli: il pool fisso con tanti thread quanti i core, lo stream parallelo (dove ha senso) e un thread virtuale per compito. Sul portatile del capitolo 4:

	calcolo: i primi	**attesa: 10 000 × 1 s**
pool fisso, 16 thread	1,4 s	625 s (stimato: 625 turni)
pool fisso, 200 thread	1,5 s	50 s
pool *cached*	1,5 s	1,9 s (10 000 thread del kernel)
stream parallelo	1,4 s	non adatto
thread virtuali, uno per compito	1,5 s	1,0 s

Sul calcolo i cinque modi si equivalgono: il lavoro è lo stesso, i core sono gli stessi, e cambia solo chi distribuisce i compiti. Sull'attesa cambia tutto, e la colonna di destra è il capitolo 2 messo alla prova: il pool fisso serve 16 o 200 compiti alla volta e gli altri aspettano in coda; il pool *cached* crea diecimila thread di piattaforma, che il sistema regge (dieci gigabyte di stack riservato, non usato) e che dormono tutti insieme, al costo di diecimila thread del kernel e del tempo per crearli; i thread virtuali dormono senza occupare nulla. I thread virtuali risolvono l'attesa, non il calcolo; il pool fisso risolve il calcolo, non l'attesa; e un programma che fa entrambe le cose, come un server che riceve richieste e le elabora, usa entrambi.

Esercizi

Esercizio 7.1: Gli orari (⋆)

Un compito dura due secondi. Scrivere gli istanti di inizio delle prime cinque esecuzioni con `scheduleAtFixedRate(task, 0, 3, SECONDS)` e con `scheduleWithFixedDelay(task, 0, 3, SECONDS)`. Poi lo stesso con il periodo di un secondo. In quale dei quattro casi le esecuzioni si sovrappongono, se il pool ha due thread?

Esercizio 7.2: Il periodico che muore (⋆)

Un compito con `scheduleAtFixedRate` stampa l'ora ogni secondo; alla quinta esecuzione lancia `ArithmeticException`. Che cosa si vede sulla console dalla sesta in poi? Come si scopre che cosa è successo, e come si scrive il compito perché non succeda?

Esercizio 7.3: La lista che perde (⋆⋆)

Che cosa può stampare questo frammento, e perché? Dire tre modi di ottenere la lista giusta, e quale è quello buono.

```
List<Integer> l = new ArrayList<>();
IntStream.range(0, 100_000).parallel().forEach(l::add);
System.out.println(l.size());
```

Esercizio 7.4: Virtuali o no (⋆⋆)

Per ciascun caso dire se i thread virtuali aiutano, non cambiano niente, o peggiorano: (a) un server che tiene cinquemila connessioni quasi sempre inattive; (b) la compressione di cento file su otto core; (c) diecimila compiti che fanno una richiesta HTTP e sommano i risultati; (d) un compito che tiene un blocco `synchronized` per cento millisecondi mentre legge dalla rete, in Java 21.

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 7.5: Periodico e lento (⋆)

Con un `ScheduledExecutorService` a un thread, avviare un compito che stampa l'ora (con i millisecondi) e dorme tre secondi, con periodo di un secondo, prima con `scheduleAtFixedRate` e poi con `scheduleWithFixedDelay`, per quindici secondi ciascuno. Prevedere gli orari, poi verificarli. Infine fare in modo che il compito, alla quarta esecuzione, lanci un'eccezione, e mostrare come il programma se ne accorge interrogando lo `ScheduledFuture`.

Soluzione: progetto `sol-d04-c07`, package `it.unibs.pajc.redolfi.e05`.

Attenzione 7.2: `get()` su un periodico

`future.get()` su un compito periodico sano non ritorna mai (il compito non "finisce"); ritorna, con `ExecutionException`, solo quando il compito è morto. Per accorgersi della morte senza bloccarsi si usa `isDone()`, o si cattura l'eccezione nel compito stesso.

Esercizio 7.6: Dieci server lenti (⋆⋆)

Simulare dieci "server" come metodi che dormono un tempo casuale fra 200 e 800 ms e restituiscono una stringa. Scrivere un programma che li interroga tutti e stampa le risposte nell'ordine dei server, prima in sequenza, poi con un pool fisso di quattro thread, poi con un thread virtuale per server; misurare i tre tempi e commentarli. Poi portare i server a diecimila (dormono tutti un secondo): quale dei tre modi regge, e quanto ci mette?

Soluzione: progetto `sol-d04-c07`, package `it.unibs.pajc.redolfi.e06`.

Esercizio 7.7: Lo stream che scrive (⋆⋆)

Scrivere il frammento dell'esercizio 7.3 e mostrare che perde (o lancia). Poi scriverlo in tre modi corretti: con una lista sincronizzata e la misura del tempo; con `Collections.synchronizedList` sostituita da una `ConcurrentLinkedQueue`; con `collect(Collectors.toList())`, che è il modo giusto. Spiegare nel commento perché il collettore non ha bisogno di nessun lock.

Soluzione: progetto `sol-d04-c07`, package `it.unibs.pajc.redolfi.e07`.

Risposte

Risposta 7.1. Periodo 3 s, compito da 2 s: fixed rate 0, 3, 6, 9, 12 (il periodo è più lungo del lavoro, tutto regolare); fixed delay 0, 5, 10, 15, 20 (due di lavoro più tre di pausa). Periodo 1 s: fixed rate 0, 2, 4, 6, 8 (ogni esecuzione parte alla prima scadenza dopo la fine della precedente, senza recuperare i turni persi); fixed delay 0, 3, 6, 9, 12. In nessun caso si sovrappongono, nemmeno con due thread: la libreria non esegue mai due istanze dello stesso compito periodico insieme, e il secondo thread resta libero per altri compiti.

Risposta 7.2. Niente: dalla sesta in poi la console tace, senza stack trace, e il programma continua (e se il pool non viene spento, non termina nemmeno). Si scopre interrogando il future: `isDone()`, che da quel momento è `true`, e `get()`, che lancia `ExecutionException` con dentro l'`ArithmeticException`. Si evita avvolgendo il corpo del compito in un `try/catch` che registra l'errore e non rilancia: il compito "riesce" sempre dal punto di vista dell'executor, e le ripetizioni continuano.

Risposta 7.3. Un numero minore di 100 000, oppure `ArrayIndexOutOfBoundsException` dal profondo di `ArrayList`: gli `add` da thread diversi si sovrappongono sull'array interno e sul contatore `size`, come il contatore del capitolo 5. Tre modi: `Collections.synchronizedList` (corretto, ma ogni `add` prende un lock e il parallelismo si perde); una collezione concorrente come `ConcurrentLinkedQueue` (corretto, ordine non garantito); il collettore, `boxed().toList()`, che è il modo giusto: ogni thread riempie una lista propria (confinamento) e la libreria le fonde alla fine, in ordine, senza lock.

Risposta 7.4. (a) Aiutano molto: cinquemila thread che aspettano, un thread virtuale ciascuno, nessun carrier occupato. (b) Niente: è calcolo, i core sono otto, e otto thread di piattaforma in un pool fisso sono la scelta giusta. (c) Aiutano: diecimila attese insieme, tempo totale pari a quello della richiesta più lenta. (d) Peggiorano, in Java 21: il thread virtuale resta inchiodato al carrier per tutta l'attesa, e con pochi compiti così i carrier sono tutti occupati; con `ReentrantLock` al posto di `synchronized` il problema sparisce.

Capitolo 8

Thread e Swing: la coda degli eventi

Torniamo alla finestra del capitolo 1, con tutto ciò che abbiamo imparato. Sappiamo già, dal `jstack` del capitolo 2, che mentre `main` contava i primi c'era un thread di nome `AWT-EventQueue-0` che aspettava: è lui il protagonista di questo capitolo. Un'interfaccia grafica è il luogo in cui la concorrenza si incontra per forza, perché l'utente non aspetta, e in cui le regole dei capitoli precedenti si presentano in forma concentrata: un solo thread che non va mai bloccato, uno stato (i componenti) che non è thread-safe, e un modo preciso per far rientrare i risultati. Questo capitolo dà per note le basi di Swing (una finestra, dei componenti, un listener) e spiega che cosa c'è sotto, perché senza saperlo è impossibile scrivere un'interfaccia che resti viva. Alla fine `PrimeFinderApp` avrà la sua versione giusta.

8.1 Un'interfaccia è un ciclo

Un programma a eventi non "fa" qualcosa dall'inizio alla fine: aspetta. Da qualche parte c'è un ciclo che riceve gli eventi (un click, un tasto, una finestra da ridisegnare) e per ciascuno chiama il codice registrato. In Swing quel ciclo gira su un thread dedicato, l'*Event Dispatch Thread* (EDT), che nel debugger e in `jstack` si chiama `AWT-EventQueue-0`. È lui che esegue ogni listener, ogni `paintComponent`, e il `Runnable` passato a `invokeLater`. E vale una regola, scritta nella documentazione di ogni componente:

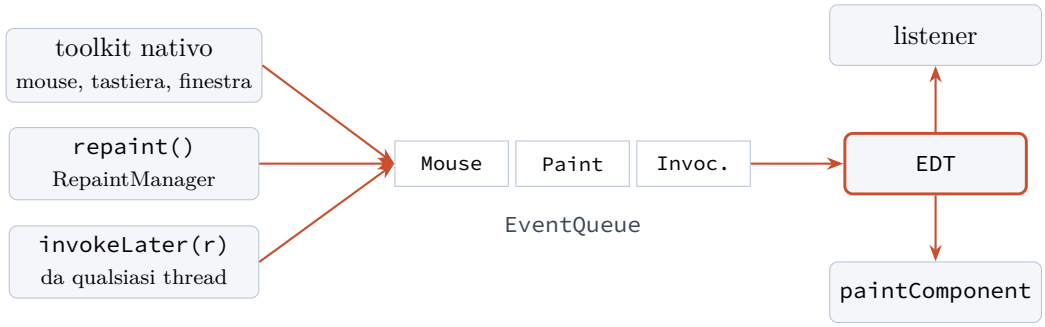
Definizione 8.1: La regola del singolo thread di Swing

I componenti Swing non sono thread-safe. Ogni accesso a un componente, dalla creazione alla lettura di una proprietà alla modifica, va eseguito dall'Event Dispatch Thread.

Perché una libreria moderna rinuncia alla thread safety, dopo tutto quello che il capitolo 6 ha offerto? Perché un'interfaccia è un albero di componenti che si modificano a vicenda in continuazione: un testo che cambia fa ridisegnare il contenitore, che ricalcola il layout, che sposta i fratelli. Proteggere ogni componente con un lock produrrebbe deadlock a ogni angolo, come i due conti del capitolo 5 moltiplicati per cento, e i toolkit che ci hanno provato sono tornati indietro. Un solo thread è la scelta di tutte le librerie grafiche di oggi: Swing, JavaFX, Android, il browser. È il confinamento del capitolo 5 applicato a un intero albero di oggetti. Il prezzo è che quel thread *non va mai bloccato*.

8.2 Come funziona `EventQueue`

Il cuore è la classe `java.awt.EventQueue`: una coda di oggetti `AWTEvent`, con più produttori e un solo consumatore. È l'immagine guida con un lavoratore solo.



pumpEvents: estrai, dispatchEvent, ricomincia

- **I produttori.** Il *toolkit nativo* ha un thread suo (`AppKit Thread` su macOS, `AWT-XAWT` su Linux, `AWT-Windows`) che riceve gli eventi dal sistema operativo e li traduce in `MouseEvent`, `KeyEvent`, `WindowEvent`, accodandoli con `postEvent`. Il programma stesso è un produttore in due modi: `repaint()` non disegna niente, chiede al `RepaintManager` di accodare un `PaintEvent` (e più richieste vicine vengono *fuse* in una sola: cento `repaint` in un ciclo costano un disegno); e `EventQueue.invokeLater(r)` incarta un `Runnable` in un `InvocationEvent` e lo accoda. È l'unico modo con cui un altro thread può far eseguire qualcosa all'EDT, e il capitolo 6 ne ha mostrato la forma generale: una `BlockingQueue` fra chi produce e chi consuma.
- **Il consumatore.** L'EDT esegue il ciclo `pumpEvents`: estrae l'evento in testa, chiama `dispatchEvent`, che lo recapita al componente giusto, che chiama i suoi listener uno dopo l'altro; per un `PaintEvent` chiama `paint`, per un `InvocationEvent` chiama `run()`. Finito, ricomincia. Un solo thread, un evento alla volta, nell'ordine della coda.

Da questo meccanismo discende tutto ciò che conta. `revalidate()` e `repaint()` sono *richieste*, soddisfatte quando l'EDT arriverà all'evento, non azioni immediate: chiamarle dentro un listener non disegna niente finché il listener non ritorna. Due `invokeLater` dallo stesso thread vengono eseguiti nell'ordine in cui sono stati accodati. E un listener che non ritorna ferma *tutta* la coda: è la finestra congelata.

8.3 Chi avvia che cosa

La `PrimeFinderApp` del capitolo 1 aveva un secondo errore, più piccolo, prima ancora del click: `main` costruiva la finestra. Il `main` gira sul thread `main`, che per la regola della definizione 8.1 non deve toccare i componenti. La forma giusta accoda la costruzione:

```
public static void main(String[] args) {
    EventQueue.invokeLater(() -> new PrimeFinderApp().setVisible(true));
}
```

`main` termina subito, e il programma no: l'EDT e il thread del toolkit non sono daemon (capitolo 2) e restano vivi finché esiste una finestra; `EXIT_ON_CLOSE` chiude la JVM alla chiusura dell'ultima. `EventQueue.invokeLater` e `SwingUtilities.invokeLater` sono lo stesso metodo; `EventQueue.isDispatchThread()` dice se il thread corrente è l'EDT, e serve per scrivere codice

che funziona chiamato da entrambe le parti. Costruire la finestra dal thread `main` funziona quasi sempre, come tutte le violazioni della regola: ed è il motivo per cui la si vede in troppi programmi.

8.4 Il blocco

Ora possiamo rileggere la prima versione di `conta()` sapendo che cosa succede:

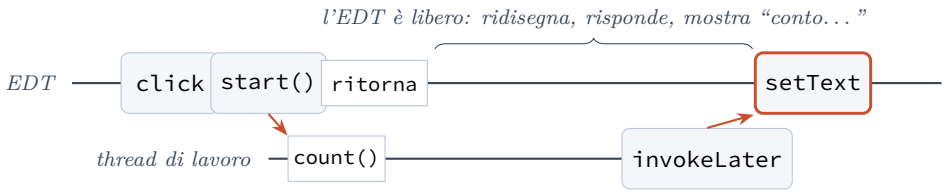
```
private void conta() {                          // sull'EDT: congela la finestra
    long max = Long.parseLong(campoMax.getText());
    lblRisultato.setText("conto...");
    long n = PrimeTester.count(1, max);         // 14 secondi
    lblRisultato.setText("primi: " + n);
}
```

`conta` è il listener del pulsante, l'EDT lo sta eseguendo, e finché non ritorna non estrae nessun altro evento: né i `MouseEvent` (la finestra non risponde), né i `PaintEvent` (non si ridisegna), nemmeno quello accodato dal primo `setText`: per questo "conto…" non compare mai. Il risultato arriva, ma dopo che tutto si è fermato. Il `jstack` del capitolo 2 lo mostrava già: un thread `RUNNABLE` dentro `isPrime`, e `AWT-EventQueue-0` in attesa; con la versione giusta del `main`, il thread dentro `isPrime` sarebbe stato l'EDT stesso.

8.5 Lavoro fuori, rientro dentro

Il calcolo va su un altro thread; ma il risultato va mostrato su un componente, e toccarlo da quel thread viola la regola. La soluzione è in due tempi: il lavoro fuori dall'EDT, l'aggiornamento rientra sull'EDT con `invokeLater`, cioè attraverso la coda.

```
private void conta() {
    long max = Long.parseLong(campoMax.getText());  // 1. leggo sull'EDT
    btnConta.setEnabled(false);
    lblRisultato.setText("conto...");
    new Thread(() -> {                                  // 2. calcolo su un altro
    thread
        long n = PrimeTester.count(1, max);
        EventQueue.invokeLater(() -> {                  // 3. rientro sull'EDT
            lblRisultato.setText("primi: " + n);
            btnConta.setEnabled(true);
        });
    }).start();
}
```



il risultato rientra come evento, e l'EDT lo esegue al suo turno

Il listener ritorna in un istante e l'EDT torna a svuotare la coda: la finestra si ridisegna, il pulsante appare disabilitato, l'etichetta dice "conto…". Quattordici secondi dopo il thread di lavoro accoda un `InvocationEvent`, e l'EDT, al suo turno, aggiorna l'etichetta. La lambda cattura `n`, che è effectively final (capitolo 3), e `invokeLater` garantisce che ciò che il thread di lavoro ha fatto prima sia visibile all'EDT (capitolo 5). Si noti che `max` viene letto *prima*, sull'EDT: leggere `campoMax.getText()` dentro il thread di lavoro sarebbe un accesso a un componente dal thread sbagliato.

Attenzione 8.1: "Ma se chiamo `setText` dal thread di lavoro, funziona"

Quasi sempre, ed è il problema. Non c'è nessun controllo e nessuna eccezione: il componente viene modificato mentre l'EDT, magari, lo sta disegnando o ne sta ricalcolando il layout, e il risultato è un'interfaccia che una volta ogni mille mostra un valore vecchio, un testo tagliato, un componente al posto sbagliato, o lancia un'eccezione dal profondo di Swing che non si riesce a riprodurre. È il contatore che perde del capitolo 5, con uno stato molto più grande di un intero. Un programma del genere, con un'etichetta aggiornata cento volte al secondo da un thread, prima o poi lo fa vedere.

`invokeAndWait(r)` è la variante che aspetta l'esecuzione di `r` prima di ritornare: serve quando il thread di lavoro ha bisogno di una risposta dall'interfaccia (un valore da un campo, una conferma). Chiamato *dall'EDT* lancia un `Error`, perché sarebbe un deadlock certo: l'EDT aspetterebbe un evento che solo lui può eseguire. In pratica si evita, e si legge tutto ciò che serve prima di avviare il lavoro.

Ricetta: eseguire sull'EDT da qualunque thread

```
void aggiorna(String testo) {
    Runnable r = () -> lblRisultato.setText(testo);
    if (EventQueue.isDispatchThread())
        r.run();                              // sono gia' sull'EDT: subito
    else
        EventQueue.invokeLater(r);            // altrimenti: in coda
}
```

Un metodo del genere può essere chiamato da un listener e da un thread di lavoro senza che chi lo chiama debba sapere dove si trova.

8.6 `SwingWorker`

Lo schema "lavoro fuori, rientro dentro" è così comune che Swing lo fornisce pronto, con in più l'avanzamento, i risultati parziali e l'annullamento: `SwingWorker<T, V>`, dove `T` è il tipo del risultato finale e `V` quello dei risultati intermedi. La terza versione di `conta()` aggiunge una barra di avanzamento alla finestra:

```
private void conta() {
    long max = Long.parseLong(campoMax.getText());
    btnConta.setEnabled(false);
    SwingWorker<Long, Void> worker = new SwingWorker<>() {
        @Override
        protected Long doInBackground() {            // su un thread di lavoro
            int nBlocchi = 100;
            long perBlocco = max / nBlocchi, totale = 0;
            for (int b = 0; b < nBlocchi && !isCancelled(); b++) {
                totale += PrimeTester.count(
                        b * perBlocco + 1, (b + 1) * perBlocco);
                setProgress(b + 1);                  // 0..100
            }
            return totale;
        }

        @Override
        protected void done() {                      // sull'EDT, alla fine
            try {
                lblRisultato.setText("primi: " + get());
            } catch (CancellationException e) {
                lblRisultato.setText("annullato");
            } catch (InterruptedException | ExecutionException e) {
                lblRisultato.setText("errore: " + e.getCause());
            }
            btnConta.setEnabled(true);
        }
    };
    worker.addPropertyChangeListener(e -> {          // sull'EDT
        if ("progress".equals(e.getPropertyName()))
            barra.setValue((Integer) e.getNewValue());
    });
    worker.execute();
}
```

I pezzi, e dove girano:

- `doInBackground()` è il compito: gira su un thread di un pool interno a Swing, e non deve toccare componenti. Restituisce il risultato o lancia. È un `Callable` con un altro nome.
- `done()` viene chiamato sull'EDT quando il compito è finito, in qualsiasi modo. `get()` qui non blocca: il risultato c'è già, oppure c'è un'eccezione da rilanciare (`ExecutionException` per un errore nel compito, `CancellationException` se è stato annullato), esattamente come con un `Future` del capitolo 4.
- `setProgress(0..100)` dal compito aggiorna la proprietà `progress`; chi si è registrato con `addPropertyChangeListener` viene avvisato sull'EDT. È il canale per la barra di avanzamento.
- `publish(v...)` dal compito e `process(List<V>)` sull'EDT sono il canale per i risultati intermedi (le righe di un file man mano che si leggono): Swing raggruppa i valori pubblicati e li consegna a blocchi, così mille `publish` non sono mille eventi.

- `cancel(true)` interrompe il thread del compito e fa valere `isCancelled()`: il ciclo di `doInBackground` lo controlla a ogni blocco e si ferma. È l'interruzione cooperativa del capitolo 3, con un nome nuovo.
- `execute()` accoda il worker; un `SwingWorker` si esegue una volta sola, per un nuovo calcolo se ne crea uno nuovo.

8.7 `javax.swing.Timer`

Un orologio, un'animazione, un controllo periodico dello stato: serve un compito ripetuto che tocca i componenti. Lo `ScheduledExecutorService` del capitolo 7 lo eseguirebbe sul thread sbagliato. `javax.swing.Timer` è un timer che consegna i suoi `ActionEvent` *sull'EDT*, attraverso la stessa coda di tutto il resto:

```
Timer timer = new Timer(1000, e ->                    // ogni secondo, sull'EDT
        lblOra.setText(LocalTime.now().toString()));
timer.start();
...
timer.stop();
```

Il listener può toccare i componenti senza nessun rientro; in cambio dev'essere breve, perché gira sull'EDT. La regola per scegliere: `javax.swing.Timer` per compiti brevi che aggiornano l'interfaccia, `ScheduledExecutorService` per il lavoro vero, con il rientro via `invokeLater` quando c'è qualcosa da mostrare. `java.util.Timer` è una terza classe, più vecchia, con un thread proprio e senza nessun legame con l'EDT: con Swing non si usa, e confonderla con quella di `javax.swing` è un errore classico che compila senza una parola.

Ricetta: thread e Swing, in cinque regole

1. **La finestra nasce sull'EDT:** `main` fa `EventQueue.invokeLater(() -> new App().setVisible(true))` e basta.
2. **L'EDT non si blocca mai:** niente calcoli lunghi, niente `sleep`, niente rete nei listener. Un listener dura millisecondi.
3. **Il lavoro va altrove:** un `SwingWorker` per un'operazione con risultato, un executor per il lavoro continuo, `javax.swing.Timer` per il periodico breve.
4. **Si rientra sempre:** ogni accesso a un componente da un altro thread passa da `invokeLater`, `done()`, `process()` o un listener di proprietà.
5. **Il modello condiviso è protetto o confinato:** i dati che il thread di lavoro produce e l'interfaccia mostra sono immutabili, oppure passano in una coda, oppure sono protetti da un lock; e si leggono dall'interfaccia prima di partire, non durante.

Esercizi

Esercizio 8.1: Dove gira (⋆)

Per ciascuno dire su quale thread viene eseguito: (a) il corpo di `main`; (b) il listener di un pulsante; (c) `doInBackground` di un `SwingWorker`; (d) `done`; (e) `process`; (f) il listener di un `javax.swing.Timer`; (g) il `Runnable` passato a `invokeLater` da un thread di lavoro; (h) `paintComponent`.

Esercizio 8.2: "conto..." (⋆)

Nella prima versione di `conta()`, `setText("conto...")` viene chiamato *prima* del calcolo, eppure non compare. Spiegare la sequenza esatta degli eventi nella coda, e perché nella seconda versione compare.

Esercizio 8.3: Due in coda (⋆)

Un thread di lavoro esegue `invokeLater(() -> a.setText("1"))` e subito dopo `invokeLater(() -> a.setText("2"))`. Che cosa mostra l'etichetta alla fine? E se le due chiamate arrivano da due thread di lavoro diversi?

Esercizio 8.4: `invokeAndWait` dal posto sbagliato (⋆⋆)

Un listener chiama `EventQueue.invokeAndWait(r)` per "essere sicuro" che `r` sia eseguito prima di continuare. Che cosa succede, e perché la libreria lo impedisce con un `Error` invece di lasciar fare?

Esercizio 8.5: Il timer sbagliato (⋆⋆)

Un programma importa `java.util.Timer` invece di `javax.swing.Timer` e aggiorna un'etichetta ogni secondo nel suo `TimerTask`. Compila, gira e l'etichetta si aggiorna. Dov'è l'errore, come lo si vede con il debugger, e come lo si corregge in una riga senza cambiare timer?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 8.6: Un orologio (⋆)

Una finestra con un'etichetta che mostra l'ora con i decimi di secondo, aggiornata da un `javax.swing.Timer` ogni 100 ms, e due pulsanti "Ferma" e "Riprendi". Verificare con `Thread.currentThread().getName()` nel listener su quale thread gira. Poi scrivere la stessa finestra con uno `ScheduledExecutorService` e il rientro con `invokeLater`, e dire quale delle due versioni si preferisce e perché.

Soluzione: progetto `sol-d04-c08`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 8.2: Chi ferma chi

Un timer o un executor avviato nel costruttore della finestra va fermato quando la finestra si chiude: un `WindowListener` con `windowClosing`, oppure `EXIT_ON_CLOSE` che termina tutto. Un executor dimenticato tiene viva la JVM anche a finestra chiusa (capitolo 4).

Esercizio 8.7: Le righe arrivano (⋆⋆)

Una finestra con una `JTextArea` in uno `JScrollPane` e un pulsante "Carica": al click un `SwingWorker<Void, String>` legge un file di testo grande (o ne simula uno, generando diecimila righe con una pausa di un millisecondo ciascuna) e pubblica ogni riga con `publish`; `process` le aggiunge alla `JTextArea`. Stampare in `process` quante righe arrivano per chiamata e osservare come Swing le raggruppa. L'interfaccia deve restare reattiva durante il caricamento.

Soluzione: progetto `sol-d04-c08`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 8.8: Annulla (⋆⋆)

Partire dalla terza versione di `PrimeFinderApp` (con `SwingWorker` e barra) e aggiungere un pulsante "Annulla" che chiama `cancel(true)`: alla pressione l'etichetta deve dire "annullato", la barra tornare a zero e il pulsante "Conta" tornare attivo. Il worker va tenuto in un campo della finestra. Dire nel commento che cosa succederebbe se il calcolo fosse un blocco unico invece di cento blocchi.

Soluzione: progetto `sol-d04-c08`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 8.9: Il conteggio dal vivo (⋆⋆⋆)

Una finestra che mostra, aggiornato ogni secondo, quanti primi sono stati trovati finora da un conteggio in corso su un pool di thread (il programma del capitolo 4, con un `LongAdder` condiviso come nel capitolo 6). Il conteggio parte da un pulsante e gira su un executor proprio; un `javax.swing.Timer` legge il `LongAdder` ogni secondo e aggiorna l'etichetta; alla fine l'executor rientra con `invokeLater` per scrivere "finito" e fermare il timer. Progettare chi tocca che cosa e da quale thread, e scriverlo nel commento prima del codice.

Suggerimento: il `LongAdder` è l'unico dato condiviso, ed è thread-safe; tutto il resto è confinato nell'EDT o nei compiti.

Soluzione: progetto `sol-d04-c08`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 8.1. (a) Sul thread `main`. (b) Sull'EDT. (c) Su un thread del pool interno di `SwingWorker`. (d) Sull'EDT. (e) Sull'EDT. (f) Sull'EDT. (g) Sull'EDT: è il senso di `invokeLater`. (h) Sull'EDT. Tutto ciò che tocca i componenti gira sull'EDT; solo `main` e `doInBackground` stanno fuori, e infatti non devono toccarli.

Risposta 8.2. Il click è un `MouseEvent`; l'EDT lo estrae e chiama il listener. Il primo `setText` cambia il testo del modello dell'etichetta e accoda un `PaintEvent`: la richiesta di ridisegnare. Poi il listener chiama `count`, per quattordici secondi, senza ritornare: l'EDT è dentro il listener e non estrae il `PaintEvent`. Quando `count` finisce, il secondo `setText` cambia di nuovo il testo (il `PaintEvent` è fuso con il primo), il listener ritorna, l'EDT estrae il `PaintEvent` e disegna: "primi: ...". "conto..." è stato nel modello, mai sullo schermo. Nella seconda versione il listener ritorna subito dopo il `setText`, l'EDT estrae il `PaintEvent` e disegna "conto..." mentre l'altro thread lavora.

Risposta 8.3. "2": i due `InvocationEvent` sono accodati nell'ordine delle chiamate e l'EDT li esegue in quell'ordine. Da due thread diversi l'ordine di accodamento dipende da chi chiama `invokeLater` per primo, e non è prevedibile: può restare "1". Se l'ordine conta, lo si impone fra i thread di lavoro (un `join`, una coda), non contando sull'EDT.

Risposta 8.4. `invokeAndWait` accoda `r` e aspetta che venga eseguito. Ma l'unico thread che esegue gli eventi è l'EDT, che è fermo nel listener ad aspettare: nessuno eseguirà mai `r`. Deadlock certo, con un solo thread. La libreria lo rileva (`isDispatchThread()` è vero) e lancia un `Error` subito, perché un blocco silenzioso della finestra sarebbe molto più difficile da diagnosticare. Dall'EDT si chiama `r.run()` direttamente, come nella ricetta della sezione 8.5.

Risposta 8.5. `java.util.Timer` esegue il `TimerTask` su un thread proprio (`Timer-0`): `setText` viene chiamato dal thread sbagliato, e "funziona" nel senso dell'attenzione della sezione 8.5. Nel debugger, sospendendo dentro `run()`, si vede il nome del thread. La correzione in una riga: dentro `run()`, `EventQueue.invokeLater(() -> lbl.setText(...))`. La correzione migliore: `javax.swing.Timer`, che fa il rientro da sé.

Capitolo 9

Palestra

Nessuna teoria nuova. Nove programmi in ordine crescente di difficoltà, ciascuno con l'indicazione di che cosa allena, che arrivano a chiudere i due problemi del capitolo 1: usare tutta la macchina e tenere viva la finestra. I primi tre sono nello stile delle prove scritte: contare i thread di un frammento, scrivere una classe con un metodo multithread, dividere un calcolo. I tre seguenti mettono insieme gli strumenti del capitolo 6 in programmi che somigliano a quelli veri: listener, code, download. Gli ultimi tre sono programmi completi con interfaccia o con una struttura da progettare. Dopo ogni enunciato, l'errore che abbiamo visto fare più spesso su quel tipo di problema; le soluzioni, con i commenti sulle scelte, sono nel repository delle soluzioni del corso, progetto `sol-d04-c09`. Il consiglio di sempre, più forte del solito: carta, penna, un disegno di chi tocca che cosa da quale thread, e la soluzione solo dopo.

Problemi

Esercizio 9.1: Contare i thread (⋆)

Allena: `start` contro `run`, il ciclo di vita di un `Thread`. Per ciascuno dei quattro frammenti dire, senza eseguire, quanti thread vengono creati oltre a `main`, che cosa stampa, e se qualcosa va storto. Poi scrivere un programma che li esegue uno alla volta (ciascuno in un `try`/`catch`) e stampa, per ciascuno, il numero di thread *vivi* subito dopo (`Thread.activeCount()`) e il nome del thread che ha stampato.

```
Runnable r = () -> System.out.println("ciao da " +
    Thread.currentThread().getName());
// (a)
new Thread(r).start(); new Thread(r).start(); r.run();
// (b)
Thread t = new Thread(r); t.run(); t.start();
// (c)
Thread u = new Thread(r); u.start(); u.join(); u.start();
// (d)
for (int i = 0; i < 3; i++) new Thread(r).run();
```

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e01`.

Attenzione 9.1: Il metodo si chiama `run` e non crea niente

In (b) e (d) si legge "`Thread`" e si contano thread: ma ciò che conta è solo `start()`. E in (c) il secondo `start()` su un thread terminato non lo "riavvia": lancia `IllegalThreadStateException`, come ogni `start()` che non sia il primo.

Esercizio 9.2: Numeri felici (⋆)

Allena: una classe con un metodo multithread, nello stile delle prove scritte. Un numero è *felice* se, sostituendolo ripetutamente con la somma dei quadrati delle sue cifre, si arriva a 1 (per esempio $19 \to 82 \to 68 \to 100 \to 1$); altrimenti il processo entra in un ciclo che contiene 4. Scrivere una classe `Felici` con `static boolean isFelice(int n)`, `static int conta(int da, int a)` e `static int contaMT(int da, int a, int nThread)`, che divide l'intervallo in `nThread` parti, le conta su un pool con `Callable` e `Future` e somma. Il `main` verifica che `conta` e `contaMT` diano lo stesso risultato fino a dieci milioni e confronta i tempi con 1, 2, 4 e 8 thread.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e02`.

Attenzione 9.2: Gli estremi degli intervalli

L'errore più frequente nelle prove scritte è nei bordi: un numero contato due volte (l'estremo in comune fra due intervalli) o mai (l'ultimo, quando `a` non è multiplo di `nThread`). L'ultimo compito prende "fino ad `a`", non "fino a `(k+1) * ampiezza`".

Esercizio 9.3: Il crivello a blocchi (⋆⋆)

Allena: dividere un calcolo che non si divide da solo; `LongAdder`; misurare. Contare i primi fino a 10^9 è troppo lento per divisioni successive. Scrivere un *crivello segmentato*: si calcolano una volta i primi fino a $\sqrt{10^9} \approx 31\,623$ con un crivello classico; poi l'intervallo si divide in blocchi di un milione, e ogni blocco, in un compito, viene crivellato con quei primi in un array di `boolean` locale al compito, contando i sopravvissuti in un `LongAdder`. Misurare con un pool di tanti thread quanti i core e confrontare con la versione a un thread (e, per curiosità, con le divisioni successive fino a 10^8). Risultato atteso: 50 847 534.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e03`.

Attenzione 9.3: L'array è del compito

L'array dei `boolean` va creato *dentro* il compito, uno per blocco: è il confinamento del capitolo 5. Un unico array condiviso di un miliardo di elementi costerebbe un gigabyte e, scritto da più thread, sarebbe un conflitto; un array per blocco costa un megabyte e non è condiviso con nessuno.

Esercizio 9.4: I listener (⋆⋆)

Allena: `CopyOnWriteArrayList`, la modifica durante l'iterazione. Scrivere una classe `Sorgente` con `addListener`, `removeListener` e `notifica(String evento)`, che chiama tutti i listener registrati. Più thread notificano eventi in continuazione mentre un altro thread aggiunge e toglie listener; uno dei listener, al terzo evento ricevuto, si *rimuove da solo* dentro la notifica. Prima con una `ArrayList`: mostrare che cosa succede (quale eccezione, e da quale thread). Poi con `CopyOnWriteArrayList`: spiegare nel commento perché un listener che si rimuove durante la notifica riceve comunque l'evento in corso, e perché va bene così.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e04`.

Attenzione 9.4: Sincronizzare il `for` non basta

Chi mette `synchronized` attorno al `for` di `notifica` evita la `ConcurrentModificationException` dagli altri thread, ma non dal listener che si rimuove *dentro* il `for`, sullo stesso thread: il monitor è rientrante e lo lascia passare. La copia per scrittura è la risposta, non il lock.

Esercizio 9.5: Code al supermercato (⋆⋆)

Allena: produttori e consumatori con `BlockingQueue`, chiusura ordinata, statistiche condivise. Un thread genera clienti (un `record Cliente(int id, long arrivo, int articoli)`) a intervalli casuali e li mette in una `BlockingQueue`; tre casse, ciascuna un thread, prendono il prossimo cliente e lo servono in un tempo proporzionale agli articoli. Dopo cento clienti il generatore chiude, e le casse devono terminare tutte senza perdere nessun cliente. Alla fine stampare, per cassa, quanti clienti ha servito, e in totale il tempo medio e massimo di attesa in coda. Decidere quali dati sono condivisi e come proteggerli, e scriverlo nel commento.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e05`.

Attenzione 9.5: Una sentinella per ogni cassa

Con tre consumatori una sola sentinella ferma una cassa sola. O se ne mettono tre, o chi la trova la rimette in coda prima di uscire (capitolo 5). E le statistiche aggiornate dalle tre casse (somma e massimo delle attese) sono stato condiviso: `LongAdder` e `LongAccumulator`, oppure un lock.

Esercizio 9.6: Scaricare in parallelo (⋆⋆)

Allena: thread virtuali per l'attesa, un contatore condiviso, un thread che osserva. Simulare lo scaricamento di cinquanta file di dimensione casuale fra 1 e 20 MB: ogni "scaricamento" dorme un millisecondo per ogni 10 KB e aggiorna un `AtomicLong` con i byte ricevuti man mano, a blocchi di 100 KB. Avviare tutti i cinquanta con un thread virtuale per file; intanto un thread di piattaforma stampa ogni 200 ms, sulla stessa riga, una barra di avanzamento in caratteri (`[#####  ] 48%`) calcolata sui byte totali attesi. Alla fine il tempo totale, e il confronto con lo stesso programma su un pool fisso di quattro thread.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 9.6: La barra non è esatta, e va bene

Il thread che stampa legge l'`AtomicLong` mentre cinquanta thread lo aggiornano: vede un valore sempre coerente (è atomico) ma già vecchio di qualche microsecondo. Per una barra di avanzamento è esattamente ciò che serve; chi volesse "fermare tutti per leggere" starebbe risolvendo un problema che non esiste.

Esercizio 9.7: `PrimeFinderApp` completa (⋆⋆⋆)

Allena: tutto insieme; chiude il problema d'apertura. La `PrimeFinderApp` del capitolo 1, finita: un campo per il limite, uno per il numero di thread (default: i core della macchina), "Conta" e "Annulla", una barra di avanzamento, un'etichetta con il risultato e il tempo impiegato. Al click, un `SwingWorker` crea un pool con il numero di thread scelto, divide l'intervallo in 64 compiti, li consegna, e man mano che i `Future` completano aggiorna l'avanzamento; "Annulla" ferma tutto (i compiti in corso devono accorgersene entro un decimo di secondo); alla chiusura della finestra nessun thread resta vivo. Con cento milioni e sedici thread il risultato deve arrivare in poco più di un secondo, con la finestra sempre reattiva.

Suggerimento: il pool vive dentro `doInBackground`, in un `try` con risorse; i compiti controllano `Thread.currentThread().isInterrupted()` ogni centomila numeri; `cancel(true)` interrompe il thread del worker, che fa `shutdownNow()` sul pool nel `finally`.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 9.7: Chi interrompe chi

`cancel(true)` interrompe il thread del `SwingWorker`, non i sedici thread del pool. Il worker deve propagare: `shutdownNow()` manda `interrupt()` a tutti i compiti in corso, e i compiti devono guardare la bandiera. Tre livelli, ciascuno cooperativo.

Esercizio 9.8: Un Mandelbrot in piccolo (⋆⋆⋆)

Allena: rendering progressivo, annullamento dei compiti obsoleti, il rientro delle immagini. Un pannello di 600×400 pixel che disegna l'insieme di Mandelbrot: per ogni pixel, il numero di iterazioni di $z \leftarrow z^2 + c$ prima che $|z| > 2$ (al massimo 200) dà il colore. Il calcolo è diviso in tre compiti a risoluzione crescente (un pixel ogni 8, ogni 2, tutti): ciascuno produce una `BufferedImage` e la consegna con `invokeLater`, così l'immagine grezza compare subito e si affina. Al click del mouse l'immagine si centra lì e lo zoom raddoppia: i compiti non ancora partiti vanno annullati (`Future.cancel`), quelli in corso devono interrompersi (controllo di `isInterrupted()` a ogni riga) e *non* consegnare un'immagine obsoleta. Usare un pool di tanti thread quanti i core.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e08`.

Attenzione 9.8: `interrupted()` consuma la bandiera

Un compito che controlla `Thread.interrupted()` a ogni riga *azzera* la bandiera al primo controllo positivo: se poi, prima di consegnare, la controlla di nuovo, la trova abbassata e consegna l'immagine obsoleta, che copre quella nuova. È un errore visto in un programma vero: si usa `isInterrupted()`, che legge senza toccare.

Esercizio 9.9: La cache che calcola una volta sola (⋆⋆⋆)

Allena: `ConcurrentHashMap` con `computeIfAbsent`, `Future` come segnaposto, un oggetto thread-safe progettato da zero. Scrivere una classe `Memo<K, V>` che avvolge una funzione costosa `Function<K, V>` e la calcola *una volta sola per chiave*, anche se sedici thread chiedono la stessa chiave nello stesso istante: chi arriva primo calcola, gli altri aspettano il *suo* risultato invece di ricalcolare. La struttura è una `ConcurrentHashMap<K, Future<V>>`: il valore è un `FutureTask` creato con `computeIfAbsent`, e chi lo trova già presente fa `get()`. Verificare con una funzione che dorme un secondo e conta le chiamate: sedici thread che chiedono le stesse dieci chiavi devono produrre dieci chiamate, non centosessanta, e finire in poco più di un secondo. Gestire il caso in cui la funzione lancia: la chiave non deve restare "avvelenata" per sempre.
Suggerimento: `computeIfAbsent` è atomico ma non deve eseguire il calcolo dentro di sé (bloccherebbe il segmento della mappa): inserisce il `FutureTask` e lo restituisce; chi lo ha inserito chiama `run()` subito dopo, fuori dalla mappa.

Soluzione: progetto `sol-d04-c09`, package `it.unibs.pajc.redolfi.e09`.

Attenzione 9.9: Calcolare dentro `computeIfAbsent`

Mettere la chiamata alla funzione costosa dentro la lambda di `computeIfAbsent` è corretto per il risultato e sbagliato per la concorrenza: la mappa tiene il lock del segmento per tutto il calcolo, gli altri thread che chiedono chiavi *diverse* nello stesso segmento aspettano, e una funzione che a sua volta usa la mappa va in deadlock. Il `Future` come segnaposto separa l'inserimento (atomico e istantaneo) dal calcolo (lungo e fuori dal lock).

Capitolo 10

Dove andare adesso

La cassetta degli attrezzi è completa per i programmi del corso: thread ed executor, `Future`, monitor e lock, le classi di `java.util.concurrent`, i thread virtuali, la coda degli eventi di Swing. Questo capitolo indica le direzioni in cui il discorso continua, con un assaggio di ciascuna e qualcosa da provare subito. I testi per ogni sentiero sono in fondo.

10.1 `CompletableFuture`: comporre invece di aspettare

Un `Future` si interroga con `get()`, e `get()` blocca. In un programma che fa molte cose asincrone (tre richieste di rete, poi un calcolo sui risultati, poi un aggiornamento) il blocco si moltiplica, e i thread passano il tempo ad aspettarsi. `CompletableFuture` JAVA 8 rovescia il modello: invece di chiedere il risultato, si dice *che cosa farne* quando arriverà.

```
CompletableFuture<Long> a = CompletableFuture.supplyAsync(() ->
    PrimeTester.count(1, 50_000_000));
CompletableFuture<Long> b = CompletableFuture.supplyAsync(() ->
    PrimeTester.count(50_000_001, 100_000_000));

a.thenCombine(b, Long::sum)                        // quando entrambi ci sono,
    somma
 .thenApply(n -> "primi: " + n)                    // trasforma
 .thenAccept(System.out::println);                 // consuma: nessun get, nessun
    blocco
```

`thenApply`, `thenCombine`, `thenCompose`, `exceptionally`, `allOf`: una piccola algebra dei risultati futuri, eseguita per default sul pool comune (capitolo 7), o su un executor passato esplicitamente. È il modello delle librerie di rete asincrone e dei client HTTP moderni (`HttpClient` ha `sendAsync`, che restituisce un `CompletableFuture`). Da provare: riscrivere l'esempio guida con 64 `supplyAsync` e un `allOf`, senza nessun `get` fino alla fine.

10.2 `ForkJoinPool` e `RecursiveTask`: dividi e conquista

Lo stream parallelo del capitolo 7 spacca il lavoro in pezzi e li consegna a un `ForkJoinPool`. Si può fare a mano, per i problemi che si dividono in modo ricorsivo: un `RecursiveTask<V>` ha un metodo `compute()` che, se il problema è piccolo, lo risolve, altrimenti lo divide in due sottocompiti, ne avvia uno con `fork()`, esegue l'altro, e unisce con `join()`.

```
class ContaPrimi extends RecursiveTask<Long> {
    private final long from, to;
    ContaPrimi(long from, long to) { this.from = from; this.to = to; }

    protected Long compute() {
        if (to - from < 100_000)                        // piccolo: calcola
            return PrimeTester.count(from, to);
        long mid = (from + to) / 2;
        ContaPrimi sinistra = new ContaPrimi(from, mid);
        sinistra.fork();                                // uno in parallelo
        long destra = new ContaPrimi(mid + 1, to).compute();   // l'altro qui
        return destra + sinistra.join();
    }
}
long n = ForkJoinPool.commonPool().invoke(new ContaPrimi(1, 100_000_000));
```

Il pool fa *work stealing*: ogni thread ha una coda di sottocompiti, e chi finisce ruba dalla coda degli altri; il bilanciamento del capitolo 4 viene da sé, a qualsiasi profondità. Da provare: misurare `ContaPrimi` contro i 64 compiti del capitolo 4, e vedere che cosa cambia con la soglia a mille e a dieci milioni.

10.3 Il modello di memoria e `VarHandle`

Il capitolo 5 ha dato le garanzie di visibilità che servono nella pratica. Il *Java Memory Model* (capitolo 17 della *Java Language Specification*) le definisce tutte, in termini di *happens-before*: una scrittura è visibile a una lettura se c'è una catena di relazioni che le ordina; altrimenti la lettura può vedere il valore vecchio, quello nuovo, o (per `long` e `double` non `volatile`) metà di ciascuno. Il compilatore e il processore possono riordinare tutto ciò che non è vincolato, ed è per questo che il programma `fermati` non terminava. `VarHandle` JAVA 9 è l'interfaccia di basso livello alle operazioni atomiche e agli ordinamenti di memoria (*plain*, *opaque*, *acquire/release*, *volatile*): è con essa che sono scritte le classi atomiche e le collezioni concorrenti. Non si usa nei programmi normali; si legge per capire che cosa c'è sotto `AtomicInteger`. Da provare: "The Java Memory Model" di Manson, Pugh e Adve, e il capitolo 16 di Goetz.

10.4 Structured concurrency e scoped values

I thread virtuali rendono economico avere un thread per compito, e questo riapre una domanda: come si gestisce un gruppo di compiti *come un'unità*? Se uno fallisce, gli altri vanno annullati; se il chiamante viene interrotto, tutti devono fermarsi; e l'errore deve risalire a chi ha avviato il gruppo. Nelle versioni di Java successive alla 21 la *structured concurrency* (`StructuredTaskScope`) dà esattamente questo: un blocco che avvia sottocompiti e non può terminare finché non sono tutti conclusi, con politiche come "il primo che riesce" o "tutti o nessuno"; e gli *scoped values* sostituiscono `ThreadLocal` per passare un contesto ai sottocompiti senza copiarlo. In Java 21 sono *preview*: non sono nel materiale del corso, ma sono la direzione in cui la libreria sta andando, e chi ha capito executor, `Future` e interruzione li riconosce al primo sguardo.

10.5 Nelle altre dispense

- *Stream, per intero* è il luogo degli stream: la pipeline, la laziness, i collettori. Il capitolo 7 ha mostrato che cosa c'è sotto `parallel()`, e perché i collettori sono il modo giusto di raccogliere da uno stream parallelo: ogni thread riempie il suo contenitore, confinato, e la libreria fonde alla fine.
- *Dal nome alla funzione* spiega le lambda e le variabili catturate: la regola dell'effectively final, che lì sembra una pignoleria, nel capitolo 3 ha mostrato il suo perché. I compiti di questa dispensa sono quasi tutti lambda.
- *Le interfacce in Java moderno*: `Runnable`, `Callable`, `Future`, `ExecutorService` sono interfacce, e le factory di `Executors` sono l'esempio più pulito di programmazione verso un'interfaccia: il codice che consegna compiti non sa, né deve sapere, quanti thread ci sono dietro.
- La dispensa sull'interfaccia grafica riprende la coda degli eventi dal lato dell'architettura: il modello, la vista, il controllore, e dove stanno i thread in un programma MVC. Il capitolo 8 ha dato le regole; lì si vede come un programma intero le rispetta.

10.6 Per approfondire

- B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea, *Java Concurrency in Practice*, Addison-Wesley, 2006. Il libro di riferimento: vecchio nei dettagli di libreria, attualissimo nei principi. I capitoli 2–5 e 10–11 coprono questa dispensa; il 16 è il modello di memoria; il `Memoizer` del capitolo 9 viene dal capitolo 5.
- J. Bloch, *Effective Java*, terza edizione, Addison-Wesley, 2018, capitolo 11 ("Concurrency"): dodici regole, fra cui "preferire gli executor ai thread" e "preferire le utilità di concorrenza a `wait` e `notify`".
- *The Java Language Specification*, capitolo 17 ("Threads and Locks"): il modello di memoria, per chi vuole la definizione di happens-before.
- JEP 444, *Virtual Threads* (Java 21): il documento di progetto dei thread virtuali, leggibile e completo, con le regole d'uso del capitolo 7 nella versione degli autori.
- Oracle, *The Java Tutorials*: "Concurrency" (il percorso di base, con `wait`/`notify` e gli executor) e "Concurrency in Swing" (la coda degli eventi, `SwingWorker`, con gli esempi ufficiali).
- La documentazione Javadoc di `java.util.concurrent`: la descrizione del package è una lettura in sé, con le garanzie di visibilità di ogni classe scritte esplicitamente nella sezione "Memory Consistency Properties".