UNIVERSITÀ DEGLI STUDI DI BRESCIA · unibs.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

L04c

Thread, terza parte

Compiti differiti e periodici, stream paralleli, Swing e la coda degli eventi, thread virtuali



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Oggi

1. **Il tempo**: compiti differiti e periodici
2. **Stream paralleli**, ora che sappiamo che cosa c'è sotto
3. **Thread e Swing**: l'Event Dispatch Thread, `invokeLater`, `SwingWorker`, `javax.swing.Timer`
4. **Thread virtuali**: molti a molti

01

Il tempo



Esecuzione differita

`ScheduledExecutorService`, da `Executors.newScheduledThreadPool(n)`, 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
```

Esecuzione periodica

A FREQUENZA FISSA

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

```
compito da 3 s, periodo 1 s:
inizi a 0, 3, 6, 9
```

Il periodo si conta **dall'inizio** della precedente: un orologio, un campionamento

CON PAUSA FISSA

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

```
compito da 3 s, pausa 1 s:
inizi a 0, 4, 8, 12
```

La pausa si conta **dalla fine** della precedente: una manutenzione che lascia respiro

L’eccezione che ferma tutto

> Un'eccezione non catturata in un compito periodico **ferma le ripetizioni**, in silenzio: lo `ScheduledFuture` la conserva, nessuno lo interroga, il compito smette di girare senza che appaia niente.

```
Runnable task = () -> {
    try {
        faiIlLavoro();
    } catch (RuntimeException e) {
        log.warning("compito fallito: " + e);   // registra, e continua
    }
};
executor.scheduleAtFixedRate(task, 0, 1, TimeUnit.SECONDS);
```

Un compito periodico avvolge il suo corpo in un `try/catch` che registra l’errore e lascia continuare le ripetizioni.

02

Stream paralleli



Gli stream paralleli, ora

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

- la libreria **spacca la sorgente** in pezzi, li consegna al `ForkJoinPool` comune (work stealing, un thread per core) e fonde i risultati
- il programma dei primi in quattro righe, stesso tempo, stesse regole: puro calcolo, elementi indipendenti, **niente stato condiviso**

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 *aspetta* (una lettura di rete dentro una lambda) tiene fermo un thread del pool: con pochi compiti così **tutto il parallelismo della JVM si ferma**
- gli stream paralleli sono per il **calcolo**, non per l'I/O

> Per isolare un lavoro o scegliere il numero di thread: `Executors.newWorkStealingPool(n)` è un `ForkJoinPool` privato, e vi si sottomettono i compiti a mano.

03

Thread e Swing: la coda degli eventi



Un’interfaccia è un ciclo

Un programma a eventi non «fa» qualcosa dall’inizio alla fine: **aspetta**. Un ciclo 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): nel debugger si chiama `AWT-EventQueue-0`
- è lui che esegue ogni listener, ogni `paintComponent`, ogni `Runnable` passato a `invokeLater`

```
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.
```

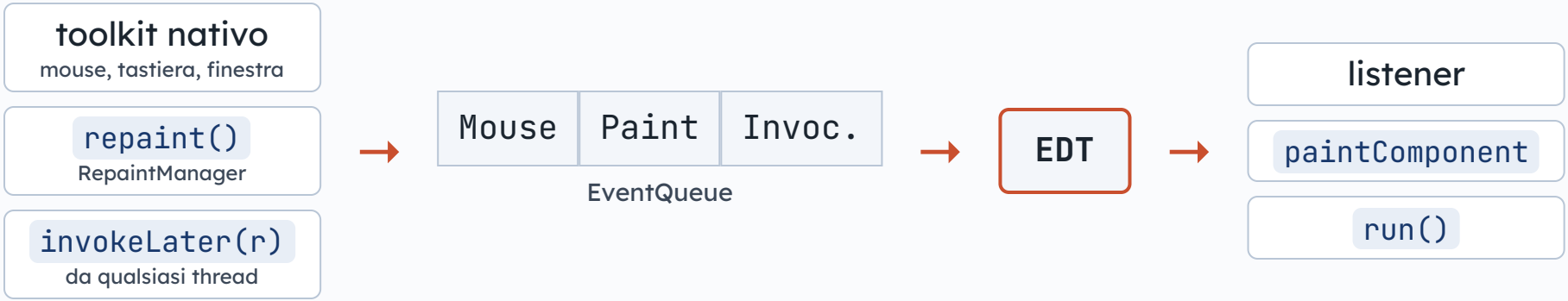
Perché un solo thread

- un’interfaccia è un albero di componenti che si modificano a vicenda di continuo: un testo che cambia fa ridisegnare il contenitore, che ricalcola il layout, che sposta i fratelli
- un lock per componente produrrebbe **deadlock a ogni angolo**, come i due conti; 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 web

> Il prezzo: quel thread **non va mai bloccato**.

Come funziona `EventQueue`

Una coda di `AWTEvent`, con dei produttori e un consumatore.



`pumpEvents`: estrai l'evento in testa, `dispatchEvent`, ricomincia

Produttori e consumatore

- il **toolkit nativo** ha un thread suo (`AppKit Thread` su macOS, `AWT-XAWT` su Linux) che traduce gli eventi del sistema in `MouseEvent`, `KeyEvent`, `WindowEvent` e li accoda
- `repaint()` **non disegna niente**: accoda un `PaintEvent`, e più richieste vicine vengono fuse in una sola
- `EventQueue.invokeLater(r)` incarta un `Runnable` in un `InvocationEvent` e lo accoda: l'**unico modo** con cui un altro thread fa eseguire qualcosa all'EDT
- l'EDT esegue `pumpEvents`: estrai l'evento in testa, `dispatchEvent` al componente giusto, che chiama i suoi listener; ricomincia. **Un solo thread, un evento alla volta**

Chi avvia che cosa

```
public class PrimeFinderApp extends JFrame {
    private final JTextField campoMax = new JTextField("100000000", 10);
    private final JButton btnConta = new JButton("Conta");
    private final JProgressBar barra = new JProgressBar(0, 100);
    private final JLabel lblRisultato = new JLabel("-", SwingConstants.CENTER);
    public PrimeFinderApp() {
        super("Numeri primi");
        // ... i componenti nei contenitori, con un BorderLayout ...
        btnConta.addActionListener(e -> conta());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
    }
    public static void main(String[] args) {
        EventQueue.invokeLater(() -> new PrimeFinderApp().setVisible(true));
    }
}
```

Il `main` non tocca la finestra

- `main` gira sul thread `main`, che per la regola non deve toccare i componenti: **accoda la costruzione** con `invokeLater` e termina subito
- il programma non termina con lui: l'EDT e il thread del toolkit non sono daemon e restano vivi finché esiste una finestra; `EXIT_ON_CLOSE` chiude la JVM all'ultima
- `EventQueue.invokeLater` e `SwingUtilities.invokeLater` sono lo stesso metodo
- `EventQueue.isDispatchThread()` dice se il thread corrente è l'EDT

Il blocco

La prima versione di `conta()` è quella che tutti scrivono:

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

- al click la finestra si blocca per quattordici secondi: non si ridisegna, non risponde, il sistema la segnala come «non risponde»
- non mostra nemmeno «conto…»: l'EDT sta eseguendo il listener e finché non ritorna **non estrae nessun altro evento**, nemmeno il `PaintEvent` del `setText`

Lavoro fuori, rientro dentro

```
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 = new PrimeTester(1, max).call();
        EventQueue.invokeLater(() -> {                  // 3. rientro sull'EDT
            lblRisultato.setText("primi: " + n);
            btnConta.setEnabled(true);
        });
    }).start();
}
```

Il listener ritorna subito; dopo 14 s il thread accoda un `InvocationEvent` e l'EDT aggiorna l'etichetta. `max` si legge **prima**, sull'EDT.

Se non si rientra

Chiamare `setText` direttamente dal thread di lavoro **quasi sempre funziona**, ed è il problema.

- nessun controllo, nessuna eccezione: il componente viene modificato mentre l'EDT magari lo sta disegnando
- una volta ogni mille: un valore vecchio, un testo tagliato, un'eccezione dal profondo di Swing che non si riesce a riprodurre
- un'etichetta aggiornata cento volte al secondo da un thread, prima o poi lo fa vedere

Da qualunque thread

`invokeAndWait(r)` aspetta l'esecuzione di `r`: serve quando il thread di lavoro ha bisogno di una risposta dall'interfaccia. Chiamato **dall'EDT** lancia un `Error`: sarebbe un deadlock certo.

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

Un metodo così si chiama da un listener e da un thread di lavoro senza che chi lo chiama debba sapere dove si trova.

`SwingWorker`: il compito

Lo schema «lavoro fuori, rientro dentro» pronto, con avanzamento, risultati parziali e annullamento: `SwingWorker<T, V>`, `T` il risultato finale, `V` quelli intermedi.

```
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 += new PrimeTester(b * perBlocco + 1, (b + 1) * perBlocco).call();
            setProgress(b + 1);                  // 0..100
        }
        return totale;
    }
```

`SwingWorker`: il rientro

```
    @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: thread di un pool interno a Swing, **non tocca componenti**
- `done()` sull'EDT a fine compito; `get()` qui non blocca: il risultato c'è, o c'è un'eccezione da rilanciare
- `setProgress(0..100)` aggiorna la proprietà `progress`; il listener viene avvisato sull'EDT
- `publish(v...)` dal compito e `process(List<V>)` sull'EDT per i risultati intermedi, consegnati a blocchi
- `cancel(true)` interrompe e fa valere `isCancelled()`: l'interruzione cooperativa con un nome nuovo

`javax.swing.Timer`

Un orologio, un'animazione, un controllo periodico: un compito ripetuto che tocca i componenti. `ScheduledExecutorService` lo eseguirebbe sul thread sbagliato.

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

- consegna i suoi `ActionEvent` **sull'EDT**, dalla stessa coda: il listener tocca i componenti senza rientro, ma dev'essere breve
- `javax.swing.Timer` per il periodico breve sull'interfaccia; `ScheduledExecutorService` per il lavoro vero, con `invokeLater` per mostrare
- `java.util.Timer` ha un thread proprio, **nessun legame con l'EDT**: non si usa

Ricetta: thread e Swing

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**: immutabile, in una coda, o sotto un lock; si legge dall'interfaccia prima di partire, non durante

04

Thread virtuali



Il tetto dei thread di piattaforma

Uno a uno con quelli del sistema operativo.

- un server che dedica un thread a ogni connessione, con diecimila connessioni 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**

Java 21 JAVA 21 aggiunge i **thread virtuali**: thread gestiti dalla JVM, non dal kernel. Oggetti `Thread` a tutti gli effetti, stesso codice, ma leggeri (lo stack vive nell'heap e cresce quanto serve) e numerosi: milioni, non migliaia.

Montati e smontati

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 3; i++) executor.submit(() -> {
        System.out.println("prima: " + Thread.currentThread());
        pausa(100);                                  // si blocca: viene smontato
        System.out.println("dopo:  " + Thread.currentThread());
    });
}                                                    // close(): aspetta tutti
```

```
prima: VirtualThread[#36]/runnable@ForkJoinPool-1-worker-2
prima: VirtualThread[#37]/runnable@ForkJoinPool-1-worker-3
prima: VirtualThread[#34]/runnable@ForkJoinPool-1-worker-1
dopo:  VirtualThread[#36]/runnable@ForkJoinPool-1-worker-3
dopo:  VirtualThread[#34]/runnable@ForkJoinPool-1-worker-1
dopo:  VirtualThread[#37]/runnable@ForkJoinPool-1-worker-2
```

Dopo `@` il *carrier*, il thread di piattaforma che lo esegue: `#36` parte su `worker-2` e riprende su `worker-3`.

Che cosa fa la JVM

- esegue i thread virtuali **montandoli** su un piccolo pool di thread di piattaforma, i *carrier*: un `ForkJoinPool` con un thread per core
- quando un thread virtuale si blocca (`sleep`, rete, una `BlockingQueue`, un `ReentrantLock`) viene **smontato**: il suo stack viene salvato nell'heap e il carrier passa a un altro; al risveglio viene rimontato dove c'è posto
- un modello **molti a molti**: il punto in cui l'astrazione di Java smette di coincidere con quella del sistema operativo

> Uno solo si crea con `Thread.ofVirtual().start(r)` e si aspetta con `join()`, come gli altri. Un thread virtuale è sempre *daemon*: senza `join` (o la `close()` dell'executor) `main` finisce e la JVM non lo aspetta.

Diecimila compiti che dormono un secondo

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 10_000; i++)
        executor.submit(() -> pausa(1000));
}
```

	tempo
lo stesso ciclo con `newFixedThreadPool(200)`	50 s
un thread virtuale per compito	poco più di 1 s

Non è magia: i compiti **aspettano**, e aspettare in diecimila non costa niente se nessuno occupa un thread del kernel per farlo. Cinquanta turni da duecento, contro uno solo.

Regole d’uso

- **Per l’attesa, non per il calcolo**: un thread virtuale che calcola occupa il suo carrier come un thread di piattaforma. Per i numeri primi non cambia niente
- **Non si mettono in pool**: costano così poco che se ne crea uno per compito e lo si butta. `newVirtualThreadPerTaskExecutor` fa esattamente questo
- **Attenzione a** `synchronized` in Java 21: un thread virtuale bloccato *dentro* un `synchronized` resta inchiodato al carrier; nelle sezioni critiche lunghe conviene `ReentrantLock`. Le versioni successive hanno tolto il limite
- **Tutto il resto è uguale**: lock, code, atomiche, executor, interruzione. Il codice scritto per i thread di piattaforma gira sui virtuali senza modifiche

Errori frequenti

- **eccezione in un compito periodico**: le ripetizioni si fermano in silenzio
- **stream paralleli con I/O o stato condiviso**: il pool comune si ferma, o il risultato è sbagliato
- **un listener Swing che calcola**: la finestra congela
- `setText` **dal thread di lavoro**: funziona, finché non funziona
- `java.util.Timer` **con Swing**: il listener gira sul thread sbagliato
- **leggere un campo di testo dal thread di lavoro**: si legge prima, sull'EDT
- **thread virtuali per il calcolo**, o messi in un pool: nessun guadagno

Thread, in due righe

- un compito è un `Runnable` o un `Callable`; un executor lo esegue su pochi thread e restituisce un `Future`; tanti compiti piccoli, pochi thread
- due thread sullo stesso dato sono un conflitto: `synchronized`, atomiche e collezioni concorrenti lo chiudono, **non condividere** lo evita
- Swing ha un thread solo: il lavoro fuori, l'aggiornamento rientra con `invokeLater` o `SwingWorker`
- i thread virtuali sono per chi aspetta: stesso codice, a milioni