UNIVERSITÀ DEGLI STUDI DI BRESCIA · UNIBS.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

L04b

Thread, seconda parte

`Callable` e `Future`, conflitti, `synchronized`, lock e atomiche, collezioni concorrenti



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Oggi

1. **Callable e Future**: un compito che calcola, una promessa di risultato
2. **Contare i primi fino a 10^8**: quanti thread, quanti compiti
3. **Quando i thread condividono**: conflitti, visibilità, `synchronized`, `wait` e `notifyAll`, deadlock
4. `java.util.concurrent`: lock espliciti, semafori, variabili atomiche, collezioni concorrenti

01

Callable e Future



Da `Runnable` a `Callable`

`Runnable.run()` non restituisce niente e non lancia eccezioni checked: va bene per un compito che **agisce**, non per uno che **calcola** un risultato.

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

Il compito guida: contare i numeri primi in un intervallo. Non è utile in sé, ma è perfetto per i thread: **puro calcolo**, si divide in pezzi **indipendenti**, e il lavoro **non è uniforme** (i numeri grandi costano più dei piccoli).

PrimeTester

```java
public class PrimeTester implements Callable<Long> {
    private final long start, end;
    public PrimeTester(long start, long end) {
        this.start = start; this.end = end;
    }
    public Long call() {
        long n = 0;
        for (long i = start; i <= end; i++)
            if (isPrime(i)) n++;
        return n;
    }
    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;
    }
}
```

`Future`: un risultato che arriverà

`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(new PrimeTester(1, 50_000_000));
Future<Long> fb = executor.submit(new PrimeTester(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
```

Il resto di `Future`

`get()` è il punto di **sincronizzazione**: blocca il chiamante finché il risultato non c'è, poi lo restituisce. Il resto serve a non bloccarsi alla cieca.

- `isDone()` risponde subito
- `get(t, unit)` aspetta al massimo `t`, poi lancia `TimeoutException`
- `cancel(true)` interrompe il compito se è in corso, lo toglie dalla coda se non è partito
- se `call()` lancia, `get()` rilancia una `ExecutionException` che **incarta** l'originale: `getCause()` la recupera
- `get()` dichiara anche `InterruptedException`: è un'attesa come `sleep` e `join`, vale la stessa regola

`invokeAll` e `invokeAny`

Per una collezione di compiti ci sono due scorciatoie.

```
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
```

- `invokeAll` li esegue tutti e restituisce la lista dei `Future`, **già completati**
- `invokeAny` restituisce il risultato del **primo** che termina e annulla gli altri: tre server che rispondono alla stessa domanda, si prende il più veloce

02

Contare i primi fino a 10^8



Il programma

`nTask` pezzi a un pool di `nThread` thread, poi la somma. `Chrono` misura i millisecondi.

```
long max = 100_000_000L;
int nTask = 64;
int nThread = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(nThread);
List<Future<Long>> risultati = new ArrayList<>();
Chrono chrono = new Chrono();
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(new PrimeTester(da, a)));
}
long nPrimi = 0;
for (Future<Long> f : risultati)
    nPrimi += f.get();
System.out.printf("%d primi in %d ms%n", nPrimi, chrono.millis());
executor.shutdown();
```

Le misure

Un portatile con 16 core (12 veloci e 4 efficienti), al variare di `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**: si alternano sugli stessi core

La legge di Amdahl

Se una frazione p del lavoro è parallelizzabile e $1 - p$ no, con n core l'accelerazione è al massimo

$$\frac{1}{(1-p) + p/n} \xrightarrow{n \to \infty} \frac{1}{1-p}$$

- la parte sequenziale (dividere l'intervallo, avviare i thread, sommare i risultati), per piccola che sia, **mette un tetto**
- il limite non è Java: è il programma

Perché 64 compiti e non 16?

Il lavoro **non è uniforme**: l'ultimo pezzo dell'intervallo costa più del primo.

- 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

```
Tanti compiti piccoli, pochi thread.
```

> `CompletableFuture` è un `Future` che si **compone**: `thenApply`, `thenCombine`, `thenAccept`, senza mai chiamare `get()`. È il modello delle librerie di rete asincrone; per il calcolo basta `Future`.

Finora, compiti indipendenti

Ogni compito lavorava sui **suoi** dati e restituiva un risultato: i thread non si sono mai toccati.

```
static int contatore = 0;

executor.submit(() -> { for (int i = 0; i < 10_000; i++) contatore++; });
executor.submit(() -> { for (int i = 0; i < 10_000; i++) contatore++; });
```

- 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
- `contatore` qui non vale 20 000, e non vale sempre lo stesso numero

> Da qui in poi: perché succede, e come si evita.

03

Quando i thread condividono



Il contatore che perde

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

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Counter counter = new Counter();
for (int i = 0; i < 10_000; i++)
    executor.submit(counter::inc);
executor.shutdown();
executor.awaitTermination(1, TimeUnit.MINUTES);
System.out.println(counter.count);
```

```
9994
```

`count++` è tre istruzioni

```
$ javap -c Counter
       0: aload_0
       1: dup
       2: getfield      #7     // leggi count
       5: iconst_1
       6: iadd                 // somma 1
       7: putfield      #7     // scrivi count
```

- fra la lettura e la scrittura lo scheduler **può sospendere il thread**
- `A` e `B` leggono entrambi 41, sommano, scrivono 42: un incremento è perso

La corsa, passo per passo

Due thread, un `Counter` a 41, un `inc()` ciascuno:

passo	thread A	thread B	`count`
1	legge 41		41
2		legge 41	41
3	somma: 42		41
4		somma: 42	41
5	scrive 42		42
6		scrive 42	42

Due incrementi, uno perso. Lo scheduler può sospendere A dopo il passo 1 per un tempo qualsiasi; B non ha modo di sapere che A è a metà.

Conflitto e sezione critica

```
Conflitto (race condition): il risultato dipende dall'ordine,
non controllato, in cui più thread accedono a uno stato condiviso.

Sezione critica: il tratto di codice che accede allo stato condiviso
e che deve essere eseguito da un thread alla volta per restare corretto.
```

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.

Visibilità: il ciclo che non termina

Ogni core ha le sue cache, e la JIT ottimizza come se il thread fosse solo.

```
class Lavoro {
    boolean fermati = false;
    void esegui() {
        long k = 0;
        while (!fermati) k++;              // la JIT legge fermati una volta sola
    }
}
```

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

`volatile`

Il thread di lavoro non rilegge mai `fermati` dalla memoria: nulla gli dice che un altro thread possa cambiarla.

- `volatile boolean fermati`: la variabile viene **sempre letta e scritta in memoria**, mai tenuta in un registro; una scrittura è visibile a tutte le letture successive, di qualsiasi thread
- `volatile` per una **bandiera** che un thread scrive e altri leggono

> `volatile` non rende atomico niente: `volatile int count` con `count++` perde incrementi esattamente come prima. Leggi, somma, scrivi restano tre operazioni.

> Il *Java Memory Model* elenca le garanzie (*happens-before*): `start()`, `join` e `get`, `synchronized`, `volatile`, `submit`. Fuori da queste, nessun ordine.

Tre problemi, non uno

- **atomicità**: leggi, modifica, scrivi devono sembrare **un'operazione sola** agli altri thread
- **visibilità**: una scrittura deve **arrivare** agli altri thread, non restare in una cache o in un registro
- **ordine**: compilatore e processore **riordinano** le istruzioni; solo le garanzie *happens-before* fissano un prima e un dopo fra thread diversi

```
synchronized, i lock e le variabili atomiche risolvono tutti e tre.
volatile risolve visibilità e ordine, non l'atomicità.
```

`synchronized`

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

```
10000
```

- ogni oggetto possiede un **monitor** (lock intrinseco): una serratura che un solo thread alla volta può tenere
- un metodo `synchronized` prende il monitor di `this` all'ingresso e lo rilascia all'uscita, **anche per eccezione**
- chi trova il monitor occupato passa in `BLOCKED` e aspetta

Il blocco `synchronized`

La forma a blocco sceglie l'oggetto e restringe la sezione critica al minimo:

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

- **si protegge lo stato, non il codice**: il monitor è dell'oggetto; due `Counter` diversi non si bloccano, due metodi diversi sullo stesso oggetto sì. *Tutti* gli accessi, anche le letture, passano dallo stesso monitor
- **il monitor è rientrante**: chi lo tiene entra in altri metodi `synchronized` dello stesso oggetto senza bloccarsi

Che cosa protegge il monitor

`Counter` con `synchronized void inc()`, due istanze `c1` e `c2`:

thread A	thread B	che succede
`c1.inc()`	`c2.inc()`	monitor diversi: nessuna attesa, nessun conflitto
`c1.inc()`	`c1.inc()`	stesso monitor: B aspetta che A esca
`c1.inc()`	`c1.count`	lettura **fuori** dal monitor: B può vedere il valore vecchio
`c1.inc()`	`synchronized (c1) { ... }`	stesso monitor, anche da fuori: B aspetta

> Il monitor vale solo per chi lo prende: `synchronized` su `inc` non protegge `count` da chi lo legge direttamente. Per questo i campi condivisi sono privati e si leggono da metodi sincronizzati.

Corta, e su quale oggetto

> Dentro il monitor ci sta ciò che tocca lo stato condiviso; il calcolo sta fuori. Un `synchronized` attorno a un metodo lungo **trasforma otto thread in uno**, con in più il costo del lock.

`static synchronized` usa il monitor di `Classe.class`: protegge i campi statici, indipendente dai monitor delle istanze.

Sincronizzare su `this` ha un difetto: chiunque abbia un riferimento può fare `synchronized (counter)` e interferire. Chi espone il proprio stato a molti usa un **lock privato**:

```
private final Object lock = new Object();
void inc() { synchronized (lock) { count++; } }
```

Coordinarsi: `wait` e `notifyAll`

Il monitor serve anche ad **aspettare**: produttore e consumatore su un buffer a capienza fissa.

```
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;
    }
}
```

Produttore e consumatore, in sequenza

`Buffer` di capienza 1, un produttore e un consumatore:

passo	produttore	consumatore	coda
1	`metti("a")`: aggiunge, `notifyAll`		`[a]`
2	`metti("b")`: piena, `wait` **rilascia il monitor**		`[a]`
3		`prendi()`: toglie `a`, `notifyAll`	`[]`
4	sveglio: **ricontrolla**, aggiunge `b`, `notifyAll`		`[b]`
5		`prendi()`: toglie `b`	`[]`
6		`prendi()`: vuota, `wait`	`[]`

Al passo 4 il `while` ricontrolla: se un secondo produttore fosse entrato prima, la coda sarebbe di nuovo piena e il thread tornerebbe a dormire.

`while`, non `if`

- `wait()` si chiama **tenendo il monitor**: lo rilascia, mette il thread in `WAITING`, e al risveglio lo riacquisisce prima di ritornare
- `notifyAll()` sveglia tutti i thread in attesa su quel monitor; `notify()` ne sveglia uno a caso, quasi sempre la scelta sbagliata
- fra la sveglia e il riacquisto del monitor un altro thread può aver cambiato di nuovo lo stato, e la JVM ammette **risvegli spuri**: si ricontrolla sempre la condizione

> Nella pratica questo schema è già scritto, e meglio, nelle code bloccanti di `java.util.concurrent`. Va conosciuto perché è ciò che sta sotto.

`BLOCKED` e `WAITING`

stato	come ci si arriva	il monitor	come si esce
`BLOCKED`	entrando in un `synchronized` occupato	lo tiene un altro	quando si libera
`WAITING`	`wait()` dentro il `synchronized`	**rilasciato** da `wait`	`notifyAll`, poi si riprende il monitor
`TIMED_WAITING`	`sleep(ms)`, `wait(ms)`	`sleep` lo **tiene**, se lo aveva	allo scadere

> `sleep` dentro un `synchronized` tiene il monitor per tutta la pausa: gli altri restano `BLOCKED`. `wait` è l'unico modo di aspettare **lasciando entrare** gli altri.

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 insieme `trasferisci(x, y, 10)` e `trasferisci(y, x, 20)`
- il primo tiene `x` e aspetta `y`; il secondo tiene `y` e aspetta `x`. **Nessuno cederà mai**

Il deadlock, passo per passo

passo	T1: `trasferisci(x, y, 10)`	T2: `trasferisci(y, x, 20)`
1	prende il monitor di `x`	
2		prende il monitor di `y`
3	chiede `y`: `BLOCKED`	
4		chiede `x`: `BLOCKED`

- da qui nessuno avanza: ciascuno aspetta un monitor che solo l'altro può rilasciare, e nessuno dei due lo rilascerà
- con l'ordine fisso, al passo 2 anche T2 chiederebbe `x`: aspetterebbe T1, che finisce il trasferimento e libera tutto

Rompere il ciclo

Delle quattro condizioni si rompe il **ciclo di attese**: lock sempre **nello stesso ordine**.

```
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);
        }
    }
}
```

> Parenti: **starvation** (la risorsa la prendono sempre altri) e **livelock** (si evitano senza avanzare).

Thread safety per costruzione

Il modo migliore di proteggere uno stato condiviso è **non averne**.

- **Immutabilità**: un oggetto che non cambia dopo la costruzione si legge da qualsiasi numero di thread senza lock: `String`, gli involucri, `List.of(...)`. Un «cambiamento» produce un oggetto nuovo
- **Confinamento**: un dato usato da un solo thread non è condiviso. Le variabili locali lo sono per natura; `PrimeTester` conta nel suo campo e condivide una volta sola, attraverso il `Future`
- **Oggetti thread-safe dichiarati**: lo dicono nella documentazione; chi non lo dice, non lo è. Non lo sono `ArrayList`, `HashMap`, `StringBuilder`; lo sono `ConcurrentHashMap`, `AtomicInteger`, le code di `java.util.concurrent`

È thread-safe?

Le domande da farsi davanti a una classe, nell'ordine:

1. **Chi la tocca?** Un thread solo: è confinata, non c'è niente da fare
2. **Cambia dopo la costruzione?** No: è immutabile, non c'è niente da fare
3. **Lo dice la documentazione?** `ConcurrentHashMap` sì, `ArrayList` no, la vostra classe lo dice solo se l'avete fatto voi
4. **Ogni accesso allo stato**, letture comprese, passa dallo stesso lock?
5. **Le operazioni composte** (leggi poi scrivi, controlla poi agisci) stanno dentro un solo blocco?
6. **Quanti lock** prende insieme, e sempre nello stesso ordine?

04

Gli strumenti di `java.util.concurrent`



Lock espliciti: `ReentrantLock`

```
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
    }
}
```

- stesso comportamento di `synchronized`, con il rilascio **a carico del programmatore**: il `finally` è obbligatorio

Lettori e scrittori: `ReadWriteLock`

Molti thread possono **leggere** insieme senza danno; il problema nasce quando qualcuno **scrive**. Due lock coordinati: lettura condivisa, scrittura esclusiva.

```
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
```

Risorse a posti limitati: `Semaphore`

Un lock ammette un thread; un semaforo ne ammette `n`: `acquire()` prende un permesso o aspetta, `release()` lo restituisce.

```
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(500
        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);
```

Variabili atomiche

Per un contatore, una bandiera, un riferimento, il lock è sproporzionato. `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference<V>`:

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

- 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**

Più di un incremento

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

- la lambda di `updateAndGet` può essere eseguita **più volte**, se nel frattempo un altro thread ha cambiato il valore: deve essere pura
- tanti thread sullo **stesso** contatore in continuazione: il compare-and-swap fallisce spesso e riprova spesso
- `LongAdder` e `LongAccumulator` tengono **una cella per thread** e sommano solo a `sum()`: più memoria, lettura meno precisa durante gli aggiornamenti, ma scalano

Misurare

Dieci milioni di incrementi da 16 thread, stesso portatile dell'altra volta:

`synchronized`	`AtomicInteger`	`LongAdder`
74 ms	391 ms	66 ms

Sotto contesa altissima il compare-and-swap ha perso dal lock; su un altro processore può rovesciarsi. **La sola regola seria è misurare.**

Collezioni concorrenti

Le collezioni di `java.util` non sono thread-safe. `Collections.synchronizedList` avvolge ogni metodo in un `synchronized`, ma **l'iterazione non è protetta**: `ConcurrentModificationException`.

```
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);
```

- `ConcurrentHashMap` ha segmenti con lock indipendenti: scritture su chiavi diverse non si bloccano, le letture non prendono lock
- `merge`, `compute`, `computeIfAbsent` sono **atomici**: la coppia «leggi, poi scrivi» qui è una chiamata sola

`BlockingQueue`

Il produttore/consumatore di prima, già scritto: `put` aspetta se la coda è piena, `take` se è vuota; `offer` e `poll` con timeout.

```
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;
});
```

Quale strumento

Il problema	Lo strumento
una bandiera: uno scrive, altri leggono	`volatile boolean` o `AtomicBoolean`
un contatore, un riferimento	`AtomicInteger`, `AtomicReference`; `LongAdder` sotto contesa
più campi da aggiornare insieme	`synchronized` su un lock privato, o `ReentrantLock`
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`
nessuno stato condiviso	la scelta migliore: immutabilità e confinamento

Errori frequenti

- `submit` **senza mai leggere il** `Future`: le eccezioni spariscono
- **un compito per thread** con lavoro non uniforme: chi finisce prima aspetta
- `count++` **su un campo condiviso** senza protezione: funziona quasi sempre
- `volatile` **al posto di un lock**: visibilità sì, atomicità no
- `synchronized` **attorno al calcolo** invece che attorno all'accesso: otto thread diventano uno
- `if` **al posto di** `while` attorno a `wait()`
- **lock presi in ordine diverso** in due punti del programma: deadlock
- `ArrayList` **e** `HashMap` **condivise**: non sono thread-safe, e non lo dicono

Compiti indipendenti, compiti condivisi

- un compito che **calcola** restituisce un `Future`; molti compiti piccoli su pochi thread, e l'accelerazione ha un tetto
- due thread sullo **stesso dato** sono un conflitto: `synchronized`, lock e atomiche lo chiudono, immutabilità e confinamento lo evitano
- per i casi ricorrenti la libreria ha già lo strumento: code bloccanti, mappe concorrenti, semafori

> Prossima lezione: compiti differiti e periodici, gli stream paralleli visti da sotto, i thread e Swing, i thread virtuali.