

PCD2018 - 15

DATI THREAD-SAFE

PROBLEMA

Qual'è il problema che incontriamo ponendoci nel quadrante dati mutabili/stato condiviso?

pcd2018.safe.AdderTest

```
public class Adder {  
    int target = 0;  
  
    public void add() {  
        ...  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

I thread sono uguali e così definiti:

```
var t1 = new Thread(() -> {  
    for (int i = 0; i < 100000; i++) {  
        target += 1;  
    }  
});  
  
var t2 = ...  
var t3 = ...
```

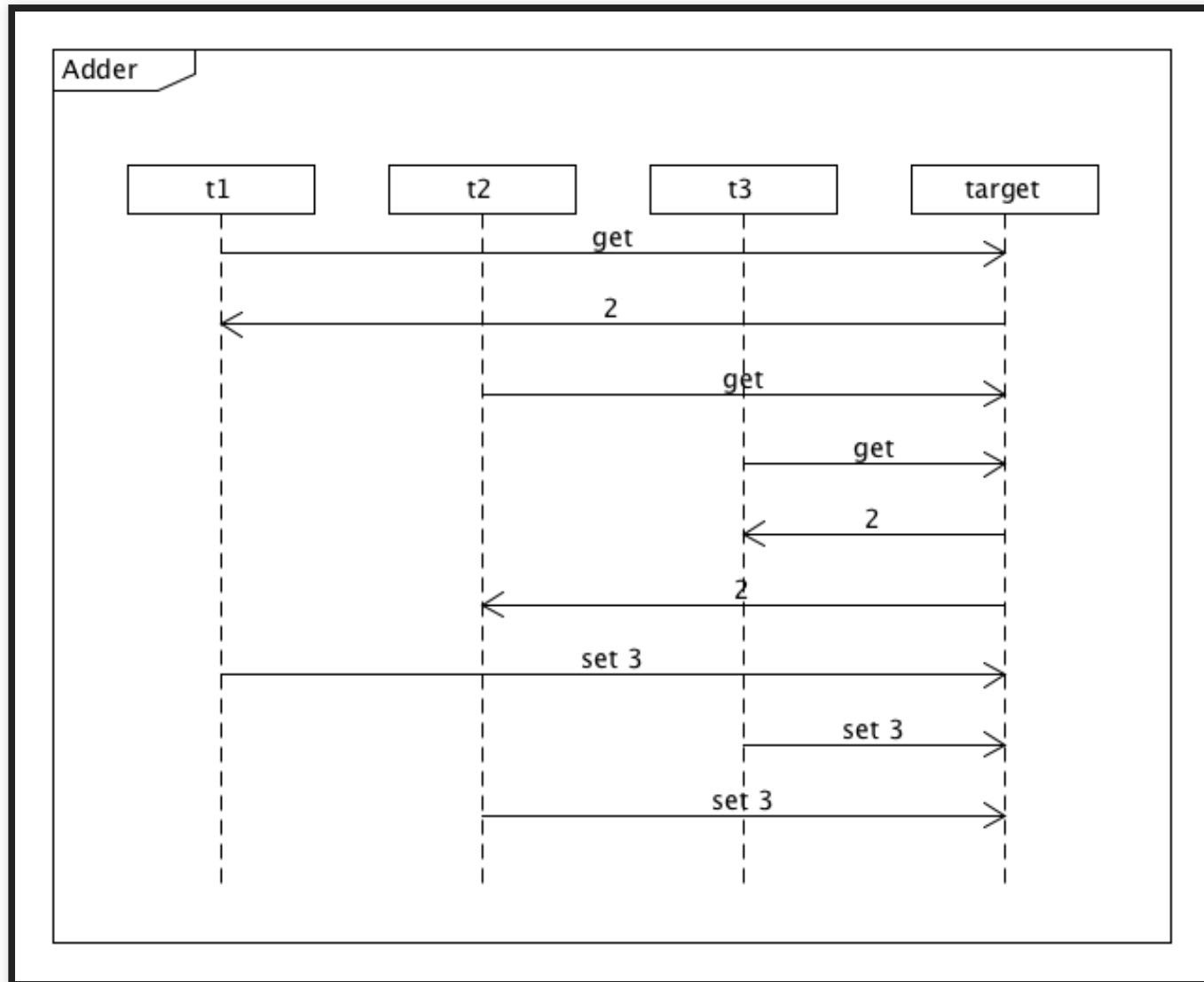
Speaker notes

allo scopo di questo test, è più semplice ripetere la definizione per ciascun thread perché l'obiettivo è riferire alla stessa variabile condivisa. Le tre lambda scritte in questo modo sono closures (cioè, si chiudono) sulla variabile `target` lessicalmente identica.

Test:

```
@Test
void test() throws InterruptedException {
    var adder = new Adder();
    adder.add();
    Thread.sleep(1000);
    assertEquals(300000, adder.target);
}
```

Cosa fallisce?



La concorrenza ci porta al non determinismo.

Il problema di condividere l'accesso a dati non è quindi solo quello del deadlock, ma anche quello della correttezza del risultato.

Speaker notes

va notato come già i passaggi di compilazione, interpretazione e JITting introducono una indeterminatezza sul reale ordine di esecuzione delle istruzioni, anche nel caso del singolo thread. Inoltre, lo stesso hardware spesso riordina l'esecuzione delle istruzioni per ottimizzare l'uso delle risorse. Quindi, in linea generale non ci si può basare sul codice sorgente per interpretare alcuni dettagli del comportamento di un programma, tantomeno in ambiente concorrente.

Una struttura dati non thread-safe non consente a più thread di operare contemporaneamente.

- nel migliore dei casi lancia una `java.util.ConcurrentModificationException`
- nel caso intermedio lo stato diventa inconsistente
- nel peggiore dei casi ottengo un'altra eccezione

Speaker notes

per es. in determinati (rari) casi `HashMap#put` è in grado di lanciare un `IndexOutOfBoundsException`.

pcd2018.safe.ListTraverser

```
list.iterator().forEachRemaining(el -> {  
    System.out.println(el);  
    try {  
        Thread.sleep(250);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
});
```

Speaker notes

poniamo di definire in questo modo un `Runnable` che percorre una lista.

pcd2018.safe.ListUpdater

```
try {  
    Thread.sleep(300);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
list.add("d");
```

Speaker notes

in questo modo invece definiamo un `Runnable` che aggiunge un elemento ad una lista.

pcd2018.safe.ListTest

```
List<String> list = new ArrayList<String>();  
list.add("a");  
list.add("b");  
list.add("c");  
var t1 = new Thread(new ListTraverser(list));  
var t2 = new Thread(new ListUpdater(list));  
t1.start();  
t2.start();  
Thread.sleep(1000);  
assertEquals(4, list.size());
```

Speaker notes

ArrayList dichiara ([nella documentazione](#)) che il suo iteratore lancia una `java.util.ConcurrentModificationException` se la collezione viene modificata durante l'attraversamento

Approfondimenti:

A Crash Course in Modern Hardware by Cliff Click

<https://www.youtube.com/watch?v=OFgxAFdxYAQ>

Adventures with concurrent programming in Java

<https://www.youtube.com/watch?v=929OrlvbW18>

In conclusione:

Se abbiamo la necessità di condividere dati fra più thread, abbiamo bisogno di strutture dati thread-safe.

ATOMIC VARIABLES

Se il nostro caso d'uso riguarda semplicemente l'incremento di un contatore, una possibile soluzione sono le classi del package `java.concurrent.atomic`

tipo	singolo
Integer	AtomicInteger
Long	AtomicLong
Object	AtomicReference

tipo	array
------	-------

Integer	AtomicIntegerArray
---------	--------------------

Long	AtomicLongArray
------	-----------------

Object	AtomicReferenceArray
--------	----------------------

Queste classi garantiscono:

- che la modifica del valore che contengono sia "atomica" e thread-safe
- che la modifica (quasi sempre) non blocchi il thread che la sta eseguendo

Quasi sempre: la funzionalità richiede la disponibilità del supporto dell'hardware attraverso istruzioni CAS:

Compare-and-swap

In mancanza di queste l'implementazione ripiega su metodi più convenzionali (meno efficienti, che bloccano il thread).

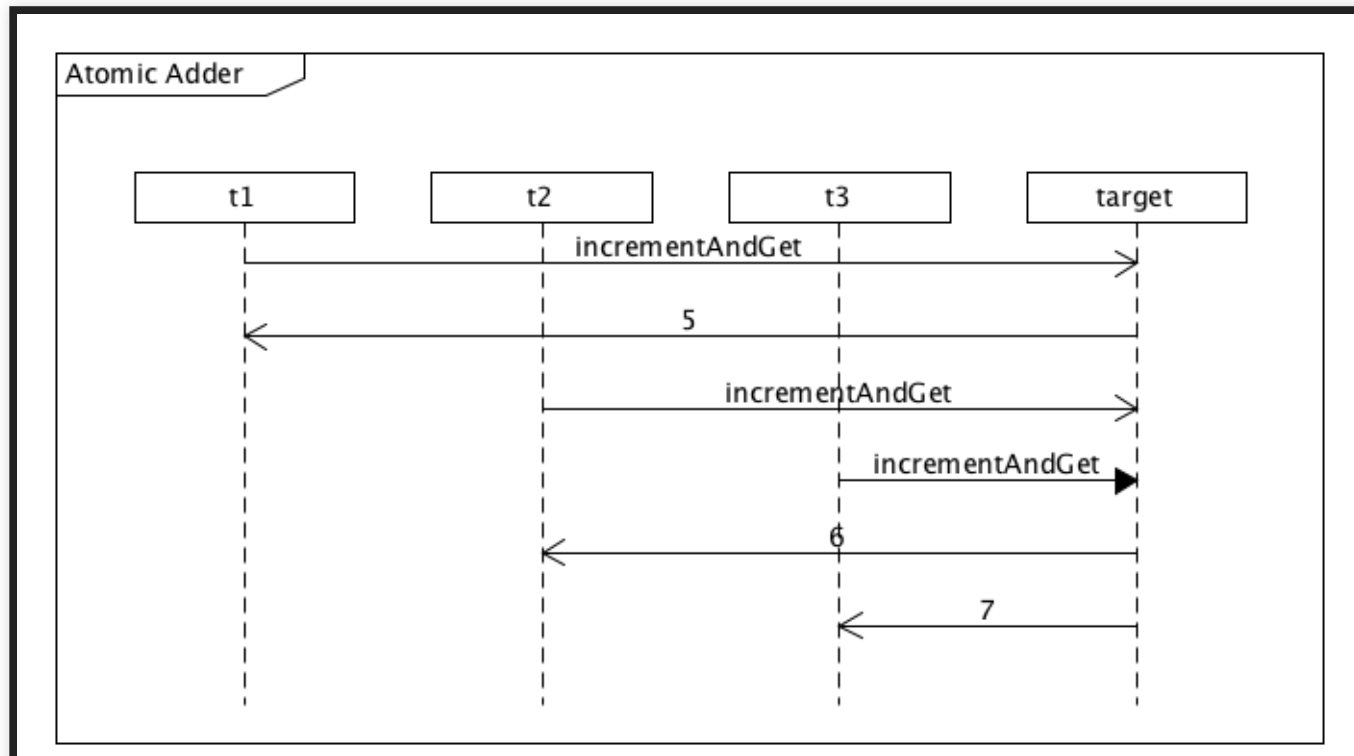
pcd2018.safe.AtomicAdder

```
public class AtomicAdder {  
    public AtomicInteger target = new AtomicInteger(0);  
  
    public void add() {  
        ...  
    }  
}
```

```
var t1 = new Thread(() -> {  
    for (int i = 0; i < 100000; i++) {  
        target.incrementAndGet();  
    }  
});
```

Speaker notes

ora l'istruzione appare anche nel sorgente come unitaria: incrementa e ritorna il nuovo valore. La documentazione spiega le garanzie di atomicità che sono fornite.



Speaker notes

la terza chiamata di `incrementAndGet` potenzialmente potrebbe rimanere bloccata, ma nella pratica la risposta è talmente veloce che è estremamente difficile (praticamente impossibile, se l'esecuzione viene implementata con un solo opcode) che le due `incrementAndGet` si accavallino. Qualora anche avvenisse, a livello hardware il costo di sincronizzazione sarebbe molto basso. Confrontate con il diagramma della prima lezione.

```
/**
 * Atomically sets the value to the given updated
 * value if the current value == the expected value.
 *
 * @param expect the expected value
 * @param update the new value
 * @return true if successful. False return indicates
 * that the actual value was not equal to the expected value.
 */
public final boolean compareAndSet(long expect, long update)
```

```
/**
 * An AtomicMarkableReference maintains an object
 * reference along with a mark bit, that can be
 * updated atomically.
 *
 * Implementation note : This implementation maintains
 * markable references by creating internal objects
 * representing "boxed" [reference, boolean] pairs.
 */
public class AtomicMarkableReference<V>;
```

```
/**
 * An AtomicStampedReference maintains an object
 * reference along with an integer "stamp", that
 * can be updated atomically.
 *
 * Implementation note : This implementation maintains
 * stamped references by creating internal objects
 * representing "boxed" [reference, integer] pairs.
 */
public class AtomicStampedReference<V>;
```

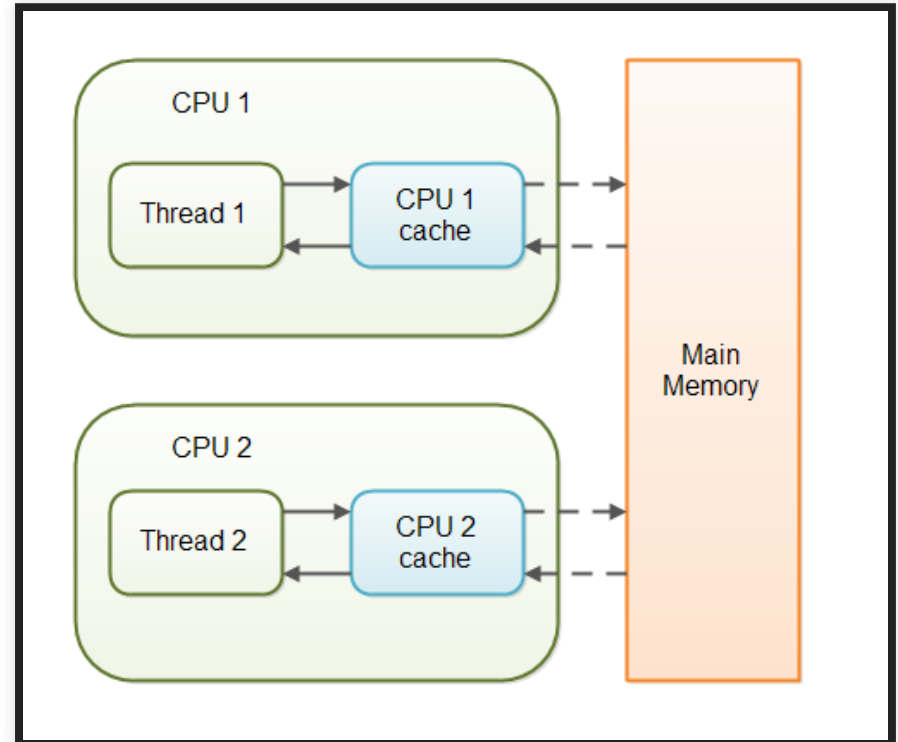
VOLATILE

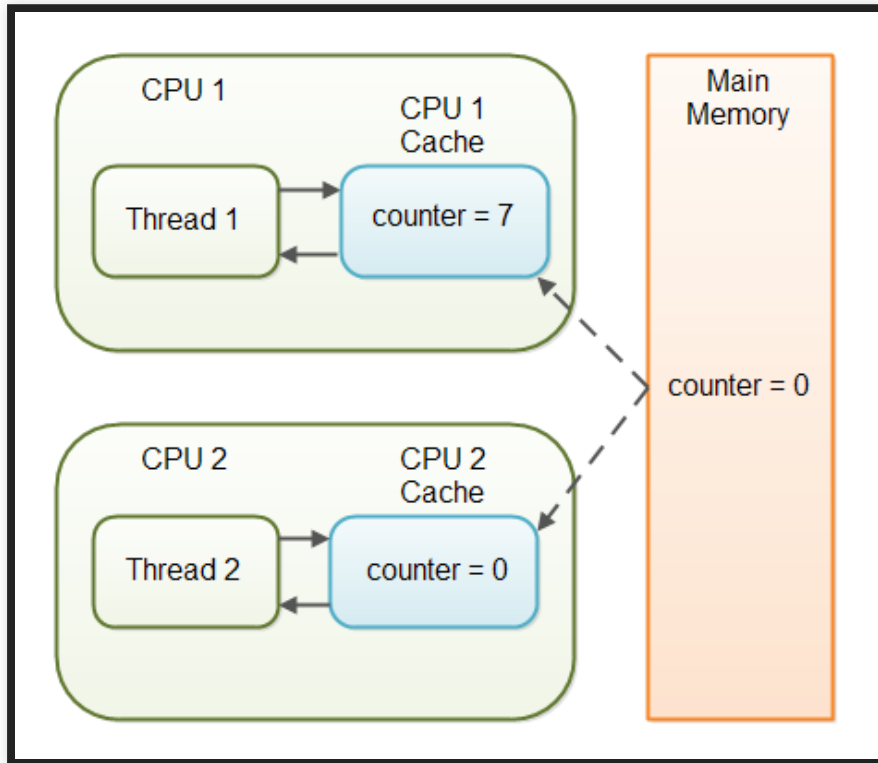
La parola chiave `volatile` indica che una variabile deve sempre essere letta "dalla memoria principale" e non da cache intermedie.

Speaker notes

Dobbiamo parlare di questa parola chiave, anche se il suo uso è molto particolare e delicato. Ne parliamo perché è necessario saperne l'esistenza, ma va usata con cautela.

In un'architettura hardware ormai comune, più CPU colloquiano con la stessa memoria principale. Ognuna, ha una cache locale per velocizzare l'esecuzione del codice e l'interazione con i dati.



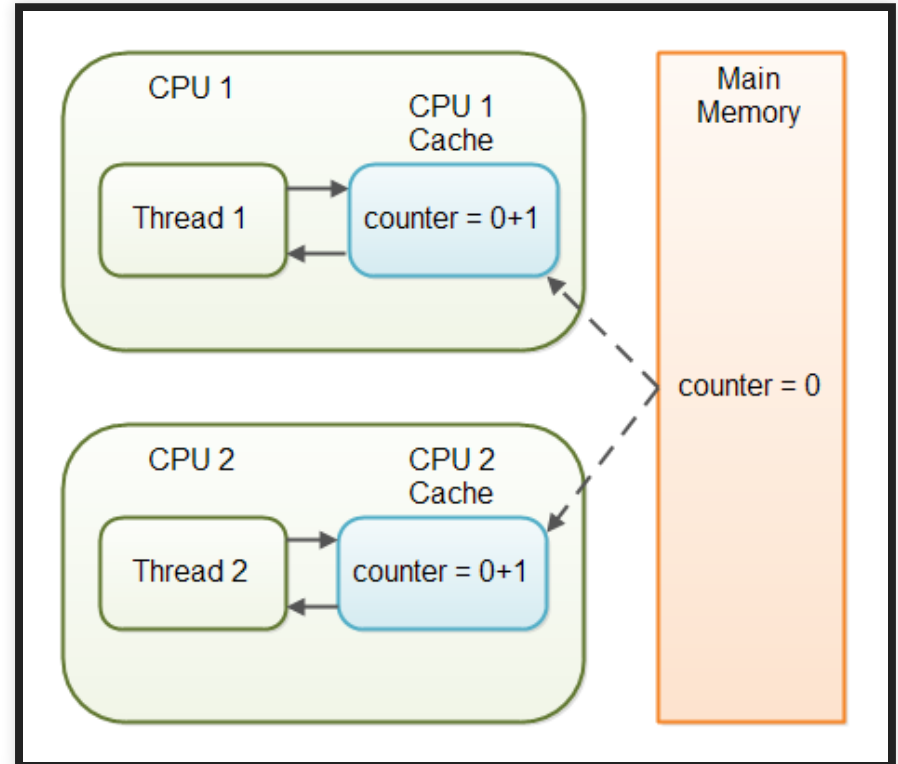


Questo significa però che thread diversi che sono stati smistati su CPU diverse vedono valori differenti, provenienti dalla cache, della stessa variabile. E' il problema della "visibilità del valore scritto"

Speaker notes

e ha origine nelle ottimizzazioni e nei compromessi necessari per usare un'architettura multiprocessore e multithread.

Una variabile dichiarata "volatile" viene sempre letta dalla memoria principale in modo da garantire la visibilità dell'ultima scrittura.



Secondo la specifica della JVM, `volatile` stabilisce una relazione di `happens-before` in determinati casi di accesso alla variabile.

Una *relazione di happens – before* è una garanzia forte fornita dal compilatore riguardo l'ordinamento dell'esecuzione delle istruzioni espresse dal codice.

Speaker notes

A causa della grande differenza di prestazioni fra il canale della memoria e la CPU, per motivi di efficienza il compilatore riordina aggressivamente l'ordine di esecuzione

Per approfondire *happens-before*

What Came First: The Ordering of Events in Systems

<https://www.infoq.com/presentations/events-riak-go>

Cliff Click's Blog

<http://cliffc.org/blog/>

<https://itunes.apple.com/us/podcast/id1286422919>

Stabilire una relazione di happens-before è, ovviamente, costoso: richiede il supporto dell'hardware e limita la capacità del compilatore di ottimizzare il codice riordinandone le istruzioni.

La Java Language Specification 8 al capitolo 17.4.4 "Happens-before Order" definisce che:

*A write to a volatile field (§8.3.1.4)
happens-before every subsequent read
of that field.*

Speaker notes

Si tratta quindi di una primitiva di sincronizzazione a tutti gli effetti <http://cr.openjdk.java.net/~iris/se/11/latestSpec/java-se-11-jvms-draft-diffs.pdf> <https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4>

Tuttavia...

pcd2018.safe.VolatileTest

```
class VolatileHolder {  
    volatile int counter = 0;  
}
```



```
@Test
public void volatileCounter() {
    VolatileHolder holder = new VolatileHolder();

    ExecutorService executor = Executors.newFixedThreadPool(4);
    IntStream.range(0, 10000).forEach(i ->
        executor.submit(() -> holder.counter++));
    awaitDone(executor);

    assertEquals(10000, holder.counter);
}
```

Speaker notes

`awaitDone(ExecutorService)` attende che il servizio abbia esaurito tutti i task accodati.

La garanzia fornita da `volatile` è utile alla
correttezza del programma solo se

*nessun thread scrive nella variabile volatile un valore
dipendente dal valore che ha appena letto dalla stessa
variabile*

In un certo senso le classi `atomic` sono una generalizzazione e semplificazione di alcuni usi di `volatile`.

Usate con cautela questo costrutto.

Trovate le immagini ed un approfondimento in questo articolo:

<http://tutorials.jenkov.com/java-concurrency/volatile.html>

CONCURRENT DATA STRUCTURES

Nel package `java.util.concurrent` si trovano le versioni ottimizzate per la concorrenza di molte delle collezioni più comuni.

In generale, sono pensate per essere più efficienti della versione sincronizzata delle loro controparti, per es.

```
Collections.synchronizedMap(new  
    HashMap( ) )
```

Speaker notes

rispetto alla sincronizzazione della completa struttura dati, queste classi limitano la sincronizzazione alle sole sezioni critiche, in modo da ottenere complessivamente una maggiore efficienza.

CONCURRENTMAP

L'interfaccia `ConcurrentMap` aggiunge a `Map` garanzie di atomicità ed ordinamento delle operazioni.

```
/**  
 * A Map providing thread safety and atomicity  
 * guarantees.  
 */  
public interface ConcurrentMap<K,V>  
    extends Map<K,V>
```



```
/**  
 * If the specified key is not already associated with a  
 * value, associate it with the given value.  
 * The action is performed atomically.  
 */  
V putIfAbsent(K key, V Value)
```

```
/**  
 * Replaces the entry for a key only if currently mapped  
 * to some value.  
 * The action is performed atomically.  
 */  
V replace(K key, V Value)
```

```
/**  
 * Replaces the entry for a key only if currently mapped  
 * to a given value.  
 * The action is performed atomically.  
 */  
V replace(K key, V oldValue, V newValue)
```

Analogamente `ConcurrentNavigableMap` con
`NavigableMap`.

Alcune implementazioni, come `ConcurrentHashMap`, offrono metodi come `reduce`, `search` e `foreach` che possono operare su tutte le chiavi suddividendo autonomamente il lavoro in più thread.

```
/**
 * Returns the result of accumulating the given
 * transformation of all (key, value) pairs using the
 * given reducer to combine values, or null if none.
 *
 * @param the elements needed to switch to parallel
 * @param the transformation for an element
 * @param a commutative associative combining function
 */
public <U> U reduce(long parallelismThreshold,
    BiFunction<? super K,? super V,? extends U> transformer,
    BiFunction<? super U,? super U,? extends U> reducer)
```

Speaker notes

`reduce` riduce la mappa ad un risultato unico. Vanno fornite due funzioni: una per trasformare le coppie chiave, valore nel tipo del risultato, e l'altra per sommare i risultati parziali. Il tipo del risultato deve quindi essere dotato di un'operazione di somma con le consuete proprietà commutativa e associativa.

pcd2018.safe.ReducePerf

```
Random rnd = new Random();  
ConcurrentHashMap<String, Long> map =  
    new ConcurrentHashMap<String, Long>();  
IntStream.range(0, 10000)  
    .forEach(i -> map.put("k" + i, new Long(rnd.nextInt(1000))))
```

Speaker notes

Per provare il funzionamento di queste istruzioni, costruiamo una mappa di long casuali.

```
long start = System.currentTimeMillis();  
Long parres = map.reduceEntries(500,  
    entry -> entry.getValue(), (a, b) -> a + b);  
long partime = System.currentTimeMillis() - start;
```

Speaker notes

Avviamo la sua riduzione, suggerendo tramite l'apposito parametro di usare il parallelismo.


```
long start = System.currentTimeMillis();  
Long parres = map.reduceEntries(10000001,  
    entry -> entry.getValue(), (a, b) -> a + b);  
long partime = System.currentTimeMillis() - start;
```

Speaker notes

Per fare un confronto, verifichiamo il tempo usato se suggeriamo di non usare il parallelismo.

Quiz: costruire un esempio in cui è l'implementazione parallela la più efficace.

Suggerimento: il lavoro di riduzione deve essere superiore all'overhead introdotto dal parallelismo...

```
/**
 * Returns a non-null result from applying the given
 * search function on each (key, value), or null if none.
 * Upon success, further element processing is suppressed.
 *
 * @param the elements needed to switch to parallel
 * @param a search function, that returns non-null on
 *       success
 */
public <U> U search(long parallelismThreshold,
    BiFunction<? super K,? super V,? extends U> searchFunction)
```

Speaker notes

la funzione `search` permette di applicare una ricerca parallela nella mappa. Il risultato è il primo non nullo ritornato dalla funzione di chiave e valore. La ricerca è parallela, ma trovato il risultato tutti i thread di ricerca vengono fermati.

```
/**
 * Performs the given action for each (key, value).
 *
 * @param the elements needed to switch to parallel
 * @param the action (can have side-effects)
 */
public void forEach(long parallelismThreshold,
    BiConsumer<? super K,? super V> action)
```

Speaker notes

infine la funzione `forEach` permette di eseguire un effetto collaterale per ciascuna coppia chiave, valore.

Le funzioni usate nei metodi di trasformazione delle mappe devono:

- non dipendere dall'ordinamento
- non dipendere da uno stato condiviso durante il calcolo

Inoltre, per i metodi diversi da `forEach` *non devono avere effetti collaterali*

Speaker notes

Ovviamente, in `forEach` l'effetto collaterale è lo scopo della funzione. Le prime due condizioni sono invece indispensabili per garantire, in termini generali, la corretta esecuzione dell'operazione parallela.

Per ogni algoritmo vi sono varie versioni:

- con una trasformazione opzionale prima dell'uso del valore
- iterazione solo sulle chiavi o solo sui valori
- con risultati primitivi (`int`, `double` ecc.)

Le operazioni di riduzione, ricerca ed esecuzione di effetti non sono atomiche nel loro complesso, ma ogni coppia chiave-valore non nulla ha una garanzia di `happens-before` con il suo uso nell'iterazione.

BLOCKINGQUEUE

L'interfaccia `BlockingQueue` aggiunge alla classica `Queue` metodi con cui è possibile scegliere la semantica dell'operazione di accodamento e prelievo.

Diventa così possibile richiedere il comportamento desiderato all'interno di una esecuzione concorrente.

Accodamento

metodo	risultato negativo
<code>add(e)</code>	eccezione
<code>offer(e)</code>	<code>false</code>
<code>put(e)</code>	attesa
<code>offer(e, time, unit)</code>	attesa limitata

Speaker notes

si intende "risultato negativo" quando un elemento non è disponibile per l'accodamento o il prelievo

Prelievo

metodo	risultato negativo
<code>remove()</code>	eccezione
<code>poll()</code>	<code>null</code>
<code>take()</code>	attesa
<code>poll(time, unit)</code>	attesa limitata

Speaker notes

il primo metodo ritorna immediatamente con un'eccezione se non ci sono elementi da rimuovere. Il secondo ritorna immediatamente con un valore speciale. Il terzo ed il quarto attendono, per un periodo indefinito o specificato.

Lettura

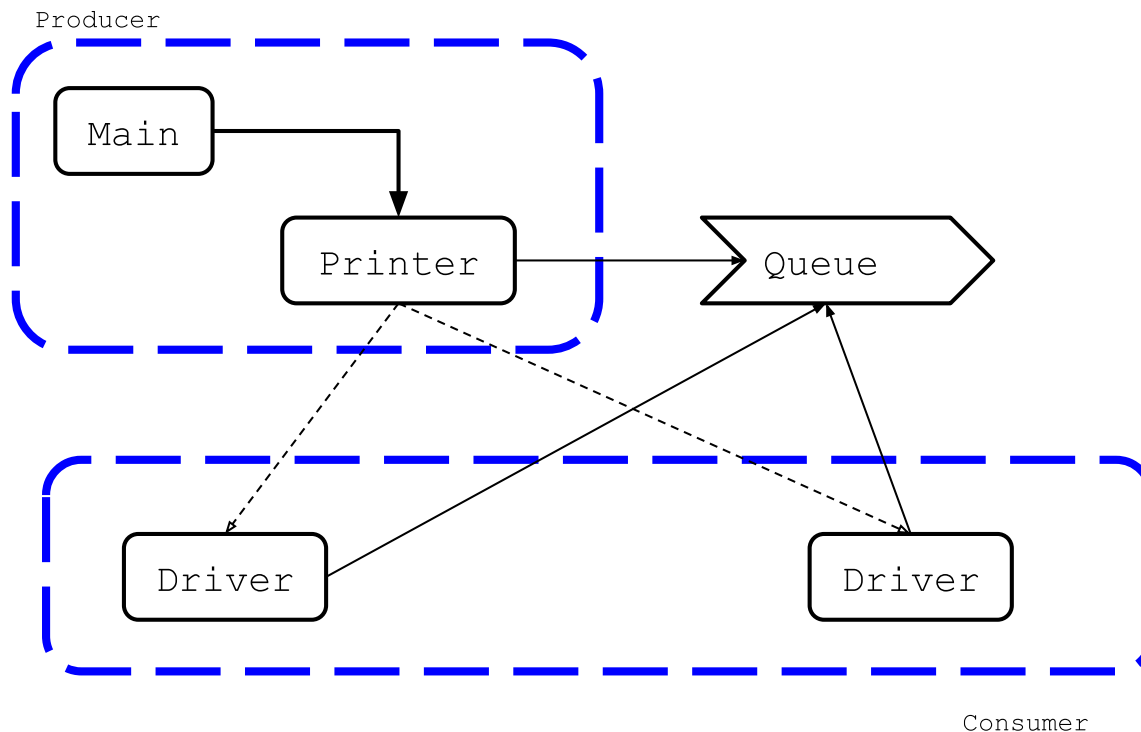
metodo	risultato negativo
<code>element ()</code>	eccezione
<code>peek ()</code>	<code>null</code>

```
/**
 * Removes all available elements from this
 * queue and adds them to the given collection.
 * This operation may be more efficient than
 * repeatedly polling this queue.
 *
 * @param c the collection to transfer elements into
 * @return the number of elements transferred
 */
int drainTo(Collection<? super E> c)
```

Speaker notes

attenzione che il risultato non è definito (non interviene un'eccezione) se la coda viene modificata durante l'operazione: gli elementi accodati potrebbero essere riportati nella collezione, oppure ignorati. Questa operazione è più efficiente che un ciclo di `poll` fino ad esaurimento, ma non è atomica.

Le varie implementazioni di `BlockingQueue`, ciascuna con le sue specifiche caratteristiche, sono la scelta naturale per implementare sistemi Produttore-Consumatore.



Speaker notes

Implementiamo per esempio un sistema costruito in questo modo: la Main class carica comandi nella classe Printer; la classe Printer mantiene una coda dei lavori da effettuare, ed istanzia un numero di Driver che pescano dalla coda ed eseguono i lavori.

pcd2018.safe.PrinterOperator

```
Printer concurrent = new ConcurrentPrinter(8);
Thread thread[] = new Thread[10];
System.out.println("Preparing...");
IntStream.range(0, 10).forEach((i) -> thread[i] =
    new Thread(() -> {
        System.out.println("Queueing job " + i);
        concurrent.printJob(new Object());
        System.out.println("Job " + i + " queued");
    }));
System.out.println("Starting.");
for (int i = 0; i < 10; i++) { thread[i].start(); }
```

Speaker notes

la classe main crea la stampante, e una decina di thread che accodano un job sulla stampante stessa.

pcd2018.safe.ConcurrentPrinter

```
ConcurrentPrinter(int printers) {  
    // limit printers to effective cores  
    size = printers < cores ? printers : cores;  
    // size and build the queue  
    queue = new LinkedBlockingQueue<PrintJob>(QUEUE_SIZE);  
    // start the executor  
    executor = Executors.newFixedThreadPool(size);  
    // start drivers  
    IntStream.range(0, size).forEach((a) ->  
        executor.execute(new PrinterDriver(queue)));  
}
```

Speaker notes

La stampante inizializza nel costruttore la coda, l'Executor e avvia i driver.

pcd2018.safe.ConcurrentPrinter

```
@Override
public void printJob(Object document) {
    try {
        queue.put(new PrintJob(document));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Speaker notes

un `PrintJob` è una semplice value class che tiene il tempo di quando è stata creata per poter misurare il tempo di attesa in coda.

pcd2018.safe.PrinterDriver

```
public void run() {  
    try {  
        while (true) {  
            PrintJob job = queue.take();  
            System.out.printf(...);  
            int duration = rnd.nextInt(2500);  
            Thread.sleep(duration);  
            System.out.printf(...);  
        }  
    } catch (InterruptedException ex) {  
        System.out.println("Printer shutting down.");  
    }  
}
```

Speaker notes

il driver si mette in attesa di un elemento dalla coda, e simula la sua esecuzione in stampa. I messaggi danno conto del tempo d'attesa del job in coda.

Quiz: usare `TimeUnit.X.wait` al posto di `Thread.sleep()` nella classe precedente genera un'eccezione: quale e perché?

La soluzione richiede il materiale della prossima lezione.

Esercizio 1: implementare un `SerialPrinter` che crea un solo driver ed usa un solo thread.

Esercizio 2: Casualmente, il numero di job che vengono accodati è pari a

`ConcurrentPrinter.QUEUE_SIZE`.

E' facile immaginare cosa succede se sono di meno.
Cosa avviene se sono di più?

Esercizio 3: La classe `PrinterOperator` accoda i job tramite threads separati.

Cosa succede, nelle condizioni dell'esercizio 2, se invece la classe accodasse direttamente i job?

Esercizio 4: La classe `PrinterOperator` così come è scritta non termina: la JVM non si chiude perché l'`Executor` del `ConcurrentPrinter` non viene chiuso.

Come si può fare per permettergli di chiudere correttamente, *dopo* aver eseguito tutti i job?

Nota: questo è un esercizio di design; non è detto che sia risolvibile nel design attuale.

Altre varianti:

- `TransferQueue`: interfaccia per una coda in cui i produttori aspettano i consumatori
- `BlockingDeque`: interfaccia che permette di prendere un elemento dalla coda o dalla testa

Speaker notes

Deque è una interfaccia di `j.u.collection`, Transfer no.

- `ArrayBlockingQueue`: implementazione basata su array, con possibilità di *fairness*
- `LinkedBlockingDeque`,
`LinkedBlockingQueue`,
`LinkedTransferQueue`: implementazioni basate su liste collegate

Speaker notes

cercare la dimensione su di una lista collegata non è un'operazione nè $O(1)$ nè tantomeno precisa

- `PriorityBlockingQueue`: coda ordinata per priorità
- `DelayQueue`: un elemento non può essere preso prima di un ritardo impostato
- `SynchronousQueue`: ogni produttore deve attendere un consumatore (capacità nulla)

Altre strutture dati interessanti:

Disruptor

<http://lmax-exchange.github.io/disruptor/>

Altri Esempi:

<http://winterbe.com/posts/2015/05/22/java8-concurrency-tutorial-atomic-concurrent-map-examples/>

THREAD LOCAL VARIABLES

Finora abbiamo visto come condividere la stessa variabile fra più thread.

Un'approccio alternativo è invece garantire che la stessa variabile abbia un valore indipendente e separato per ciascun Thread.

```
/**
 * These variables differ from their normal counterparts
 * in that each thread that accesses one (via its get
 * or set method) has its own, independently initialized
 * copy of the variable.
 */
public class ThreadLocal<T>
```


Una variabile `ThreadLocal` esiste in una copia differente ed indipendente per ciascun Thread che attraversa la sua dichiarazione.

```
/**  
 * Creates a thread local variable. The initial value of  
 * the variable is determined by invoking the get method  
 * on the Supplier.  
 */  
static <S> ThreadLocal<S>  
    withInitial(Supplier<? extends S> supplier)
```

Speaker notes

il `Supplier` permette l'inizializzazione a partire da una strategia esterna.

```
/**
 * Returns the value in the current thread's copy of this
 * thread-local variable. If the variable has no value
 * for the current thread, it is first initialized to the
 * value returned by an invocation of the initialValue()
 * method.
 */
public T get()
```

Speaker notes

questa chiamata, sulla stessa variabile lessicale, avrà un risultato differente per ogni thread.

```
/**  
 * Removes the current thread's value for this thread-local  
 * variable.  
 */  
public void remove()
```

```
/**  
 * Sets the current thread's copy of this thread-local  
 * variable to the specified value.  
 */  
public void set(T value)
```

```
/**  
 * Returns the current thread's "initial value" for  
 * this thread-local variable.  
 *  
 */  
protected T initialValue()
```

pcd2018.safe.LocalVar

```
class LocalVar {  
    private static final var nextId = new AtomicInteger(0);  
  
    ThreadLocal<Integer> counter;  
  
    LocalVar() {  
        counter = ThreadLocal.withInitial(() ->  
            nextId.incrementAndGet());  
    }  
  
    Integer get() { return counter.get(); }  
}
```

Speaker notes

il valore `nextId` è globale; ogni thread accede sempre allo stesso. Il contatore `counter` invece è privato di ciascun thread.

pcd2018.safe.LocalReader

```
class LocalReader implements Runnable {  
    private final LocalVar var;  
    private final int item;  
  
    @Override  
    public void run() {  
        System.out.println(Thread.currentThread().getName() +  
            ", item " + item + ": read " + var.get());  
    }  
}
```

Speaker notes

questo Runnable legge e stampa il valore della variabile threadlocal.

pcd2018.safe.LocalMain

```
ExecutorService executor = Executors.newFixedThreadPool(4);  
LocalVar var = new LocalVar();  
IntStream.range(0, 20).forEach((a) ->  
    executor.execute(new LocalReader(var, a)));  
executor.shutdown();
```

Speaker notes

questo main lancia diversi `Runnable` che condividono la medesima istanza di `LocalVar`. Eppure, ciascuno di loro vi legge un valore diverso da un oggetto che, sintatticamente, dovrebbe essere lo stesso per tutti. E' invece, tramite la classe `ThreadLocal`, garantito come separato per ciascun thread.

Le variabili `ThreadLocal` hanno la cattiva abitudine di assomigliare molto a delle variabili globali.

Usare con cautela.

PUBBLICITÀ

Amazon Corretto

<https://www.infoq.com/news/2018/11/amazon-corretto-java>