

Reti e Laboratorio 3

Modulo Laboratorio 3

a.a. 2024-2025

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 12

Iterator on Concurrent Collections

Atomic Variables, Volatile

6/12/2024

COLLEZIONI ED ITERATORI

- le Collection JAVA supportano diversi tipi di iteratori
- si distinguono riguardo al comportamento di una collezione in presenza di “concurrent modification”
- cosa accade quando la collezione viene modificata, mentre un iteratore la sta scorrendo, e questa modifica arriva “dall'esterno” dell'iteratore?
- **fail-fast**
 - se c'è una modifica strutturale (inserzione, rimozione, aggiornamento), dopo che l'iteratore è stato creato, l'iteratore la rileva e solleva una `ConcurrentModificationException`
 - fallimento immediato dell'operatore, per evitare comportamenti non deterministici
 - la maggior parte delle collezioni “concurrenti” sono fail-fast
Vector, ArrayList, HashMap, ed altre....

FAIL FAST: HASHMAP

```
import java.util.HashMap; import java.util.Iterator;import
java.util.Map; public class FailFastExample {
    public static void main(String[] args)
    { Map<String, String> cityCode = new HashMap<String, String>();
      cityCode.put("Delhi", "India");
      cityCode.put("Moscow", "Russia");
      cityCode.put("New York", "USA");
      Iterator iterator = cityCode.keySet().iterator();
      while (iterator.hasNext()) {
          System.out.println(cityCode.get(iterator.next()));
          cityCode.put("Istanbul", "Turkey"); }}}}
```

India

Exception in thread "main"

java.util.ConcurrentModificationException

at java.util.HashMap\$HashIterator.nextNode(Unknown Source)

at java.util.HashMap\$KeyIterator.next(Unknown Source)

at FailFastExample.main(FailFastExample.java:19)

COLLEZIONI ED ITERATORI

- **fail-safe** (“snapshot”) introdotti in JAVA 1.5 con le concurrent collections
 - creano una copia della collezione, al momento della creazione dell'iteratore e lavorano su questa copia
 - non sollevano `ConcurrentModificationException`
 - l'iteratore accede ad una versione non aggiornata della collezione
 - `CopyOnWriteArrayList`
- **weakly consistent** introdotti in JAVA 1.5 con le concurrent collections
 - l'iteratore e modifiche operano sulla stessa copia
 - no `ConcurrentModificationException`, comportamento fail-safe
 - l'iteratore considera gli elementi che esistevano al momento della costruzione dell'iteratore e può riflettere le modifiche che sono avvenute dopo la costruzione dell'iteratore, anche se non è garantito
 - `ConcurrentHashMap`, ...

WEAK CONSISTENCY: CONCURRENT HASH MAP

da [JavaDocs](#);

“The view's iterator is a "weakly consistent" iterator that will never throw `ConcurrentModificationException`, and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction.”

- l'iteratore
 - non clona la struttura al momento della creazione
 - la collezione può catturare le modifiche effettuate sulla collezione dopo la sua creazione
 - non solleva `ConcurrentModificationException`
- alcuni metodi, `size()` e `isEmpty()`
 - possono restituire un valore “approssimato”
 - “weakly consistent behaviour”

UN ITERATORE WEAKLY CONSISTENT

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Iterator;
public class FailSafeItr {
    public static void main(String[] args)
    {ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<String, Integer>();
    map.put("ONE", 1);
    map.put("TWO", 2);
    map.put("THREE", 3);
    map.put("FOUR", 4);
    Iterator <String> it = map.keySet().iterator();
    while (it.hasNext()) {
        String key = (String)it.next();
        System.out.println(key + " : " + map.get(key));
        // Notice, it has not created separate copy
        // It will print 7
        map.put("SEVEN", 7); }}}
    // the program prints ONE : 1 FOUR : 4 TWO : 2 THREE : 3 SEVEN : 7
```

FAIL SAFE: COPYONWRITEARRAYLIST

- come risulta evidente dal nome, effettua una copia dell'array tutte le volte che viene effettuata un'operazione di modifica (add, set, etc..)
- “**snapshot style iterator**”: usa un riferimento ad una copia dello stato dell'array nel momento in cui l'iteratore è creato
 - l'array riferito non viene mai modificato durante la vita dell'iteratore: l'iteratore non cattura le modifiche effettuate dopo la sua creazione
 - thread-safe: ogni thread lavora su una propria copia
 - fail safe: non solleva `ConcurrentModificationException`
- operazione di copia molto costosa
 - è adatto quando ci sono più accessi in lettura che modifiche

ANCORA SUL MULTITHREADING

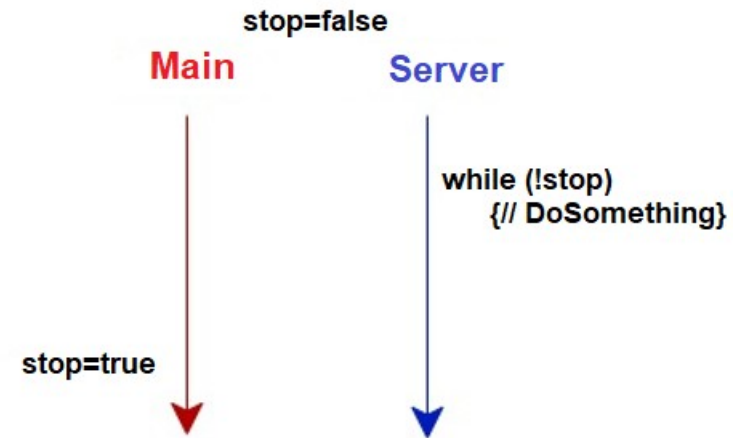
- Variabili volatile
- Variabili Atomic

PERCHE' VOLATILE?

```
public class Server extends Thread
{
    boolean stop = false; int i;

    public void run()
    {
        while(! stop) {};
        System.out.println("Server is stopped....");
    }

    public void stopThread()
    {
        stop = true;
    }
}
```



```
public class StoppingAThread
{
    public static void main(String args[]) throws InterruptedException
    {
        Server myServer = new Server();
        myServer.start();

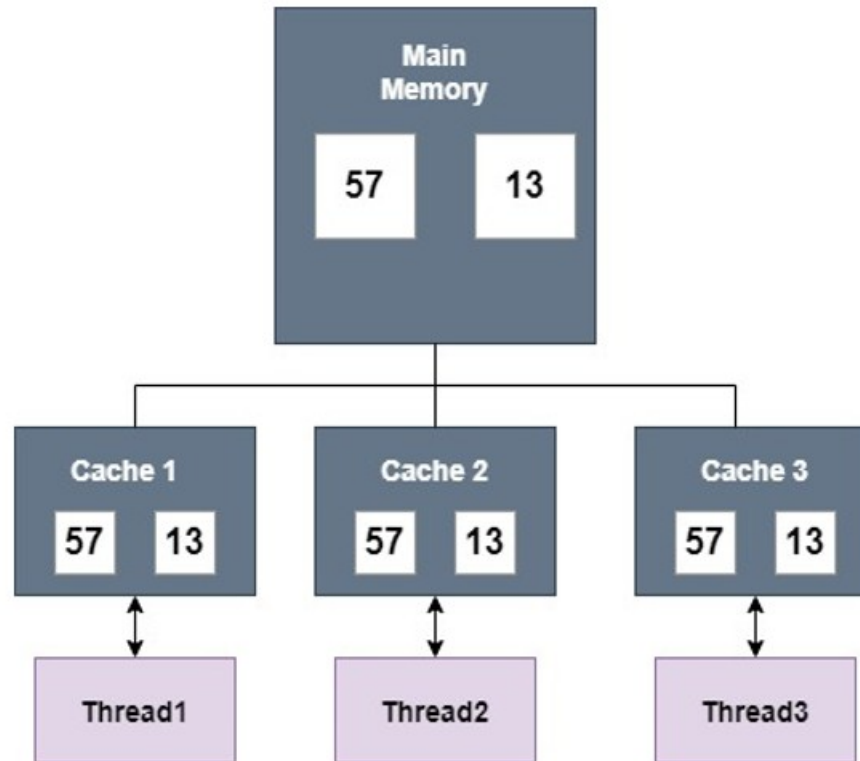
        System.out.println(Thread.currentThread().getName() + " is stopping Server thread");
        Thread.sleep(1000);
        myServer.stopThread();

        System.out.println(Thread.currentThread().getName() + " is finished now");
    }
}
```



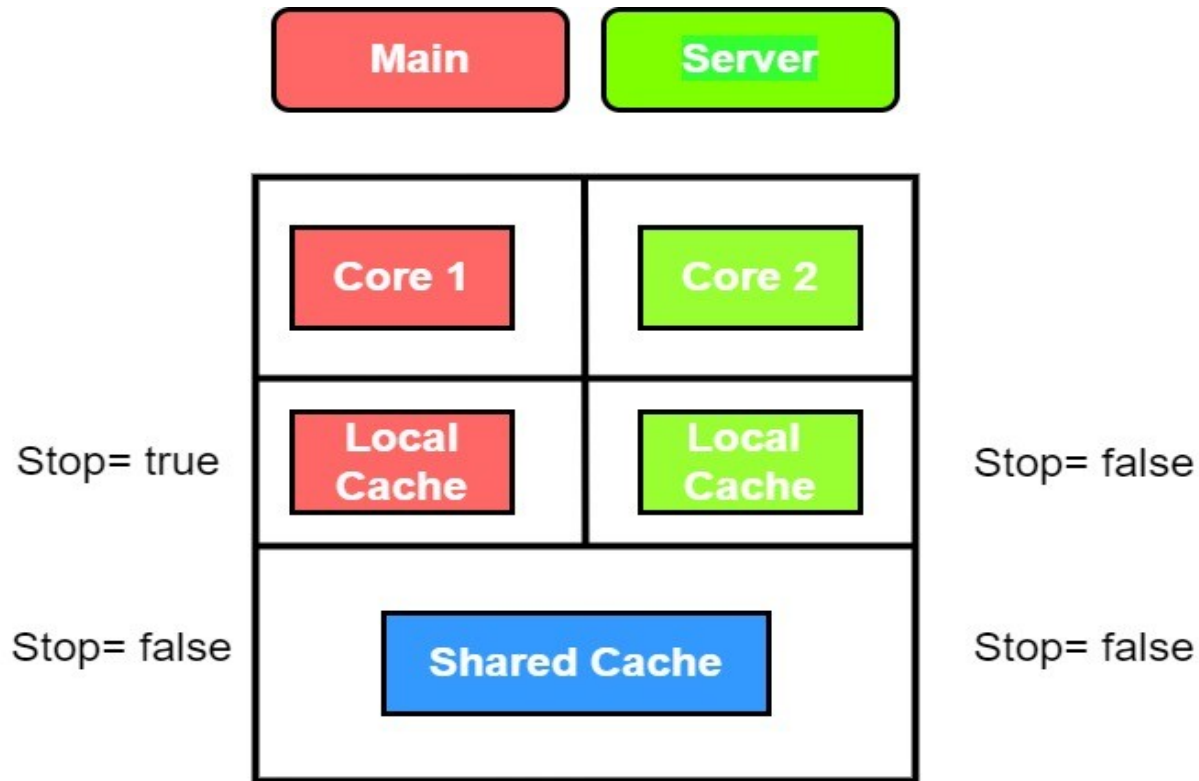
il programma non termina!

IL PROBLEMA DELLA VISIBILITA'



architettura di riferimento, utile per capire il problema della visibilità

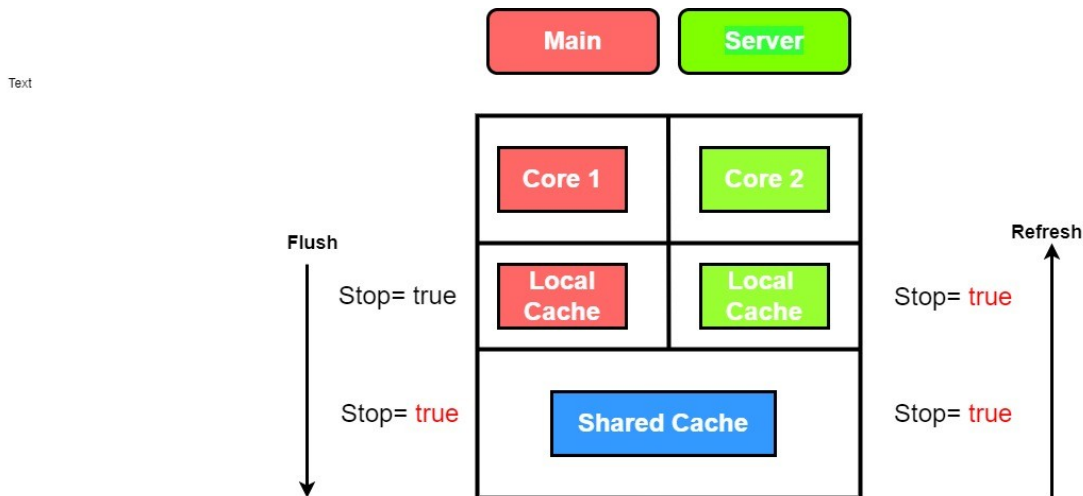
IL PROBLEMA DELLA VISIBILITA'



- quando il **Main** aggiorna **Stop**, è possibile che la modifica non sia riportata nella memoria condivisa
- il **Main** aggiorna la variabile **Stop** nella propria cache, ma la modifica non viene riportata nella memoria condivisa

IL MODIFICATORE VOLATILE

- il problema riguarda la “visibilità” della modifica, non la sincronizzazione: read e write di un booleano sono atomiche
- modifichiamo la dichiarazione con la keyword **volatile**
`volatile boolean stop = false`
- l'aggiornamento ad una variabile **volatile** è sempre effettuato nella main memory
 - flush della cache
- il valore della variabile **volatile** è sempre letto dalla memoria



VOLATILE: VISIBILITA' DI SCRITTURE

- tutte le scritture su una variabile volatile sono riportate direttamente nella memoria condivisa
- inoltre, tutte la variabili visibili dal thread che sta eseguendo la modifica vengono anche sincronizzate sulla memoria condivisa
- esempio:

```
this.nonVolatileVarA = 34;
```

```
this.nonVolatileVarB = new String("Text");
```

```
this.volatileVarC = 300;
```

- quando viene eseguita la terza istruzione, sulla variabile volatileC, i valori delle due variabili non-volatile vengono sincronizzati in memoria condivisa

VOLATILE: VISIBILITA' DI LETTURE

- quando viene letto il valore di una variabile volatile, viene garantito che tale valore venga letto direttamente dalla memoria condivisa
- inoltre, viene fatto il refresh di tutte la variabili visibili dal thread che sta eseguendo la lettura

- esempio:

```
c = other.volatileVarC;
```

```
b = other.nonVolatileB;
```

```
a = other.nonVolatileA;
```

- la prima istruzione è la lettura di una variabile volatile. Quando questa variabile viene letta dalla memoria, viene effettuato il refresh anche delle due altre variabili

UNA SOLUZIONE ALTERNATIVA

```
public class Server extends Thread
```

```
{ Boolean stop = false; int i;
```

```
public void run()
```

```
{ synchronized(stop) {};
```

```
while(! stop)
```

```
{synchronized(stop) {}};
```

```
System.out.println("Server is stopped....");}
```

```
public synchronized void stopThread(){ stop = true; }}
```

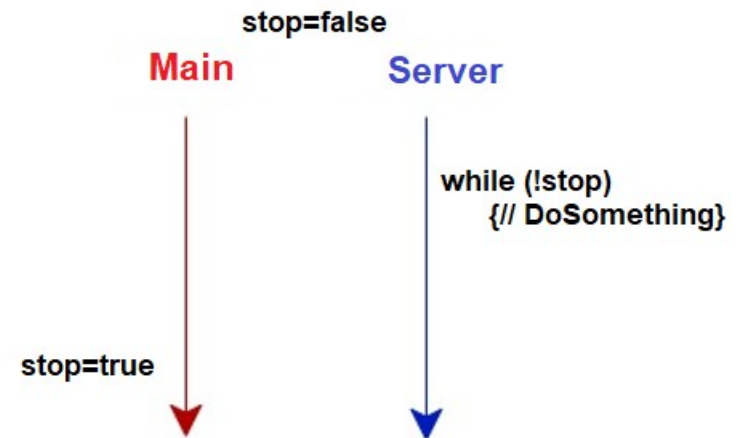
```
public class StoppingAThread
```

```
{...}
```

sincronizzarsi sulla variabile stop ha lo stesso

effetto di usare il modificatore volatile

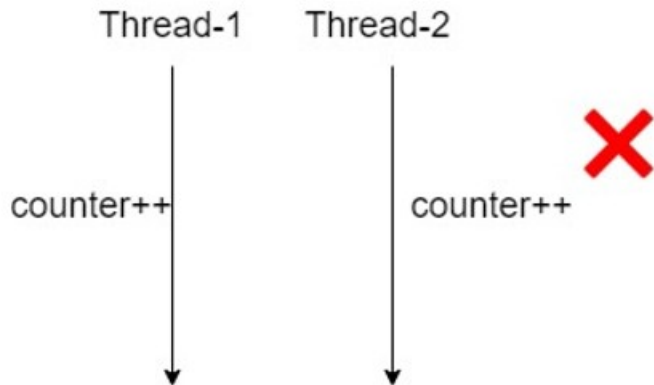
la variabile deve essere definita Boolean, per poter acquisire la lock



SINCRONIZZAZIONE: VISIBILITA'

- blocchi e metodi sincronizzati forniscono garanzia di visibilità simile a quella offerta dal modificatore `volatile`
- quando un thread entra in un metodo o blocco sincronizzato, viene effettuato un refresh di tutte le variabili visibili dal thread
- quando un thread esce da un blocco sincronizzato, tutte le variabili visibili dal thread vengono scritte in memoria
- monitor garantisce sia sincronizzazione che visibilità
- quando usare `volatile`?
 - quando la variabile condivisa è di tipo semplice
 - per acquisire la lock occorrerebbe fare il cast al corrispondente oggetto
 - tipico del pattern “termina l'esecuzione di un thread”

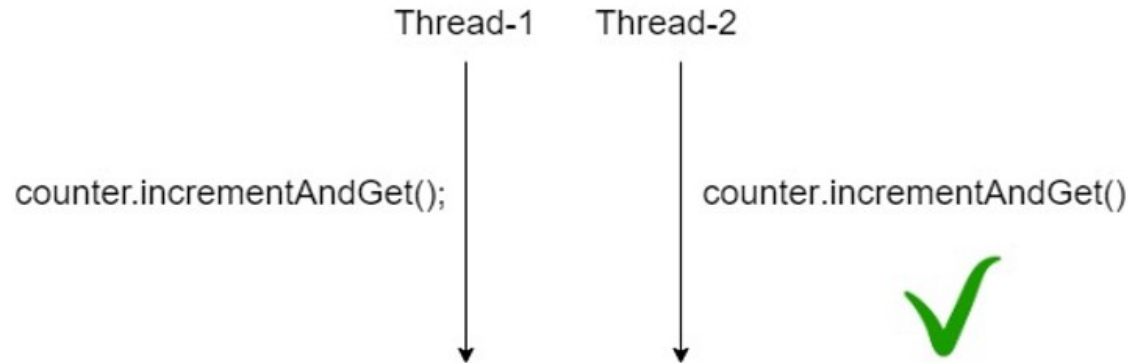
SINCRONIZZAZIONE SU VARIABILI



Thread-1	Thread-2
Read value (=1)	
	Read value (=1)
Add 1 and write (=2)	
	Add 1 and write (=2)

- l'incremento di una variabile (volatile o meno) **non è atomico**
- se più thread provano ad incrementare una variabile in modo concorrente, un aggiornamento **può andare perduto** (anche se la variabile è volatile)
- ovviamente il problema può essere risolto con le lock
- soluzione alternativa: usare le variabili `Atomic`

ATOMIC VARIABLES



```
AtomicInteger value = new AtomicInteger(1);
```

- operazioni atomiche che non richiedono sincronizzazioni esplicite o lock: è la JVM che garantisce la atomicità
 - incrementAndGet(): atomically increments by one
 - decrementAndGet(): atomically decrements by one
 - compareAndSet(int expectedValue, int newValue)
- molte altre classi
 - AtomicLong
 - AtomicBoolean

ATOMIC VARIABLES: UN ESEMPIO

```
import java.util.concurrent.*; import java.util.concurrent.atomic.*;
public class AtomicIntExample {
public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(2);
    AtomicInteger atomicInt = new AtomicInteger();
    for(int i = 0; i < 10; i++){
        CounterRunnable runnableTask = new CounterRunnable(atomicInt);
        executor.submit(runnableTask);
    }
    executor.shutdown(); }
class CounterRunnable implements Runnable {
    AtomicInteger atomicInt;
    CounterRunnable(AtomicInteger atomicInt){this.atomicInt = atomicInt;}
    @Override
    public void run() {
        System.out.println("Counter- " + atomicInt.incrementAndGet());}}
```

JAVA.UTIL.CONCURRENT.ATOMIC



WHAT COMES NEXT?



- corso di web scraping
- complementare Laurea Triennale in Informatica
- secondo semestre

LABORATORIO DI WEB SCRAPING

- *Python*
 - *fondamenti*
- *Reperimento di Dati*
 - *API: user authentication, costruzione di queries, esempi: Google Big Queries*
 - *Web Scraping: pagine statiche : BeautifulSoup. Pagine dinamiche : Selenium*
- *Analisi dei Dati*
 - *Dati Tabellari,*
 - *la libreria Pandas: Selecting, Group_by, Merging e Concatenating, Pivoting*
 - *Visualizing: Matplotlib*
 - *Data Clustering, Time Series*
 - *Dati Relazionali*
 - *la libreria NetworkX: graph models: random graphs, scale free networks, small worlds.*
 - *proprietà caratteristiche: diametro, centrality, clustering coefficient, communities*
- *Casi d'uso: cryptocurrencies*
 - *struttura delle transazioni (Bitcoin, Ethereum)*
 - *blockchain exploreres*
 - *NFT Markets*
 - *grafi di transazioni*

WHAT COMES NEXT?



- corso di Peer to Peer & Blockchain
- Laurea Magistrale in Informatica
- secondo semestre

PEER TO PEER E BLOCKCHAINS

- Sistemi p2p: concetti generali, unstructured overlays: Flooding, Random Walks, Epidemic Diffusion, structured overlays: Distributed Hash Tables (DHT), Routing on a DHT, IPFS
- Blockchains: concetti di base, strumenti crittografici di base (digital signatures, cryptographic hash, Zero Knowledge,...)
- structure dati: Merkle trees, Merkle Patricia tries
- Bitcoin
 - struttura delle transazioni e dei blocchi. Meccanismo di Mining : Proof of Works, ricompense
 - attacchi, 51%, double spending
 - tracciabilità e mixing. Struttura della rete P2P
- Ethereum:
 - smart contract, gas. Meccanismo di consenso: dalla PoW alla PoS
 - Solidity, programmazione di smart contracts. Smart contract security e attacks
- Applicazioni della tecnologia dei Distributed Ledgers: cryptocurrencies, tokens, Self Sovereign Identity, Supply-chains
- L'ecosistema delle blockchain
 - exchangers, mixers, miners

WHAT COMES NEXT? PROPOSTE DI TESI

- applicazione di tecniche crittografiche in collaborazione con la Prof. Bernasconi
 - zero-knowledge
 - authenticated data structures
- analisi di blockchain transactions
 - analisi di transazioni da blockchain
 - analisi di cryptoexchangers
- applicazioni della blockchain
 - identità digitale: Self Sovereign Identity (SSI)
 - collegamento blockchain/mondi virtuali: Decentraland
- tecnologie legate a blockchain
 - cross-chain system

Buon lavoro!

