Reti e Laboratorio
Modulo Laboratorio III
a.a. 2026-2027

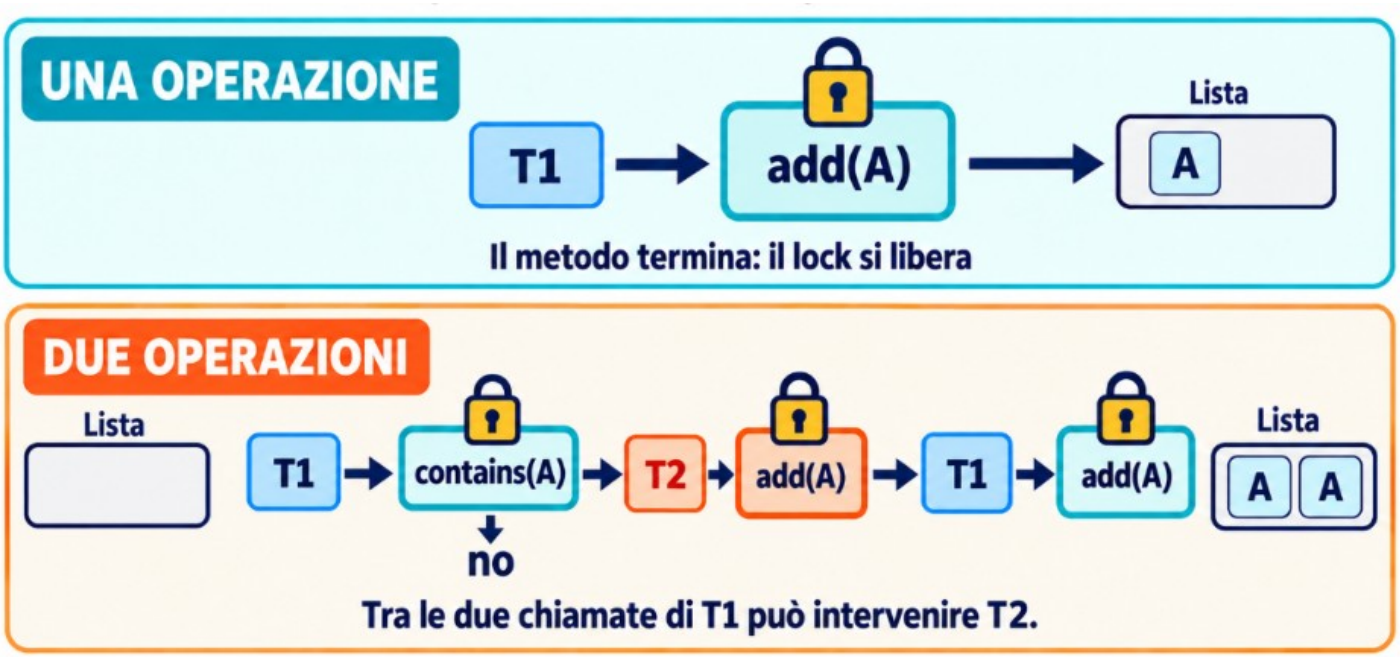
docente: Laura Ricci

laura.ricci@unipi.it

Lezione 5
Concurrent Collections: iteratori, Atomic Variables, Volatile
12/10/2026

COLLEZIONI SINCRONIZZATE

- `Vector` o le collezioni sincronizzate con `Collections.synchronized` sono struttura dati thread safe, ma che significa, esattamente?
- nelle collection sincronizzate, ogni chiamata ai metodi della collection
  - è protetta singolarmente
  - una sequenza di chiamate, compresa l'intera iterazione, non diventa automaticamente atomica



OPERAZIONI COMPOSTE NON SAFE: LIST

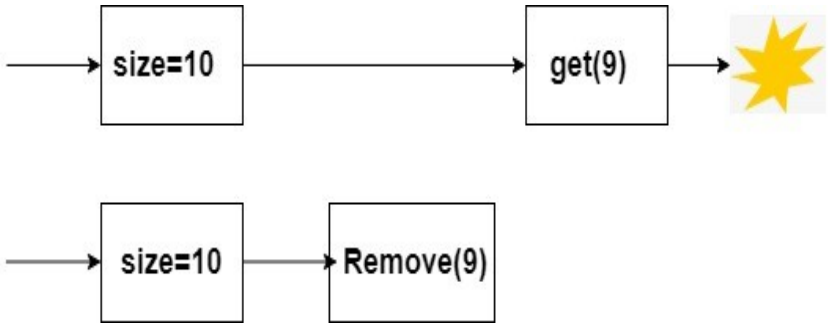
```
if(!synchList.isEmpty())
        synchList.remove(0);
```

- `isEmpty()` e `remove()` sono entrambe operazioni atomiche, ma la loro combinazione non lo è.
- scenario di errore:
  - una lista con un solo elemento.
  - il primo thread verifica che la lista non sia vuota e viene deschedulato prima di rimuovere l'elemento.
  - un secondo thread rimuove l'elemento, il primo thread torna in esecuzione e prova a rimuovere un elemento non esistente
- `Java Synchronized Collections` sono conditionally thread-safe.
  - le operazioni individuali sulle collezioni sono safe, ma funzioni composte da più di una operazione singola possono non risultarlo.

OPERAZIONI COMPOSTE NON SAFE: VECTOR

```
public class UnsafeVector{
public static <T> T getLast (Vector<T> list){
   int     lastIndex = list.size() - 1;
   return (list.get(lastIndex)); }

public static void deleteLast (Vector<T> list){
   int     lastIndex = list.size() - 1;
   list.remove(lastIndex); } }
```



- in caso di accessi concorrenti, questo programma genera `ArrayIndexOutOfBoundsException`
- `Vector` è una struttura dati thread safe, ma la sequenza di operazioni non è atomica

OPERAZIONI COMPOSTE NON SAFE: MAP

```
import java.util.*;
import java.util.concurrent.*;
public class CHashMap {
   private Map<String, Object> theMap =
                new ConcurrentHashMap<>();
   public Object getOrCreate(String key) {
     Object value = theMap.get(key);
     try { Thread.sleep(5000);
         } catch(Exception e) {};
     if (value == null) {
       value = new Object();
       theMap.put(key, value); }
     return value.hashCode();}}
```

```
public class Main {
  public static void main(String [] args)
   {CHashMap ex= new  CHashMap();
  Thread t1 = new Thread (new Runnable()
     {public void run()
        {System.out.println
                 (ex.getOrCreate("5"));};});
     t1.start();
     Thread t2 = new Thread (new Runnable()
       {public void run()
 {System.out.println
                (ex.getOrCreate("5"));};});
     t2.start();
     }}
```

- `GetOrCreate` non è atomica
- `t1` e `t2` stampano due valori diversi, entrambi associati alla stessa chiave,

GARANTIRE INDIVISIBILITA': BLOCCHI SINCRONIZZATI

- usando le lock implicite, si può sincronizzare un segmento di codice di un metodo

```
synchronized(lockObject)
        {// synchronized statements}
```

- un thread
  - acquisisce la lock implicita sull'oggetto indicato (`lockObject`), quando entra nel blocco sincronizzato
  - la rilascia quando termina il blocco sincronizzato
- sincronizzare un intero metodo equivale ad inserire il codice del metodo di un blocco sincronizzato sull'oggetto `this`

```
class Program {
    public void foo() {
        synchronized(this){
                                …} } }
```

GARANTIRE INDIVISIBILITA': BLOCCHI SINCRONIZZATI

- richiesta una sincronizzazione esplicita da parte del programmatore per sincronizzare una sequenza di operazioni

```
synchronized(synchList)
    {
       if(!synchList.isEmpty())
            synchList.remove(0);
                    }
```

- tipico esempio di utilizzo di blocchi sincronizzati
- il thread che esegue l'operazione composta acquisisce la lock sulla struttura `synchList` più di una volta:
  - quando esegue il blocco sincronizzato
  - quando esegue i metodi della collezione

ma...il comportamento corretto è garantito perchè le lock sono rientranti

INDIVISIBILITA' CON BLOCCHI SINCRONIZZATI

```
@ThreadSafe
 public class SafeVector{
    public static <T> T getLast (Vector<T> list) {
          synchronized (list)
            {
              int lastIndex = list.size() - 1;
              return (list.get(lastIndex));
            }}
  public static void deleteLast (Vector<T> list) {
        synchronized (list)
           {
              int  lastIndex = list.size() - 1;
              list.remove(lastIndex); }
          }}
```

WAIT/NOTIFY E BLOCCHI SINCRONIZZATI

- attendere del verificarsi di una condizione su un oggetto diverso da `this`

```
synchronized (obj)
    while (!condition)
          {try {obj.wait ();}
    catch (InterruptedException ex){...}}
```

- segnalare una condizione

```
synchronized(obj){
    condition=...;
    obj.notifyAll()}
```

INDIVISIBILITA': OPERAZIONI PREDEFINITE

le concurrent collections offrono un insieme di operazioni composte atomiche

- sequenze di operazioni di uso comune
- definite come un'operazione unica
- la `JVM` traduce la singola operazione "ad alto livello" in una sequenza di operazioni a più basso livello
- garantisce inoltre la corretta sincronizzazione su tale operazione
  - ...secondo la filosofia "do not re-invent the wheel..."

INDIVISIBILITA': OPERAZIONI PREDEFINITE

- funzioni del tipo "query-then-update", "test-and-set"
- per ConcurrentHashMap è utile la `putIfAbsent`

```
public interface ConcurrentMap<K,V> extends Map<K,V> {
   V putIfAbsent(K key, V value);
      // Insert into map only if no value is mapped from K
      // returns the previous value associated to the key
      // or null if there is no mapping for that key
   boolean remove(K key, V value);
      // Remove only if K is mapped to V
   boolean replace(K key, V oldValue, V newValue);
      // Replace value only if K is mapped to oldValue
   V replace(K key, V newValue);
      // Replace value only if K is mapped to some value    }
```

OPERAZIONI COMPOSTE: PUTIFABSENT

```
import java.util.*;
import java.util.concurrent.*;
public class Main1 {
static Map<String, Object> theMap = new ConcurrentHashMap<>();
public static void main(String [] args)
  { Thread t1 = new Thread (
        new Runnable() {public void run()
                {Object obj1 = new Object();
                 System.out.println(theMap.putIfAbsent("5",obj1));};});
    t1.start();
    Thread t2 = new Thread (new Runnable() {public void run()
                  {Object obj2 = new Object();

System.out.println(theMap.putIfAbsent("5",obj2));};});
    t2.start();}}
```

COLLEZIONI ED ITERATORI A CONFRONTO

- 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?
- comportamenti possibili
  - fail-fast
  - fail safe
  - weakly consistent

ITERATORI FAIL-FAST

- collezioni fail-fast:

  `Vector, Collections.synchronizedList(…), Collections.synchronizedMao(…), …`

- non sono concurrent collections, ma sono sincronizzate da JAVA (`Vector`) o applicando esplicitamente un wrapper `Collections.synchronized…`

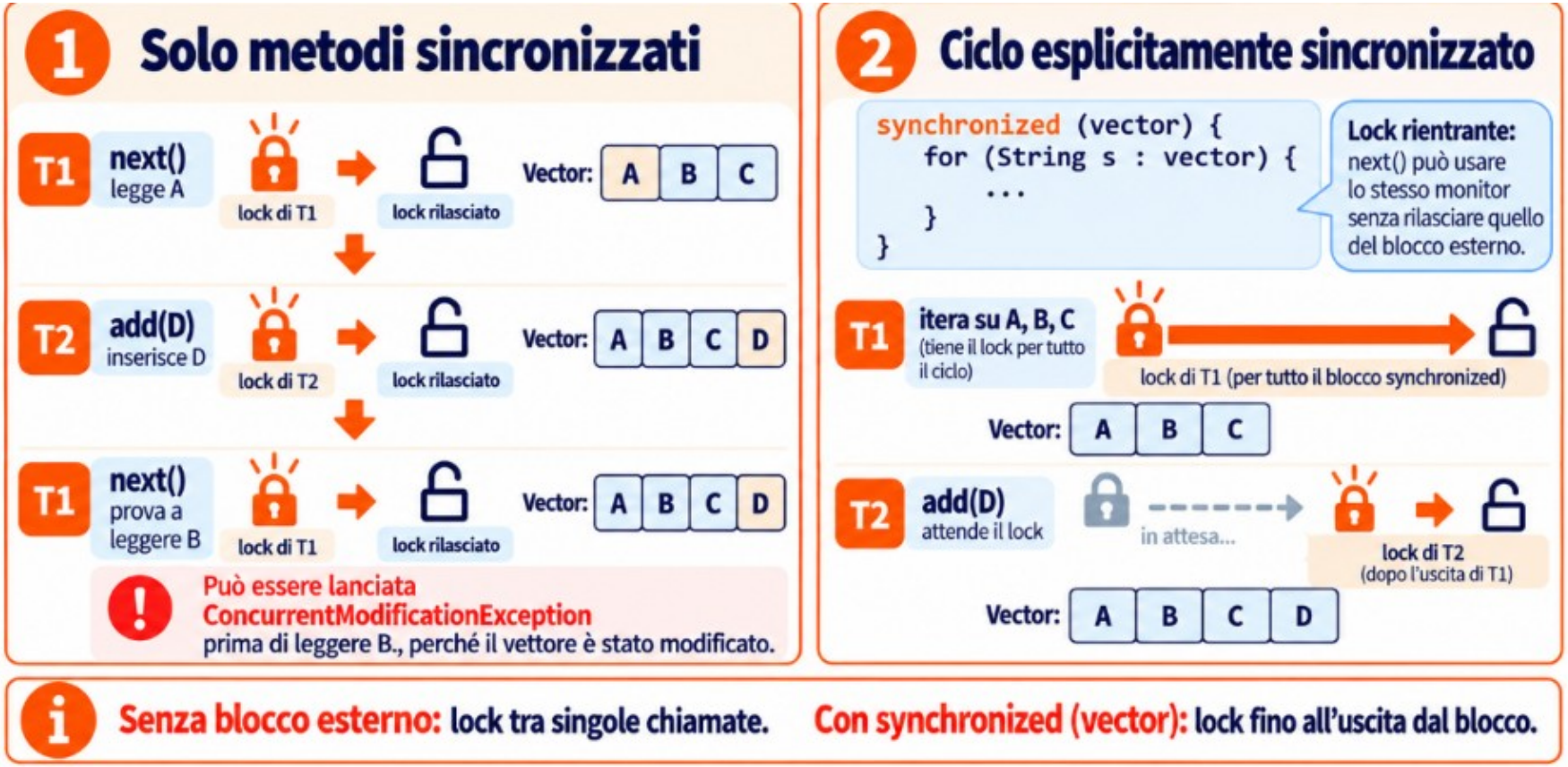
- nelle collezioni sincronizzate, la sincronizzazione protegge ogni singola operazione, ma non l'intera iterazione dell'iteratore: tra una lettura e la successiva, un altro thread può modificare la collezione

- se, dopo che l'iteratore è stato creato, c'è una modifica strutturale (inserzione, rimozione, aggiornamento), l'iteratore la rileva e solleva

  `ConcurrentModificationException`

- fallimento immediato dell'operatore, per evitare comportamenti non deterministici

ITERATORI FAIL-FAST



ITERATORI 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)
```

ITERATORI FAIL SAFE

- 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: `CopyOnWriteArrayList`
  - non sollevano `ConcurrentModificationException`



ITERATORI WEAKLY CONSISTENT

- weakly consistent introdotti in `JAVA 1.5` con le concurrent collections
- ad esempio, iteratori su `ConcurrentHashMap`

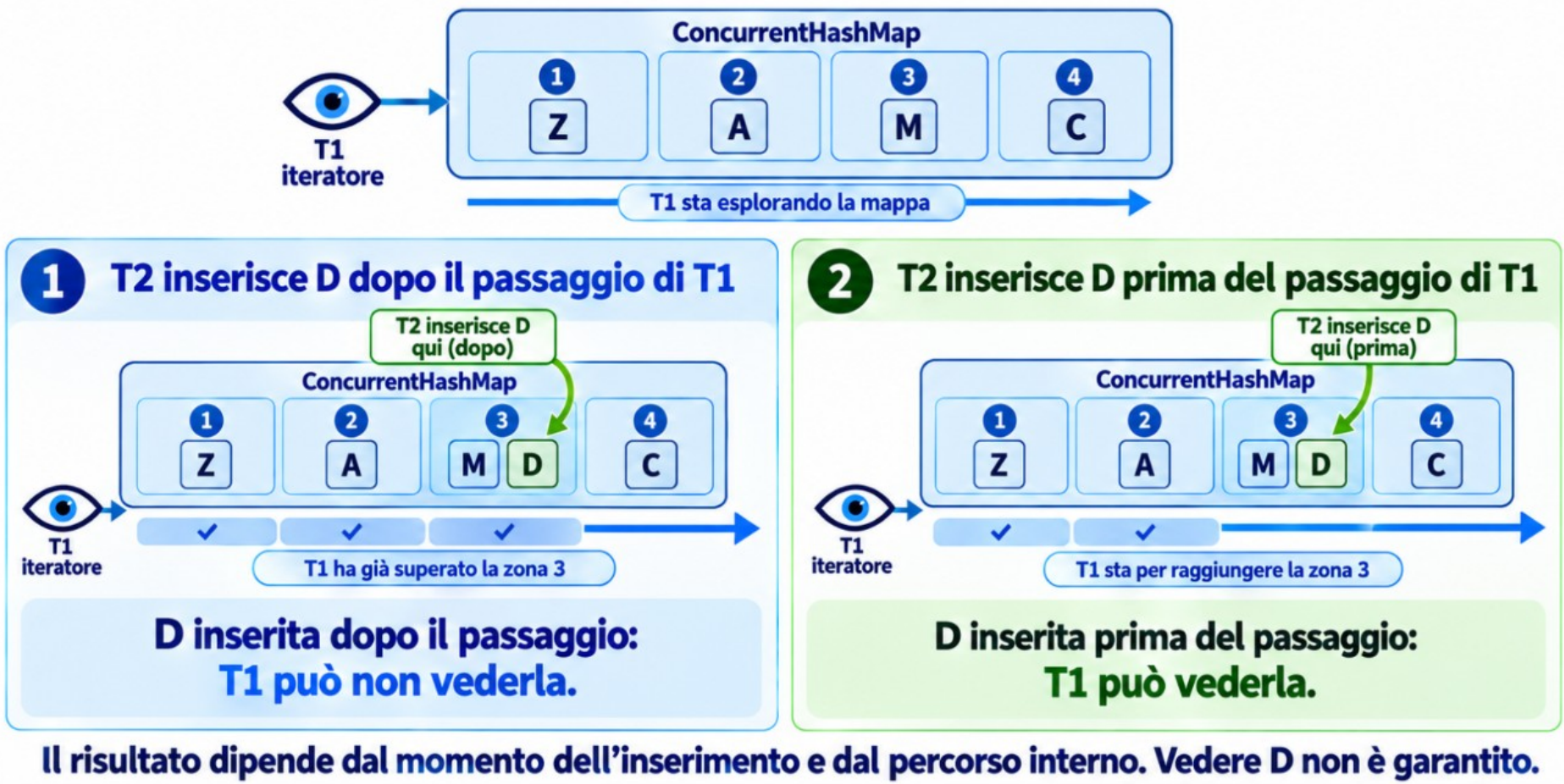
```
ConcurrentHashMap<String, Integer> mappa = new ConcurrentHashMap<>();
mappa.put("A", 10);
mappa.put("B", 20);
mappa.put("C", 30);

Iterator<String> iteratore = mappa.keySet().iterator();

while (iteratore.hasNext()) {
    String chiave = iteratore.next();
    System.out.println(chiave);
}
```

  - l'iteratore e le modifiche operano sulla stessa copia
  - se la `ConcurrentHashMap` viene modificata dopo che l'itetore è stato creato
    - 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

ITERATORI WEAKLY CONSISTENT



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."

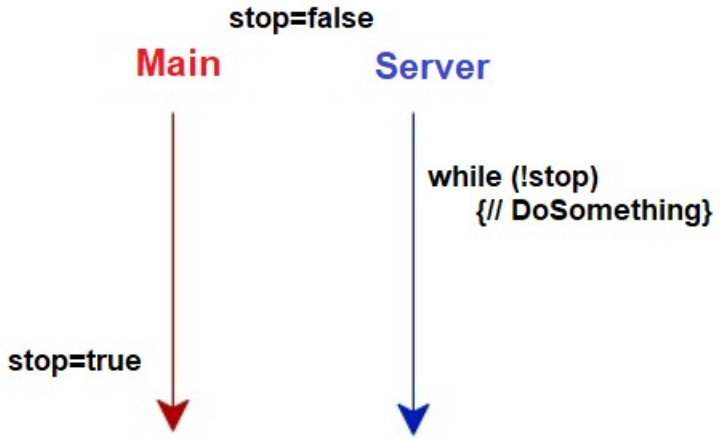
- l'iteratore
  - non clona la struttura al momento della creazione
  - la collezione può catturare le modifiche effettute sulla collezione dopo la sua creazione
    - non solleva `ConcurrentModificationException`
- alcuni metodi, `size()` e `isEmpty()`
  - possono restituire un valore "approssimato"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

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;}}
```

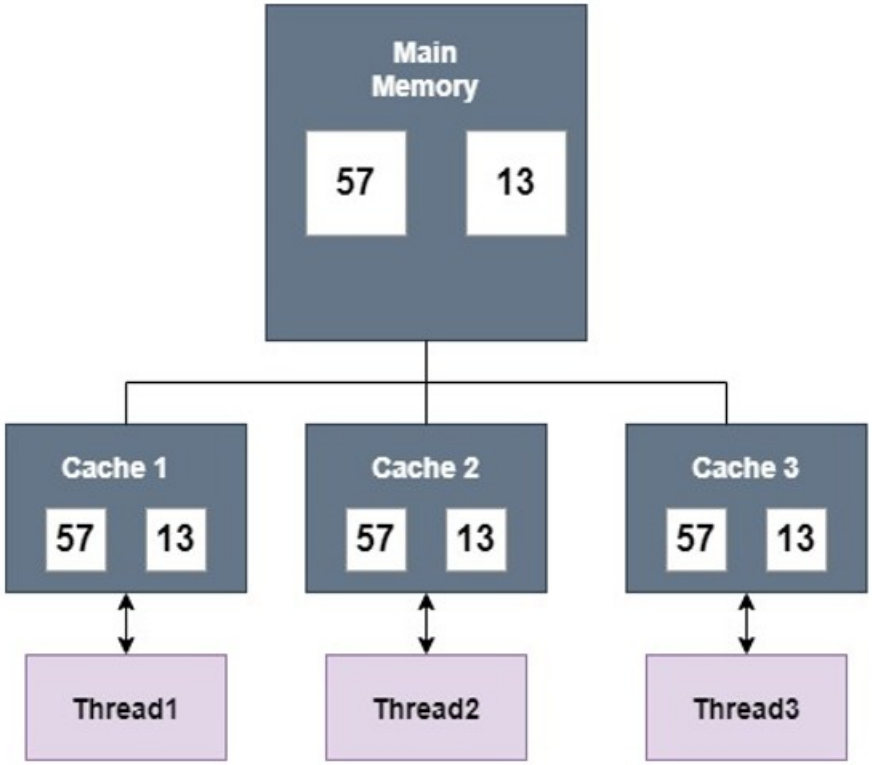



```
public class StoppingAThread
{ public static void main(String args[]) throws InterruptedException
{ Server myServer = new Server();
  myServer.start();
  System.out.println(Thread.currentThread().getName() + " is stopping Server thread");
  Thread.sleep(1000);
  myServer.stopThread();
  System.out.println(Thread.currentThread().getName() + " is finished now"); }}
```



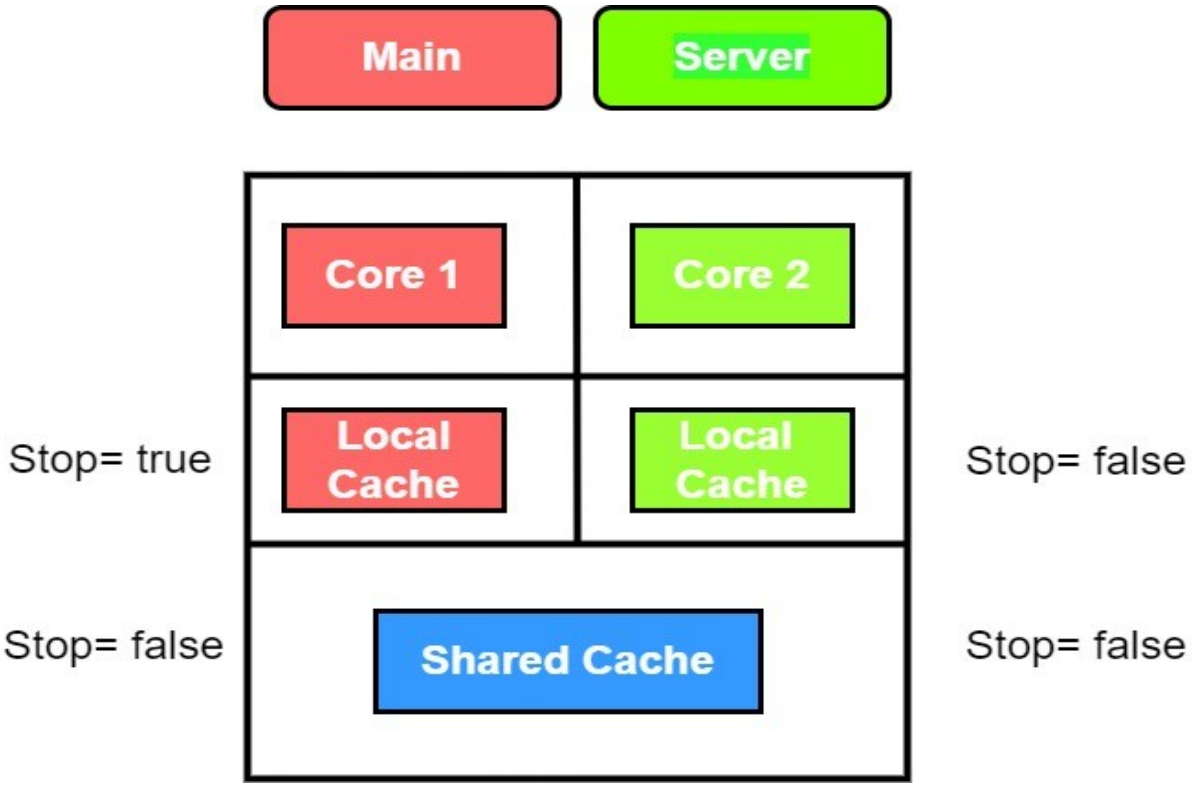
il programma non termina!

IL PROBLEMA DELLA VISIBILITA'



architettura di riferimento, utile per capire il problema della visibilità

IL PROBLEMA DELLA VISIBILITA'



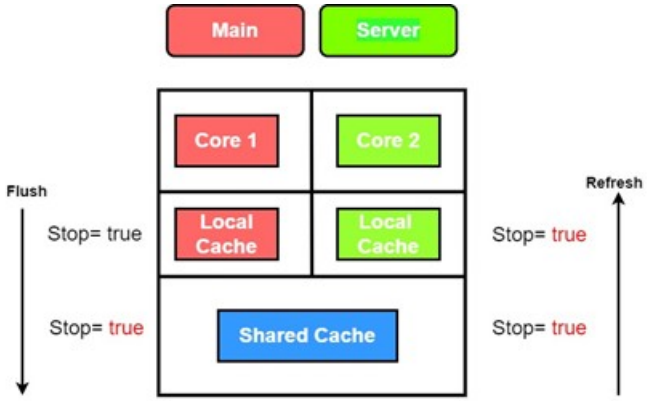
- quando il `Main` aggiorna `Stop`, è possibile che la modifica non sia riportata nella memoria condivisa
- il `Main` aggiorna la variabile `Stop` nella propria cache, ma la modifica non viene riportata nella memoria condivisa

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:LA RELAZIONE "HAPPENS BEFORE"

- oltre alla visibilità, volatile crea una relazione di ordine tra i thread

```
int risultato;
volatile boolean pronto;

// Thread 1
risultato = 42;
pronto = true;

// Thread 2
if (pronto) {
     System.out.println(risultato);  // stampa 42
}
```

- se il `Thread 2` legge `pronto == true`, vede anche la scrittura `risultato = 42`, fatta prima dal `Thread 1`

- la scrittura del flag volatile e la successiva lettura dello stesso flag stabiliscono una relazione happens-before

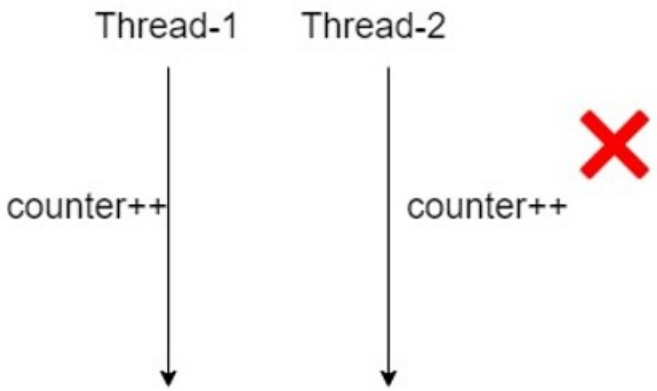
SINCRONIZZAZIONE: VISIBILITA'

- 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"

ATTENZIONE VOLATILE NON E' UNA LOCK!



ATOMIC VARIABLES: PERCHE'?



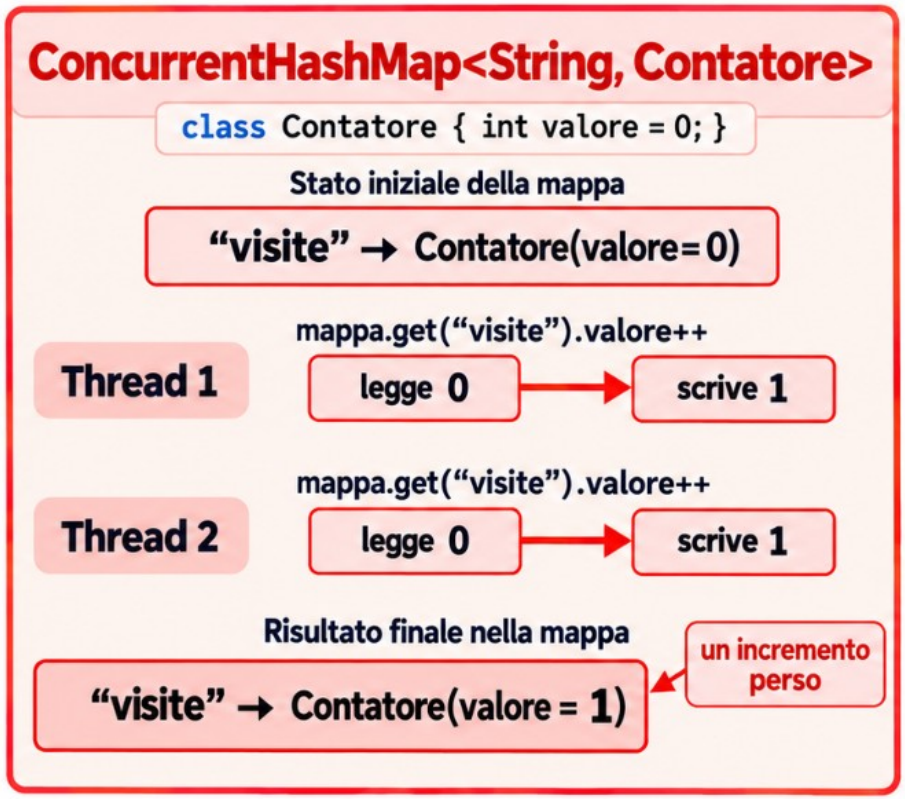
Thread-1	Thread-2
Read value (=1)	
	Read value (=1)
Add 1 and write (=2)	
	Add 1 and write (=2)

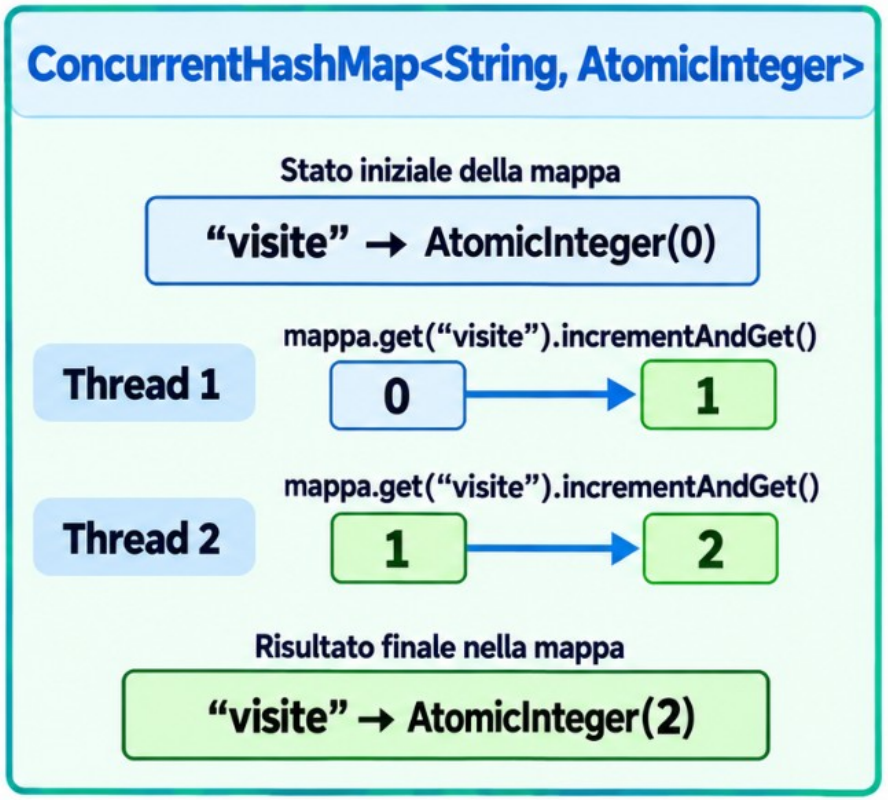
- l'incremento di una variabile (volatile o meno) non è atomico
- se più thread provano ad incrementare una variabile in modo concorrente, un aggiornamento può andare perduto (anche se la variabile è volatile)
- ovviamente il problema può essere risolto con le lock
- soluzione alternativa: usare le variabili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

ATOMIC VARIABLES: PERCHE'?

```
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());}}
```

ATOMIC VARIABLES E MAPS





JAVA.UTIL.CONCURRENT.ATOMIC

