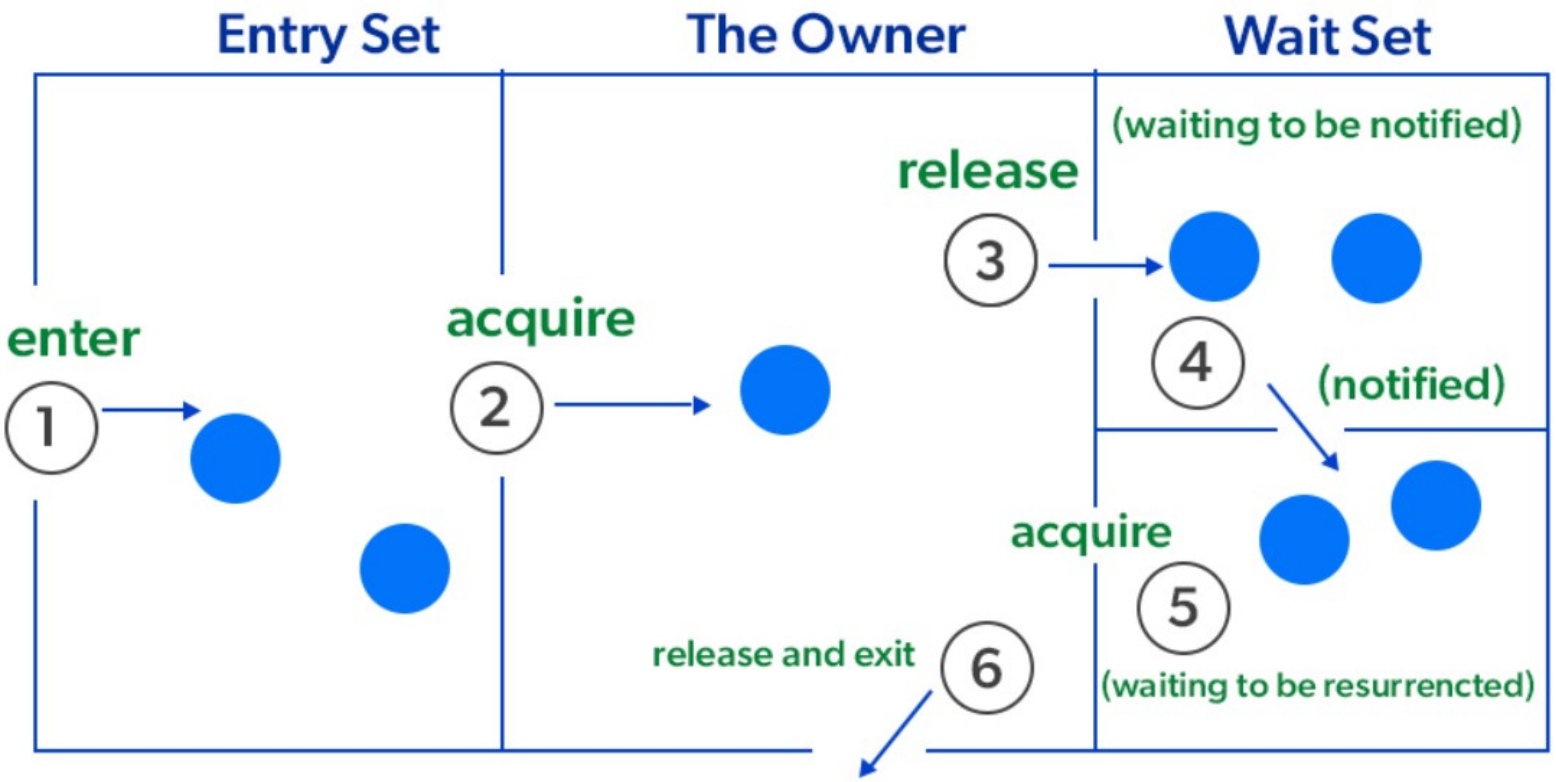
Reti e Laboratorio
Modulo Laboratorio III
AA. 2026-2027

docente: Laura Ricci

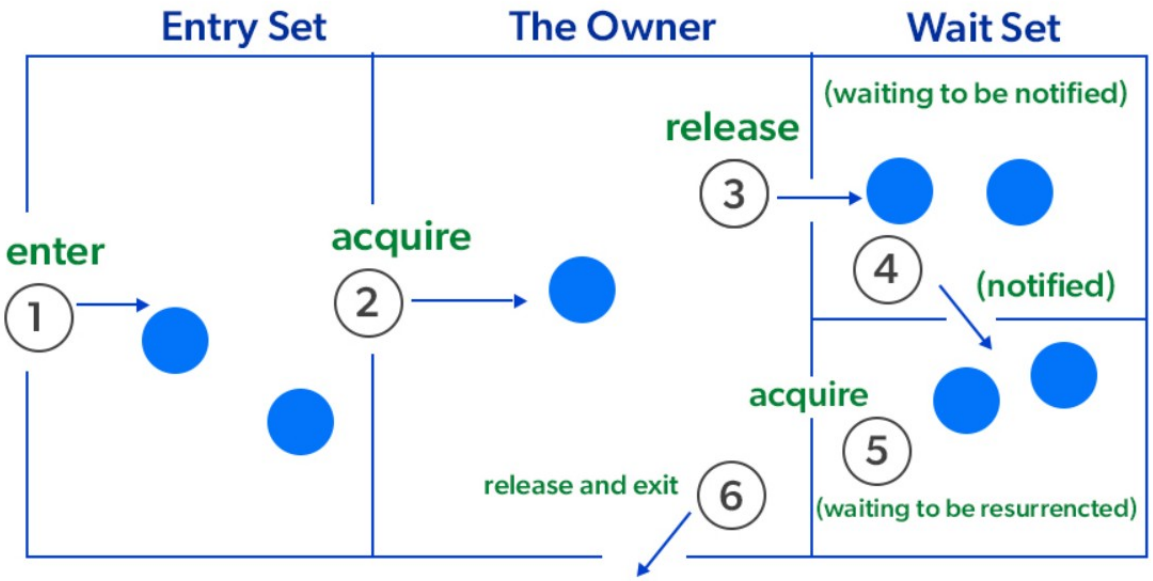
laura.ricci@unipi.it

Lezione 4
Synchronized e Concurrent Collections
08/10/2026

MONITOR: RECAP



REGOLA D'ORO: WAIT IN UN LOOP



- il wait set contiene tutti i thread in attesa di essere notificati di un evento
  - consumatori in attesa che venga inserito un elemento nella coda
  - produttori in attesa che venga liberata una posizione nella coda
- una `notifyall` risveglia tutti i thread nel wait set
- tutti i thread competono per riacquisire la lock
- quando un thread riacquisisce la lock, la condizione su cui era in attesa potrebbe non essere più verificata

IN GENERALE…

```
public synchronized void act()
        throws InterruptedException
{
  while (!cond) wait();
  // modify monitor data
  notifyAll()
}
```

- poichè la coda di attesa è unica per tutte le condizioni, un thread T potrebbe essere stato risvegliato in seguito al verificarsi di una condizione che poi diventa nuovamente falsa
  - la condizione su cui T è in attesa si è verificata
  - però un altro thread la ha resa di nuovo invalida, dopo che T è stato risvegliato
- il ciclo può essere evitato solo se si è sicuri che questo non accada

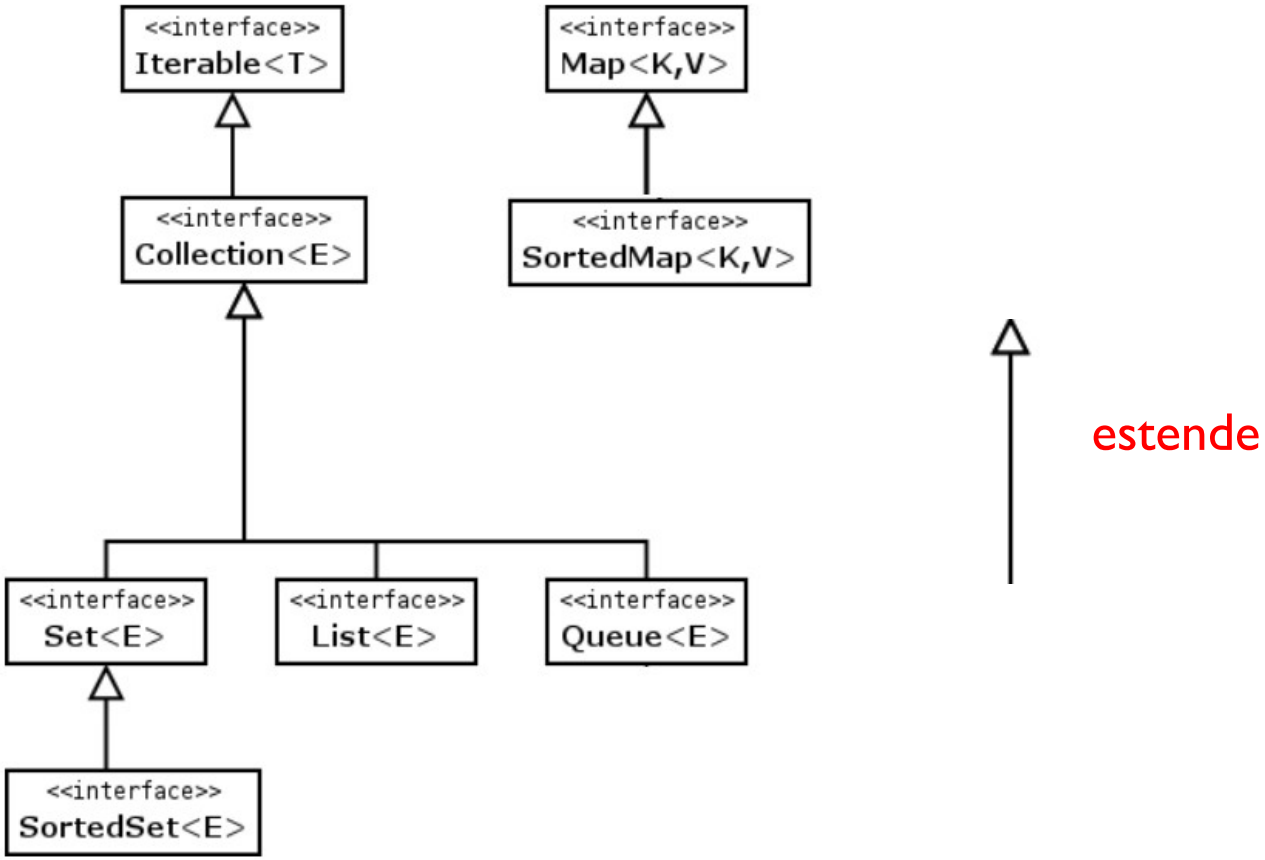
MONITOR E LOCK: CONFRONTI

- monitor: vantaggi
  - l'unità di sincronizzazione è il metodo: tutte le sincronizzazioni sono visibili esaminando firma dei metodi
  - costrutto strutturato. Diminuisce la complessità del programma concorrente: deadlocks, mancato rilascio di lock, maggior manutenibilità del software
- monitor: svantaggi: "coarse grain" synchronization, per-object synchronization, può diminuire il livello di concorrenza
- lock esplicite, vantaggi:
  - maggior numero di funzioni disponibili, maggiore flessibilità
    - `tryLock()` il thread prova ad acquisire una lock, ma non mi blocca
    - `read/write locks`: multiple reader single writer
- lock esplicite, svantaggi: codice poco leggibile, se usate in modo non strutturato

JAVA COLLECTION FRAMEWORK: RECAP

- un insieme di classi che consentono di lavorare con gruppi di oggetti, ovvero collezioni di oggetti
  - classi contenitore
  - introdotte a partire dalla release `1.2`
  - contenute nel package `java.util`
  - rivedere le implementazioni delle collezioni più importanti con lo scopo di utilizzare nel progetto le strutture dati più adeguate
- quali funzionalità aggiuntive sono state introdotte per supportare il multithreading?
  - **`syncronized collections`**
  - **`concurrent collections`**

JAVA COLLECTION INTERFACES IN JAVA.UTIL



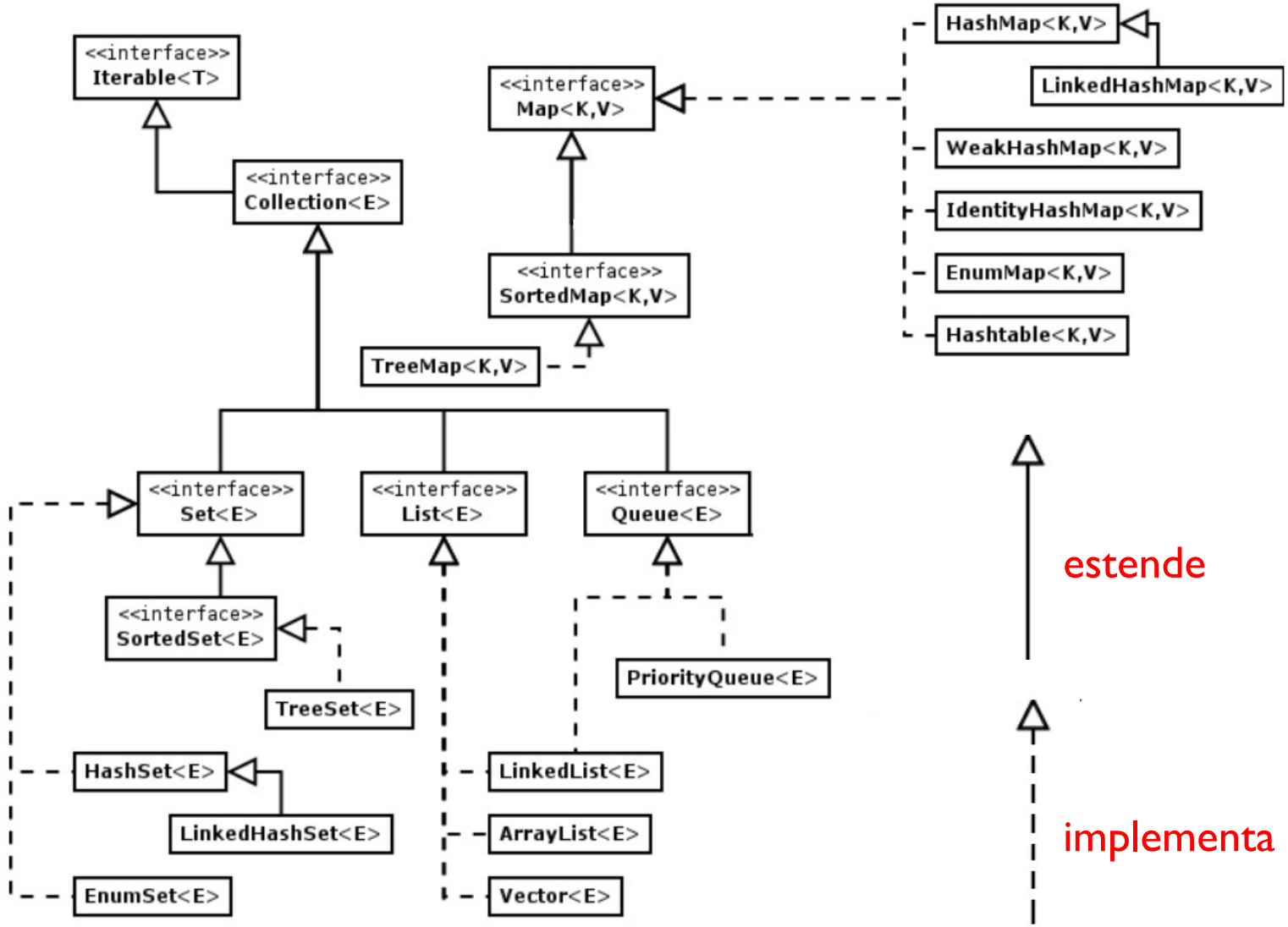
- sono riportate solo le interfacce principali presenti in `java.util`

JAVA COLLECTION: INTERFACES

tre strutture principali per rappresentare una collezione di valori, rappresentate da altrettante interfacce

- list: una collezione ordinata (una sequenza) di valori; possono esistere duplicati
- set: una collezione dove ciascun valore appare una sola volta; non ci sono duplicati, e, in generale, i valori non sono ordinati
  - prevista però una interfaccia in cui i valori possono essere ordinati
- map: una collezione in cui vi è un mapping da chiavi a valori. Le chiavi sono uniche, i valori no
  - le chiavi possono essere ordinate

JAVA COLLECTION: INTERFACES AND CLASSES



DISTRICARSI NELLA GIUNGLA DELLE CLASSI

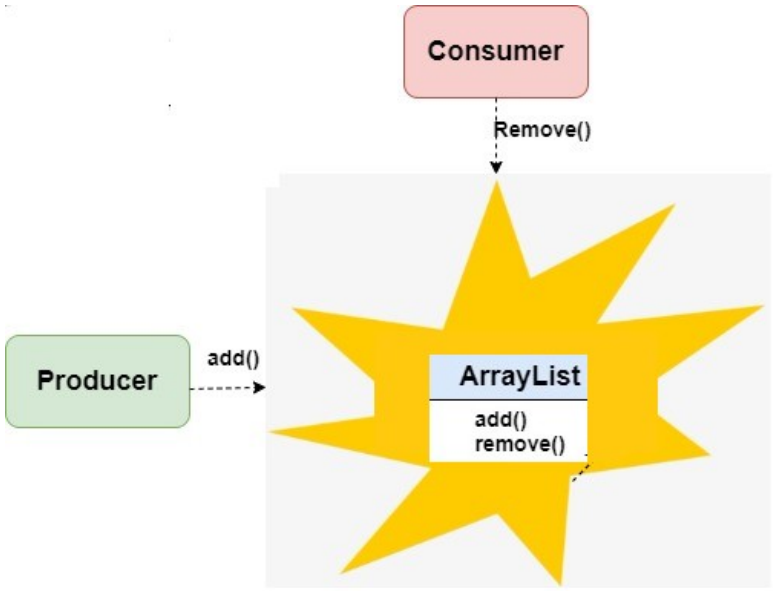
- interfaccia: `Map`
  - `HashMap` (implementazione di `Map`) non è un'implementazione di `Collection`,
  - realizza una struttura dati "dizionario" che associa termini chiave (univoci) a valori
- `Collections` (con la 's' finale!) contiene metodi utili per l'elaborazione di collezioni di qualunque tipo:
  - ordinamento
  - calcolo di massimo e minimo
  - rovesciamento, permutazione, riempimento di una collezione
  - confronto tra collezioni (elementi in comune, sottocollezioni, …)
  - aggiungere un wrapper di sincronizzazione ad una collezione

JAVA COLLECTIONS E THREAD SAFENESS

- collezioni non thread safe, non offrono alcun supporto per la sincronizzazione dei threads
  - `java.util.HashMap`
  - `Java.util.LinkedList`
  - `java.util.ArrayList`
- alternative possibili per garantire la thread safeness
  - thread safe collections, sincronizzate automaticamente da JAVA
    - `java.util.Vector`
    - `java.util.Hashtable`
  - synchronized collections
  - concurrent collections
    - introdotte in `java.util.concurrent`

JAVA COLLECTIONS E THREAD SAFENESS

- `Arraylist` non è una classe threadsafe
  - add non è un'operazione atomica
    - determina quanti elementi ci sono nella lista
    - determina il punto esatto del nuovo elemento
    - incrementa il numero di elementi della lista
    - se si eseguono due add in modo concorrente lo stato della struttura può essere inconsistente
  - analogamente per la `remove`
- `Vector` è una classe threadsafe: *ma quanto costa la sincronizzazione?*



VECTOR ED ARRAYLIST: "UNDER THE HOOD"

```
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
public class VectorArrayList {
public static void addElements(List<Integer> list)
   {for (int i=0; i< 1000000; i++)
        {list.add(i);} }
public static void main (String args[]){
   final long start1 =System.nanoTime();
   addElements(new Vector<Integer>());
   final long end1=System.nanoTime();
   final long start2 =System.nanoTime();
   addElements(new ArrayList<Integer>());
   final long end2=System.nanoTime();
   System.out.println("Vector time "+ (end1-start1));
   System.out.println("ArrayList time "+ (end2-start2)); }}
```

```
Vector time 74494150
ArrayList time 48190559
```

SYNCHRONIZED COLLECTIONS

- synchronized collection wrappers
  - "incapsulano" ogni metodo in un blocco sincronizzato
  - trasformano una `Collection` non thread safe in una thread safe
  - utilizzano un'unica "mutual exclusion lock" intrinseca per tutta la collezione, gestita dalla `JVM`
- metodi definiti nella interfaccia `Collections`
- "conditionally thread safe" collections

Collections Method
`synchronizedCollection(coll)`
`synchronizedCollection(list)`
`synchronizedCollection(map)`
`synchronizedCollection(map)`

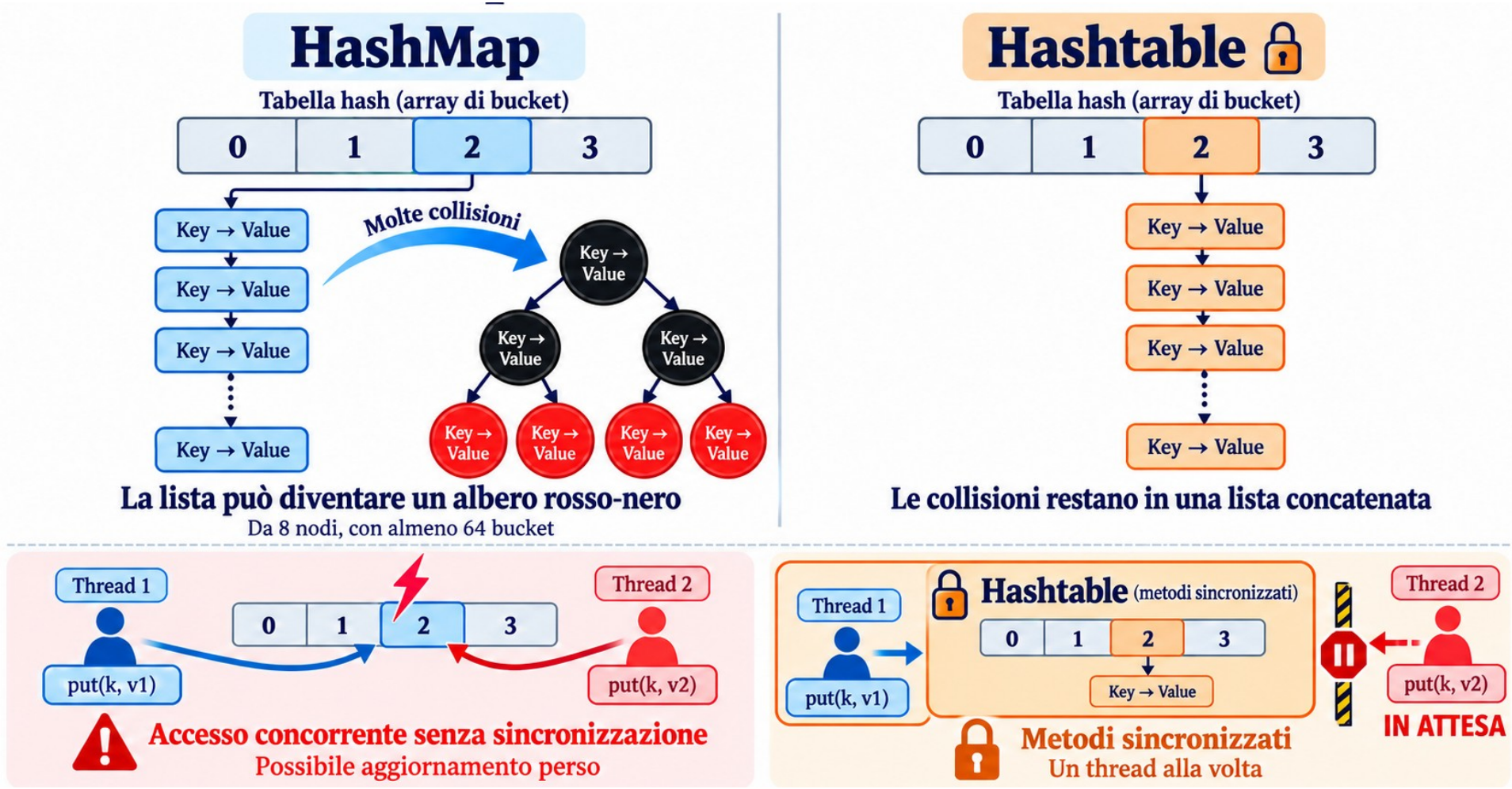


SYNCHRONIZED COLLECTIONS: VALUTAZIONE

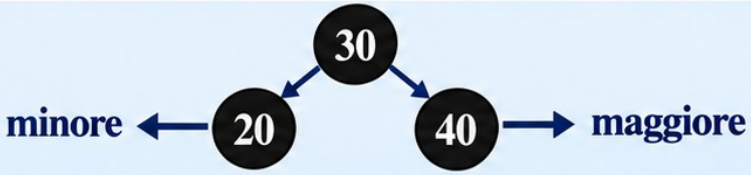
```
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
public class VectorArrayList {
  public static void addElements(List<Integer> list)
    {for (int i=0; i< 1000000; i++)
        {list.add(i);}  }
  public static void main (String args[]){
   final long start1 =System.nanoTime();
   addElements(new ArrayList<Integer>());
   final long end1=System.nanoTime();
   final long start2 =System.nanoTime();
   addElements(Collections.synchronizedList(new ArrayList<Integer>()));
   final long end2=System.nanoTime();
   System.out.println("ArrayList time "+(end1-start1));
   System.out.println("SynchronizedArrayList time "+(end2-start2));}}
```

ArrayList time 50677689
SynchronizedArrayList time 62055651

HASHMAP E HASHTABLE: BUCKET E THREAD



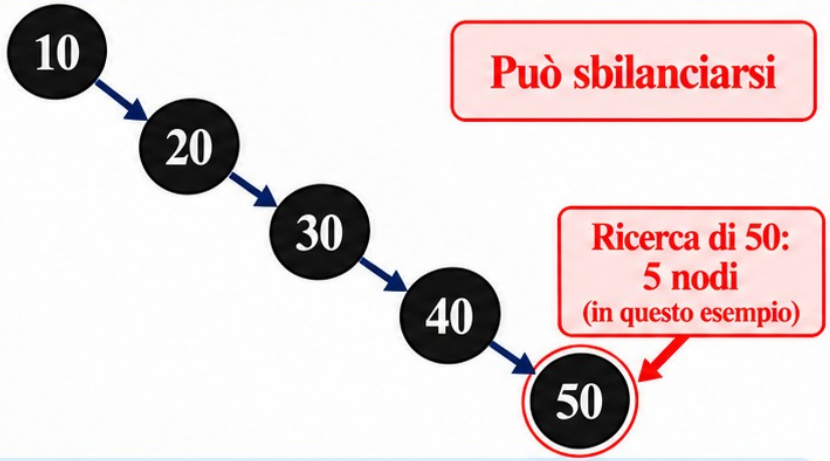
VANTAGGI DI UN ALBERO ROSSO NERO



Albero binario di ricerca:
a sinistra valori minori,
a destra valori maggiori.

Albero binario di ricerca semplice

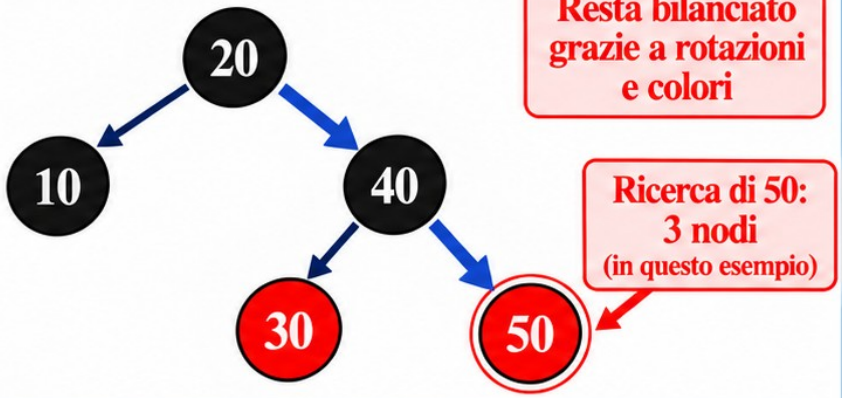
Inserisco: 10, 20, 30, 40, 50



Caso peggiore: $O(n)$

Albero rosso-nero

Inserisco: 10, 20, 30, 40, 50



Caso peggiore: $O(\log n)$

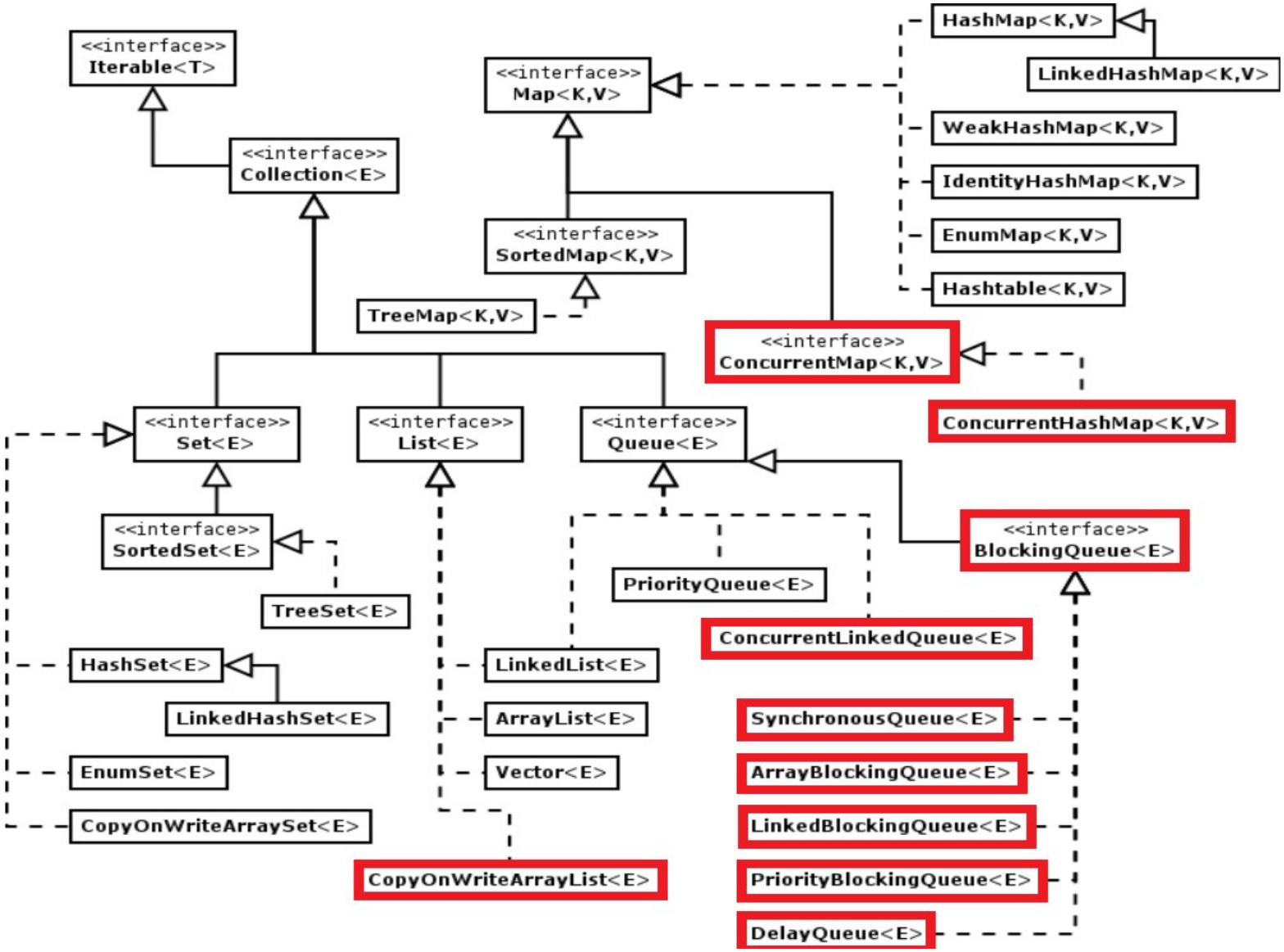
Con molti elementi, evitare una catena mantiene la ricerca efficiente.

JAVA CONCURRENCY FRAMEWORK

- sviluppato in parte da Doug Lea
  - disponibile per tre anni come insieme di librerie JAVA non standard
  - quindi integrazione in JAVA 5.0

- tre i package principali, in rosso alcuni argomenti di questa e della prossima lezione
  - java.util.concurrent
    - Executor, concurrent collections, semaphores,...
  - java.util.concurrent.atomic
    - AtomicBoolean, AtomicInteger,...
  - java.util.concurrent.locks
    - Condition
    - Lock
    - ReadWriteLock

CONCURRENT COLLECTIONS (IN ROSSO)



CONCURRENT COLLECTIONS

- evoluzione delle precedenti librerie basata sull'esperienza nel loro utilizzo
- fine-grain locking, non bloccano l'intera collezione
  - concurrent reads, writes parzialmente concorrenti
- iteratori fail safe/weakly consistent
  - restituiscono tutti gli elementi che erano presenti nella collezione quando l'iteratore è stato creato
  - possono restituire o meno elementi aggiunti in concorrenza
- forniscono alcune utili operazioni atomiche composte da più operazioni elementari
- `ConcurrentHashMap <K,V>`, sincronizzazione ottimizzata, diverse operazioni atomiche
  - `put-if-absent`, `remove-if-equal`, `replace-if-equal`
- `Blocking Queue` che abbiamo utilizzato per i threadpool

HASH MAP E HASH TABLE

- `ConcurrentHashMap`
  - introdotta in `JAVA 5`, nella libreria `java.util.concurrent`
  - thread safe, è un'evoluzione di `HashTable ed HashMap`
- differenza principale
  - `HashMap` (da `JAVA 1.2`) non è *threadsafe*
  - `HashTable` (da `JAVA 1.0`) è *threadsafe*
  - perchè un'altra collezione thread safe?

CONCURRENTHASH MAP: MOTIVAZIONE

```
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class TestCollections {
     public static void main(String[] args) throws Exception {
               evaluatingThePerformance(new Hashtable<String,Integer>(),5);
               evaluatingThePerformance(Collections.synchronizedMap
                                        (new HashMap<String,Integer>()),5);
               evaluatingThePerformance
                    (new ConcurrentHashMap<String,Integer>(),5);}
```

CONCURRENTHASH MAP: MOTIVAZIONE

```
public static void evaluatingThePerformance
                    (Map<String,Integer> maptoEvalPerf, int size) throws Exception {
      long averageTime = 0;
      for(int j=0; j<5;j++) {
         ExecutorService executorService = Executors.newFixedThreadPool(size);
         long startTime = System.nanoTime();
         for(int i=0; i<5;i++) {
            executorService.execute(new Runnable ()
                                    {public void run()
                                          {performTest(maptoEvalPerf);}});
            }
        executorService.shutdown();
        executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
        long entTime = System.nanoTime();
        long totalTime = (entTime - startTime) / 1000000L; averageTime += totalTime;
        System.out.println("500K entries added/retrieved by each thread in "+ totalTime+ "
                                                                              ms");}

       System.out.println("For " + maptoEvalPerf.getClass() + " the average time is " +
                                                          averageTime / 5 + "ms\n");}}
```

CONCURRENTHASH MAP: MOTIVAZIONE

```
public static void performTest(Map<String,Integer> maptoEvaluateThePerformance)
 {
   for(int i=0; i<500000; i++) {
    Integer randomNumber = (int) Math.ceil(Math.random() * 550000);
    Integer value =
        maptoEvaluateThePerformance.get(String.valueOf(randomNumber));
    // Put value
    maptoEvaluateThePerformance.put(String.valueOf(randomNumber),randomNumber);

      }
   }
```

- test: 500000 put e get sulla tabella passata come parametro
- attiviamo un threadpool con k threads che eseguono in parallelo il codice precedente e prendiamo i tempi di esecuzione
- ripetiamo l'esperimento un certo numero di volte e calcoliamo la media dei tempi impiegati

CONCURRENTHASH MAP: MOTIVAZIONE

```
500K entried added/retrieved by each thread in 1393 ms
500K entried added/retrieved by each thread in 1506 ms
500K entried added/retrieved by each thread in 1535 ms
500K entried added/retrieved by each thread in 1407 ms
500K entried added/retrieved by each thread in 1468 ms
For class java.util.Hashtable the average time is 1461ms
500K entried added/retrieved by each thread in 1343 ms
500K entried added/retrieved by each thread in 1530 ms
500K entried added/retrieved by each thread in 1330 ms
500K entried added/retrieved by each thread in 1243 ms
500K entried added/retrieved by each thread in 1389 ms
For class java.util.Collections$SynchronizedMap the average time is 1367ms

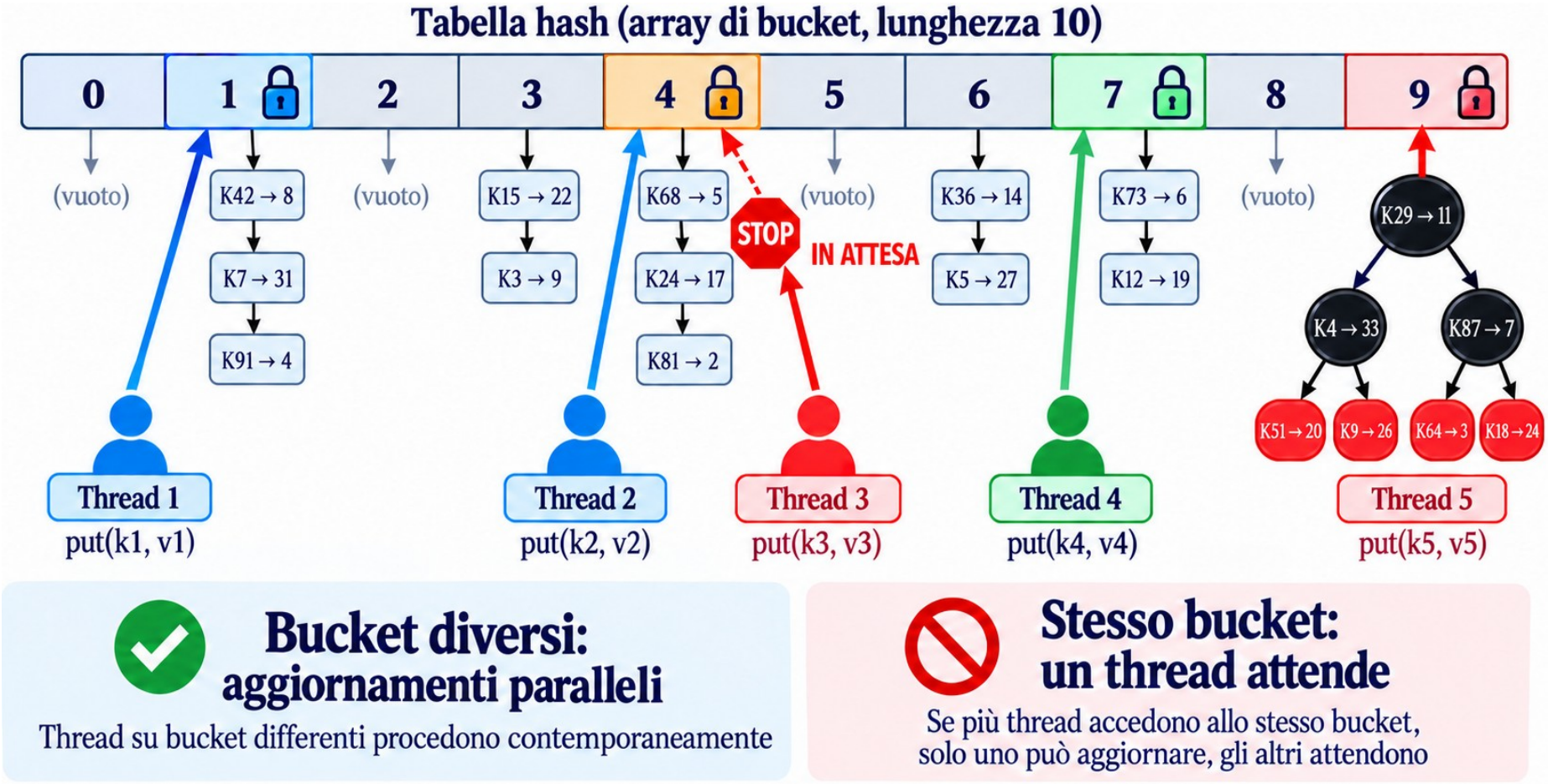
500K entried added/retrieved by each thread in 420  ms
500K entried added/retrieved by each thread in 322  ms
500K entried added/retrieved by each thread in 404  ms
500K entried added/retrieved by each thread in 385  ms
500K entried added/retrieved by each thread in 354  ms
For class java.util.concurrent.ConcurrentHashMap the average time is 377ms
```

CONCURRENT HASH MAP

- le slide precedenti mostrano che la `ConcurrentHashMap` è molto più efficiente delle precedenti versioni di tabelle chiave-valore

- idea fondamentale: usare una diversa strategia di locking che offra migliore concorrenza e scalabilità
  - introduce un array di segmenti: ogni segmento punta ad una HashMap
  - fine grained locking
    - una lock per ogni segmento, lock striping
    - numero di segmenti determina il livello di concorrenza
  - modifiche simultanee possibili, se modificano segmenti diversi
    - 16 o più threads possono operare in parallelo su segmenti diversi
    - lettori possono accedere in parallelo a modifiche

CONCURRENT HASH MAP INTERNAL



Bucket diversi: aggiornamenti paralleli

Thread su bucket differenti procedono contemporaneamente

Stesso bucket: un thread attende

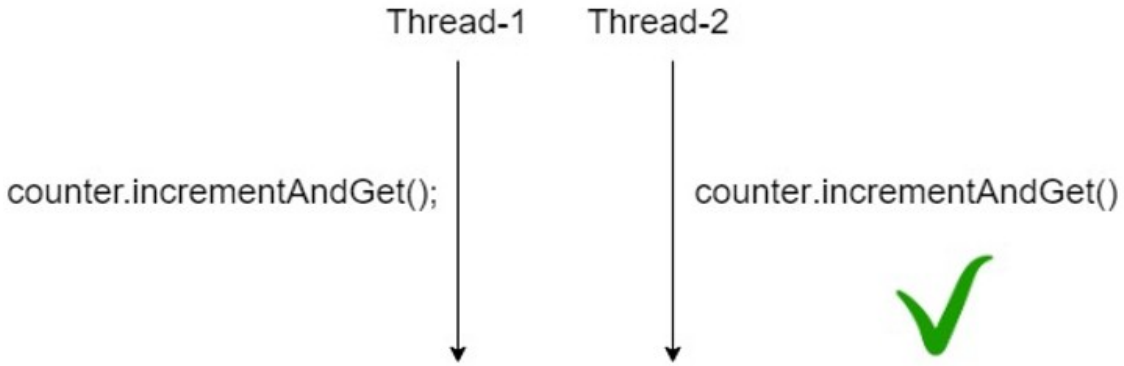
Se più thread accedono allo stesso bucket, solo uno può aggiornare, gli altri attendono

ATOMIC VARIABLES

- come si incrementa in modo atomico una posizione della `ConcurrentHAshMap`?
- aggiornamento: `get + incremento + put`
- lock associata ad ogni bucket protegge da put concorrenti, ma non rende atomica la sequenza `get + incremento + put`
- soluzione corretta

```
ConcurrentHashMap<String, AtomicInteger> mappa = new ConcurrentHashMap<>();
mappa.put("A", new AtomicInteger(0)); // inizializzazione
mappa.get("A").incrementAndGet();     // incremento atomico
```

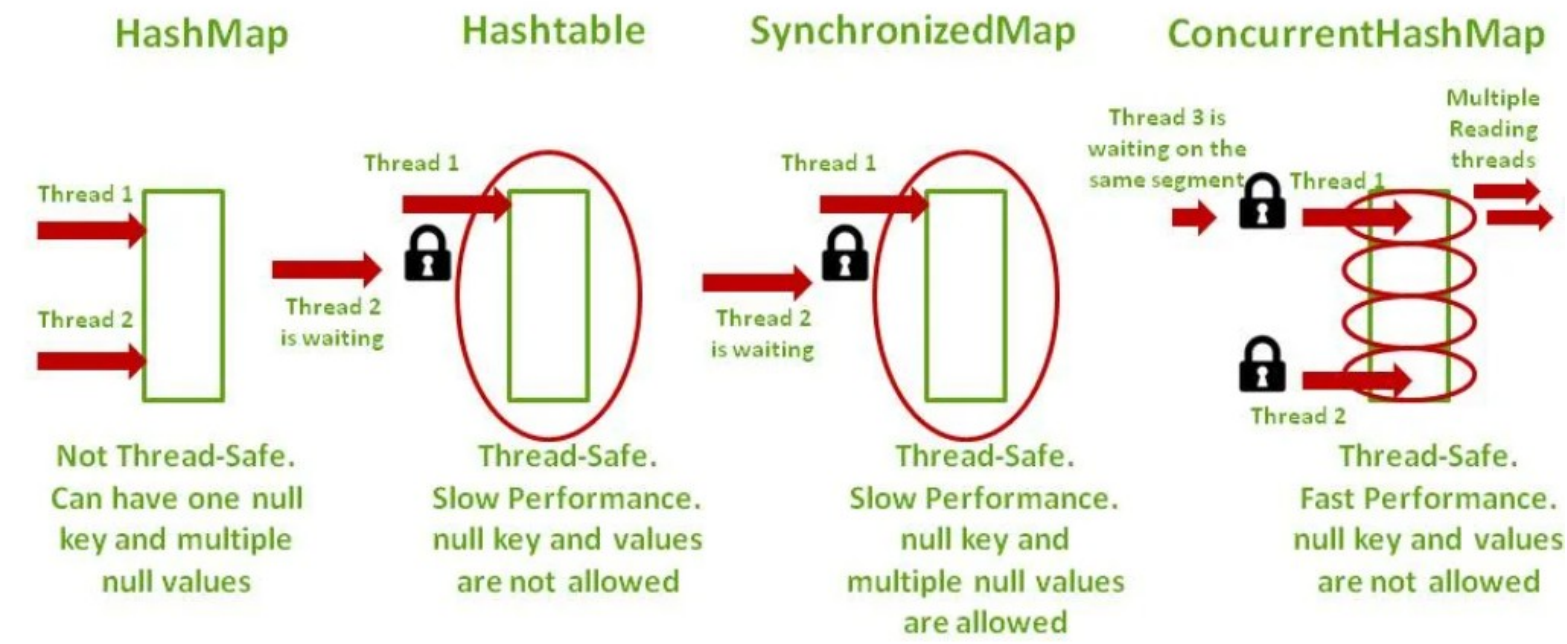
ATOMIC VARIABLES




```
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`
  - ...

STRUTTURE HASH: RECAP



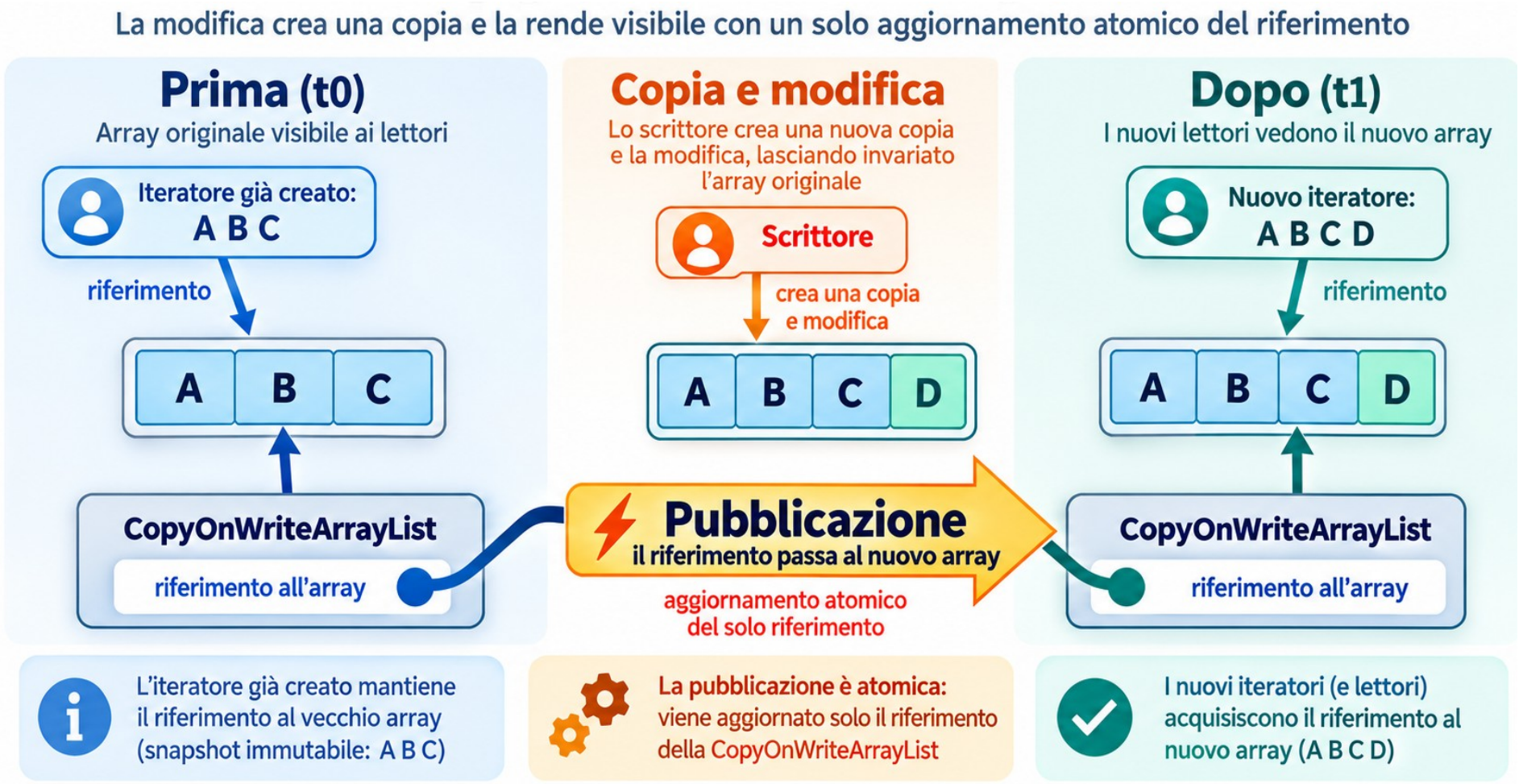
COPYONWRITEARRAYLIST

- `ArrayList` è una struttura dati non thread safe
- versione thread safe: `CopyOnWriteArrayList`
- durante l'aggiunta o la rimozione di elementi, `CopyOnWriteArrayList`
  - crea una nuova copia dell'array
  - aggiunge o rimuove gli elementi da questa copia
  - poi assegna i valori all'array originale dopo che l'operazione di aggiunta o rimozione è stata completata con successo.
- separazione tra letture e scritture
  - quando un thread scrive (aggiunge o rimuove) un elemento, non influisce sugli altri thread che leggono gli elementi,
  - migliora l'efficienza della concorrenza nelle operazioni di lettura.

COPYONWRITEARRAYLIST

- `ArrayList` è una struttura dati non thread safe
- versione thread safe: `CopyOnWriteArrayList`
- durante l'aggiunta o la rimozione di elementi, `CopyOnWriteArrayList`
  - crea una nuova copia dell'array
  - aggiunge o rimuove gli elementi da questa copia
  - poi assegna i valori all'array originale dopo che l'operazione di aggiunta o rimozione è stata completata con successo.
- separazione tra letture e scritture
  - quando un thread scrive (aggiunge o rimuove) un elemento, non influisce sugli altri thread che leggono gli elementi,
  - migliora l'efficienza della concorrenza nelle operazioni di lettura.

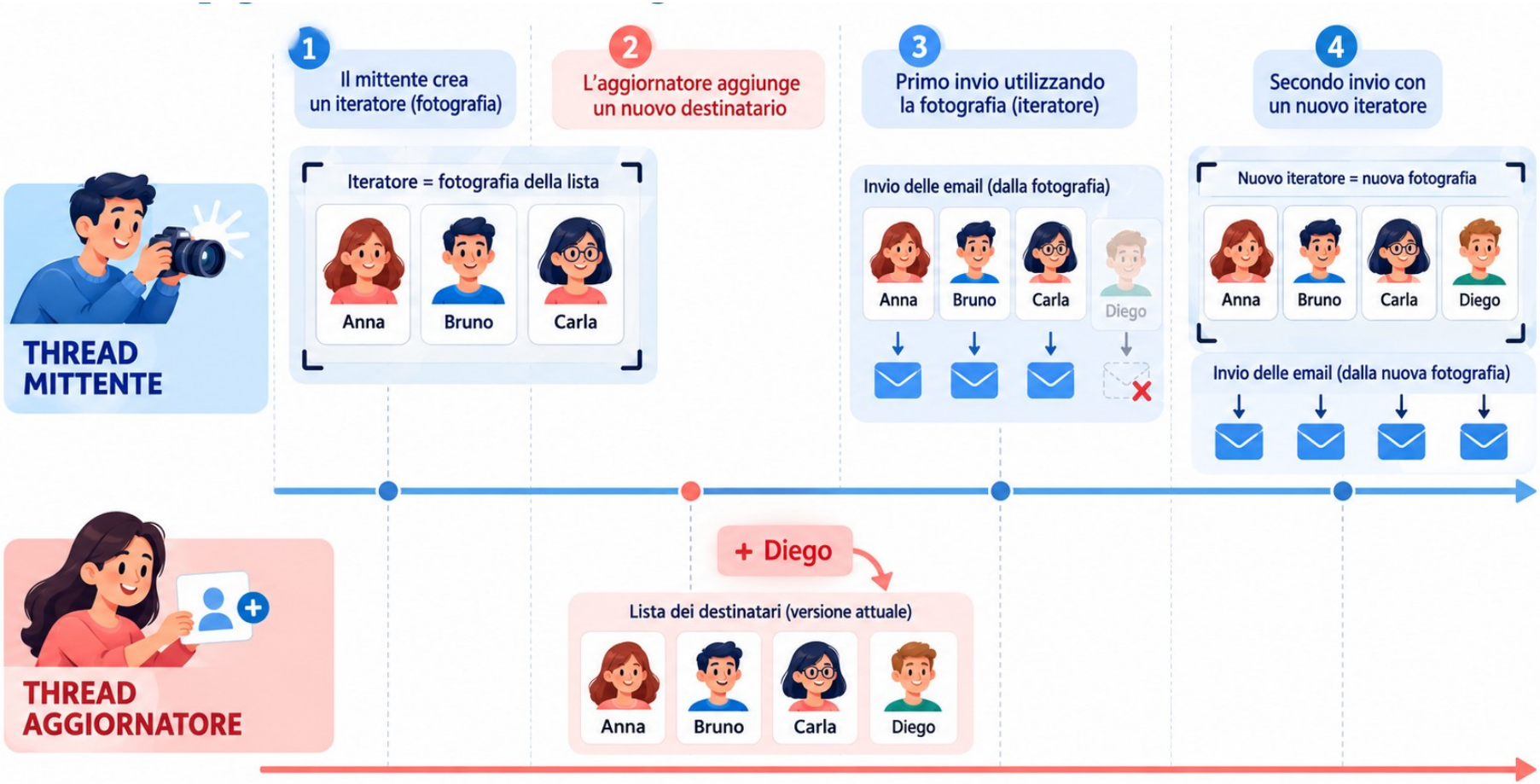
COPYONWRITEARRAYLIST



NEWSLETTER: CHI RICEVE IL NUOVO MESSAGGIO?

- una newsletter ha inizialmente tre iscritti: "Anna", "Bruno" e "Carla"
- il programma usa due thread che condividono una `CopyOnWriteArrayList<String>` contenente i destinatari.
- il thread Mittente prepara l'elenco per un primo invio, attende circa un secondo e spedisce il primo messaggio. Subito dopo spedisce un secondo messaggio consultando nuovamente la lista. Nel frattempo, il thread `AggiornatoreDestinatari` attende circa `300` millisecondi e aggiunge "`Diego`" agli iscritti.
- realizzate le tre classi `Mittente`, `AggiornatoreDestinatari` e `NewsLetterDemo`
- mostrate a chi viene inviato ciascun messaggio e spiegate perché, nell'esecuzione prevista, `Diego` non riceve il primo messaggio ma riceve il secondo

NEWSLETTER: CHI RICEVE IL NUOVO MESSAGGIO?



NEWSLETTER: CHI RICEVE IL NUOVO MESSAGGIO?

```
package NewsLetterDemo;
import java.util.concurrent.CopyOnWriteArrayList;

public class NewsLetterDemo {
    public static void main(String[] args) throws InterruptedException {
        CopyOnWriteArrayList<String> destinatari =
                new CopyOnWriteArrayList<>();

        destinatari.add("Anna");
        destinatari.add("Bruno");
        destinatari.add("Carla");

        Mittente mittente = new Mittente(destinatari);
        AggiornatoreDestinatari aggiornatore =
                new AggiornatoreDestinatari(destinatari);

        mittente.start();
        aggiornatore.start();

        mittente.join();
        aggiornatore.join();
    }
}
```

NEWSLETTER: CHI RICEVE IL NUOVO MESSAGGIO?

```
package NewsLetterDemo;

import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList;

class Mittente extends Thread {
    private final CopyOnWriteArrayList<String> destinatari;

    public Mittente(CopyOnWriteArrayList<String> destinatari) {
        this.destinatari = destinatari;
    }

    @Override
    public void run() {
        // Fotografia dei destinatari del primo invio
        Iterator<String> primoGiro = destinatari.iterator();

        try {
            Thread.sleep(1000);

            System.out.println("Primo messaggio:");
            while (primoGiro.hasNext()) {
                System.out.println("  Messaggio inviato a " + primoGiro.next());
            }

            System.out.println("\nSecondo messaggio:");
            for (String nome : destinatari) {
                System.out.println("  Messaggio inviato a " + nome);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

NEWSLETTER: CHI RICEVE IL NUOVO MESSAGGIO?

```
package NewsLetterDemo;

import java.util.concurrent.CopyOnWriteArrayList;



class AggiornatoreDestinatari extends Thread {
    private final CopyOnWriteArrayList<String> destinatari;

    public AggiornatoreDestinatari(CopyOnWriteArrayList<String> destinatari) {
        this.destinatari = destinatari;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(300);
            destinatari.add("Diego");
            System.out.println("Diego si è iscritto!\n");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

NEWSLETTER: CHI RICEVE IL NUOVO MESSAGGIO?

```
Diego si è iscritto!

Primo messaggio:
  Messaggio inviato a Anna
  Messaggio inviato a Bruno
  Messaggio inviato a Carla

Secondo messaggio:
  Messaggio inviato a Anna
  Messaggio inviato a Bruno
  Messaggio inviato a Carla
  Messaggio inviato a Diego
```

CREAZIONE DI ITERATORE

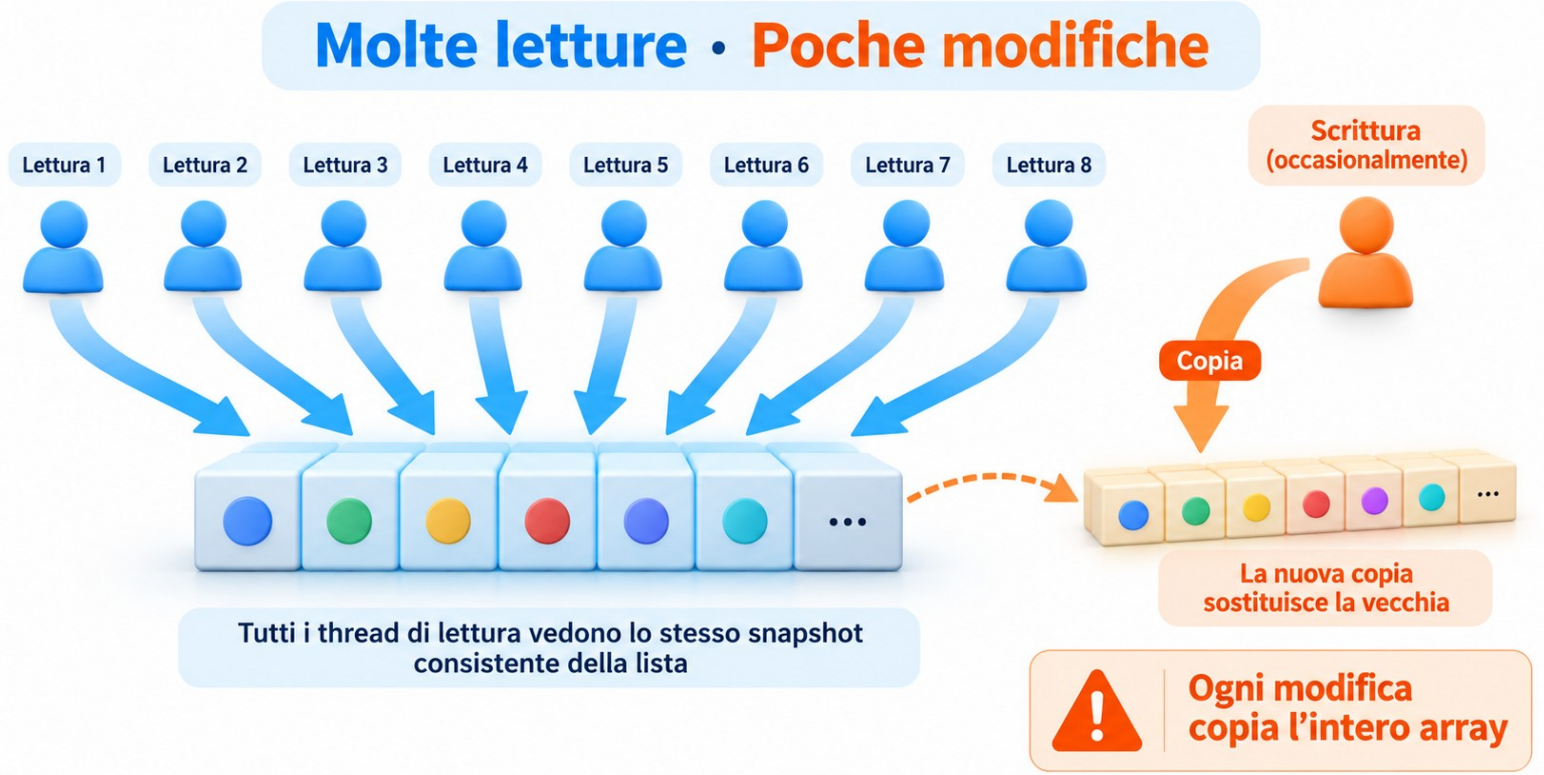
NOTA 1: Java crea un nuovo iteratore quando per ogni nuovo `for`

```
for (String nome : destinatari) {
    System.out.println(nome);
}
```

NOTA 2:

- le chiamate a `sleep()` rendono probabile che `Diego` si iscriva tra la creazione del primo iteratore e il secondo invio, ma non lo garantiscono.
- `sleep()` sospende un thread per un certo tempo; non assicura quando gli altri thread verranno eseguiti. Su una macchina molto occupata l'ordine potrebbe cambiare.
- per garantire sempre il risultato atteso, i thread dovrebbero coordinarsi esplicitamente.

COPYONWRITEARRAYLIST: QUANDO USARLA



- ottimizzato per scenari in cui il numero di letture supera di molto quello delle scritture
- letture parallele senza sincronizzazione
- se il contenitore contiene una grande quantità di dati, ogni operazione di aggiunta o rimozione può risultare molto dispendiosa in termini di tempo

ASSIGNMENT 4: UN NEGOZIO ONLINE

- un negozio online ha un catalogo condiviso, inizialmente il catalogo contiene 3 articoli, ad esempio `Telecamere`, `Cuffie`, `Tastiere`
- `20` clienti consultano il catalogo contemporaneamente. Ogni volta che un cliente visualizza un prodotto, l'app incrementa il numero di visualizzazioni di quel prodotto.
- realizzate il programma usando una `CopyOnWriteArrayList<String>` per il catalogo e una `ConcurrentHashMap<String, Integer>` per i contatori delle visualizzazioni.
- i primi `10` clienti accedono al catalogo. Un thread gestore aggiunge quindi un nuovo articolo, il `Mouse`.
- solo dopo l'aggiunta, gli altri `10` clienti prendono la propria fotografia.
- al termine del programma, verificate i valori dei diversi contatori