

Reti e Laboratorio III

Modulo Laboratorio III

AA. 2022-2023

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 4

Synchronized and Concurrent Collections

06/10/2022

ANCORA SUL MONITOR: WAIT E NOTIFY

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

“regola d'oro” testare sempre la condizione relativa al `wait` all'interno di un ciclo

- poichè la coda di attesa è unica per tutte le condizioni, un thread potrebbe essere stato risvegliato in seguito al verificarsi di una condizione che poi diventa nuovamente falsa
 - la condizione su cui il thread `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

LOCK INTRINSECHE: BLOCCHI SINCRONIZZATI

- se non si intende sincronizzare un intero metodo, si può **sincronizzare un blocco di codice** all'interno di un metodo
- un esempio semplice:

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

- sincronizzare un intero metodo equivale ad inserire il codice del metodo di un blocco sincronizzato su `this`
- l'oggetto riferito tra parentesi è un “monitor object”
- un thread
 - acquisisce la lock implicita sull'oggetto `this`, quando entra nel blocco sincronizzato
 - la rilascia quando termina il blocco sincronizzato.

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()}
```

- ne vedremo un uso concreto nel caso di classi conditionally thread safe

MONITOR E LOCK: CONFRONTI

- monitor: vantaggi
 - l'unità di sincronizzazione è il metodo: tutte le sincronizzazioni sono visibili esaminando segnatura dei metodi
 - costrutti strutturati. 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

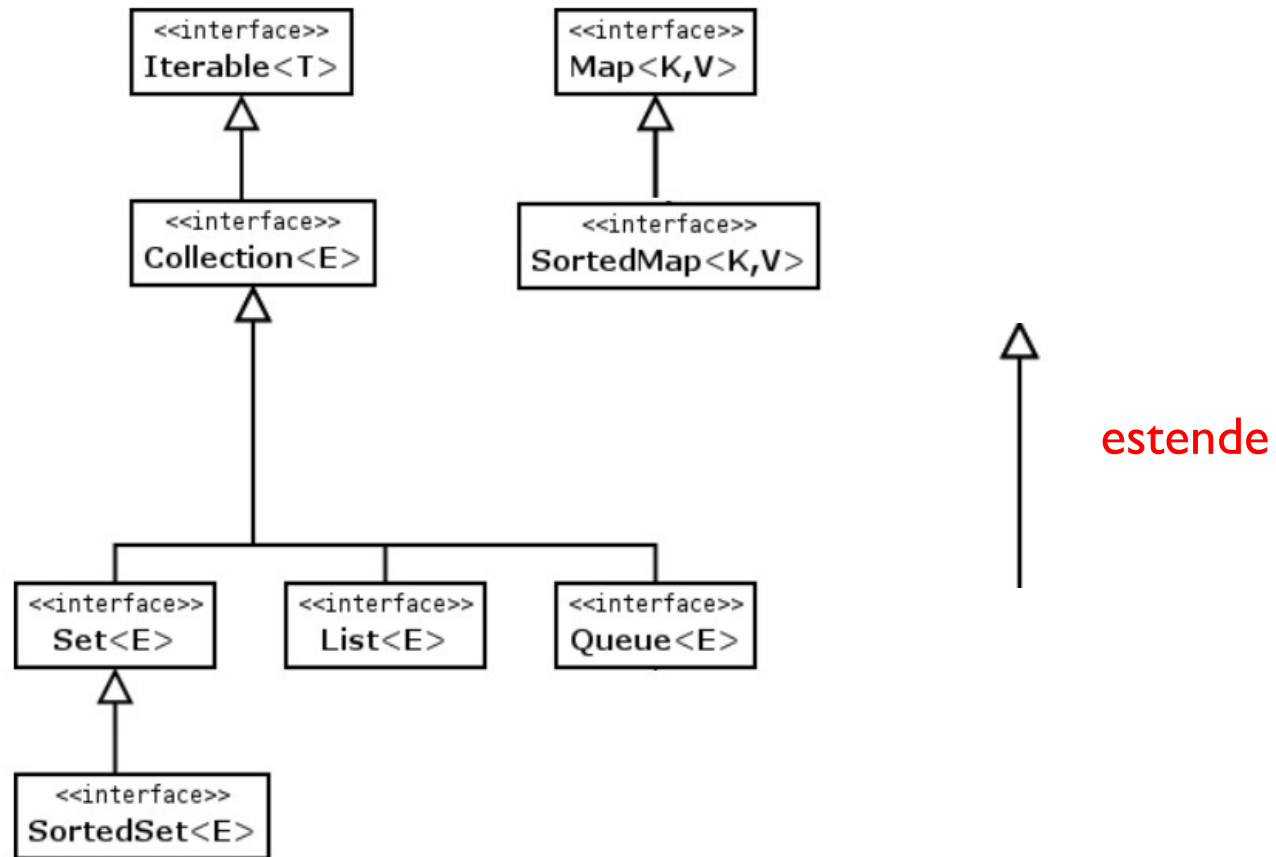
JAVA COLLECTION FRAMEWORK: BREVE RIPASSO

- 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
- che c'è di nuovo in questo corso?
 - **synchronized collections**
 - **concurrent collections**

JAVA COLLECTION FRAMEWORK: BREVE RIPASSO

- 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
- che c'è di nuovo in questo corso?
 - `synchronized collections`
 - `concurrent collections`

JAVA COLLECTION: INTERFACES IN JAVA.UTIL

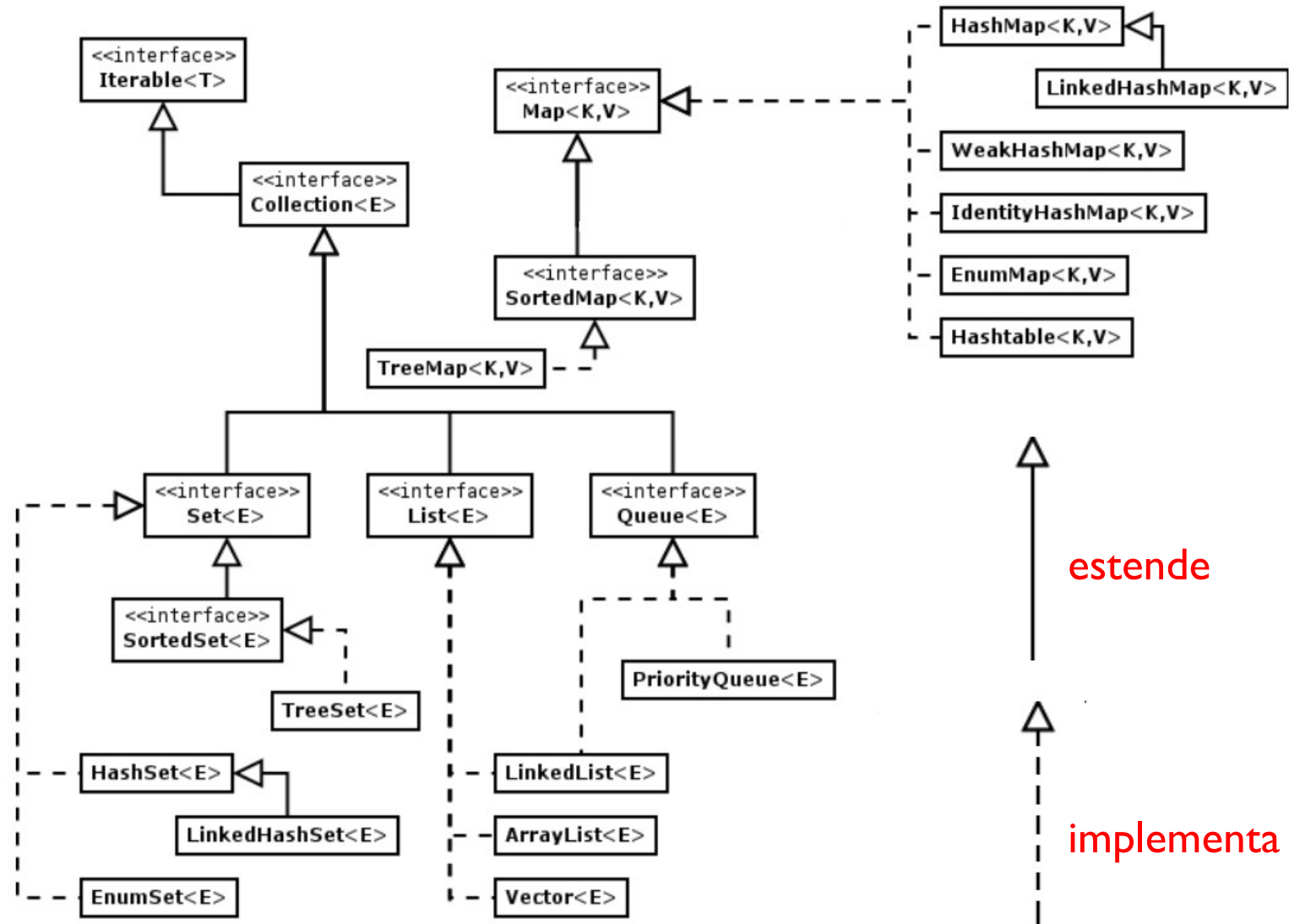


- riportate solo le interfacce principali presenti in `java.util`

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
 - le chiavi possono essere ordinate

JAVA COLLECTION: INTERFACES AND CLASSES

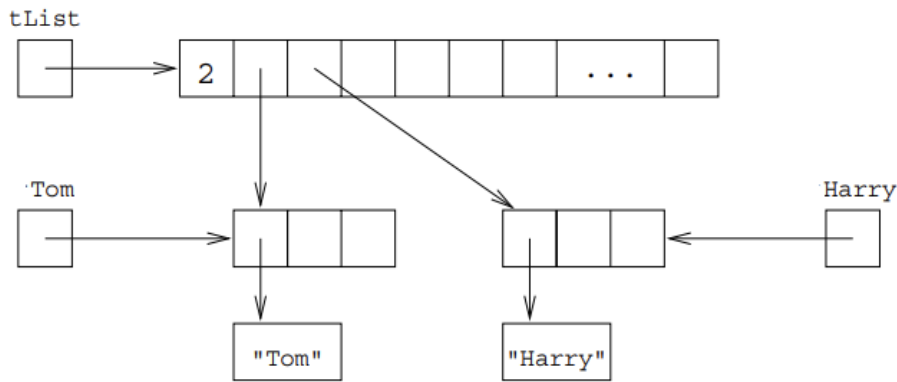


- interfaccia: `Map`
 - `HashMap` (implementazione di `Map`) non è un'implementazione di `Collection`, ma è comunque una struttura dati molto usata
 - realizza una struttura dati “dizionario” che associa termini chiave (univoci) a valori
- `Collections` (con la 's' finale !) contiene metodi utili per 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 COLLECTION FRAMEWORK: ARRAYLIST

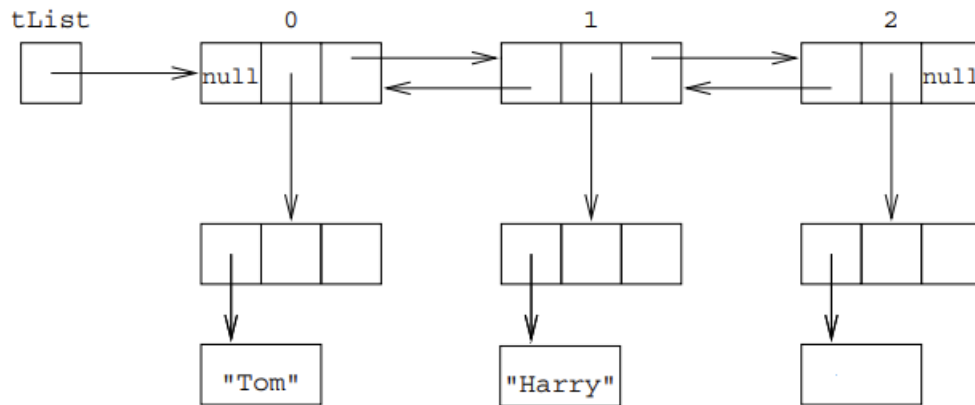
```
public class Person {  
    String name;  
    int age;  
    String type;  
    public Person (String name, int age, String type)  
    {this.name=name;  
     this.age=age;  
     this.type=type;}  
    public String toString ( ) {  
        return this.name+this.age+this.type;}  
    }  
}
```

```
import java.util.*;  
public class PersonList {  
    public static void main (String args[])  
    { Person Tom = new Person("Tom", 45,"professor");  
      Person Harry = new Person("Harry", 20,"student");  
      List <Person> tList=new  
                               ArrayList<Person> ();  
      tList.add(Tom);  
      tList.add(Harry);  
      System.out.println(Tom);  
      System.out.println(Harry);  
      System.out.println(tList.size()); } }
```



JAVA COLLECTION FRAMEWORK: LINKEDLIST

```
public class Person {  
    String name;  
    int age;  
    String type;  
    public Person (String name, int age, String type)  
    {this.name=name;  
     this.age=age;  
     this.type=type;}  
    public String toString ( ) {  
        return this.name+this.age+this.type;}  
    }  
}
```



```
import java.util.*;  
public class PersonList {  
    public static void main (String args[])  
    { Person Tom = new Person("Tom", 45,"professor");  
      Person Harry = new Person("Harry", 20,"student");  
      List <Person> tList=new  
                                  LinkedList<Person> ();  
      tList.add(Tom);  
      tList.add(Harry);  
      System.out.println(Tom);  
      System.out.println(Harry);  
      System.out.println(tList.size());    } }
```

JAVA ITERATORS

- GoF “Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation”
- usato per accedere agli elementi di una collezione, uno alla volta
 - deve conoscere (e poter accedere) alla rappresentazione interna della classe che implementa la collezione
- l'interfaccia `Collection` contiene il metodo `iterator()` che restituisce un iteratore per una collezione
 - le diverse implementazioni di `Collection` implementano il metodo `iterator()` in modo diverso
 - l'interfaccia `Iterator` prevede tutti i metodi necessari per usare un iteratore, senza conoscere alcun dettaglio implementativo

USARE GLI ITERATORI

- schema generale per l'uso di un iteratore

```
import java.util.*;

public class PersonList {

    public static void main (String args[])

    { Person Tom = new Person("Tom", 45, "professor");
      Person Harry = new Person("Harry", 20, "student");
      List <Person> pList=new LinkedList<Person> ();

      ....

      Iterator <Person> tIterator = pList.iterator();
      while (tIterator.hasNext())
      { Person tPerson = (Person) tIterator.next();
        System.out.println(tPerson);
      }
    }
```

- l'iteratore non ha alcuna funzione che lo “resetti”
- una volta iniziata la scansione, non si può fare tornare indietro l'iteratore
- una volta finita la scansione, è necessario creare uno nuovo iteratore

USARE GLI ITERATORI SU HASHMAP

```
public class Employee{  
    private String id;  
    private String name;  
    private String department;  
    public Employee(String id, String name, String department){  
        this.id = id;  
        this.name = name;  
        this.department = department;  
    }  
    public String toString(){  
        return "[" + this.id + " : " + this.name + " : " + this.department +  
        "]" ;}}}
```


USARE GLI ITERATORI SU HASHMAP

```
import java.util.*;

public class EmployeeIterator {

    public static void main (String args[])

    { HashMap<String, Employee> employeeMap = new HashMap<String, Employee>();

      employeeMap.put("emp01", new Employee("emp01", "Tom", "IT"));
      employeeMap.put("emp02", new Employee("emp02", "Jhon", "Supply Chain"));
      employeeMap.put("emp03", new Employee("emp03", "Oliver", "Marketing"));
      employeeMap.put("emp04", new Employee("emp04", "Mary", "IT"));

      Set<Map.Entry<String, Employee>> entrySet = employeeMap.entrySet();
      Iterator<Map.Entry<String, Employee>> iterator = entrySet.iterator();

      System.out.println("Iterate through mappings of HashMap");

      while( iterator.hasNext() ){

          Map.Entry<String, Employee> entry = iterator.next();

          System.out.println( entry.getKey() + " => " + entry.getValue() ); }}}}
```

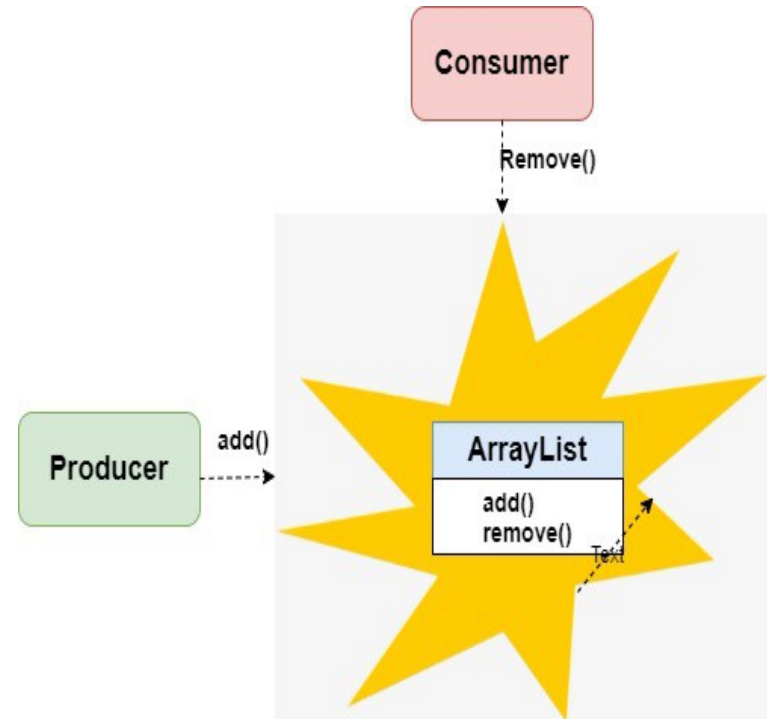
JAVA CONCURRENCY FRAMEWORK

- 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`

- collezioni non thread safe, non offrono alcun supporto per la sincronizzazione dei threads
 - `java.util.Map`
 - `Java.util.LinkedList`
 - `java.util.ArrayList`
- quali soluzioni per la sincronizzazione? Alternative possibili
 - thread safe collections, sincronizzate automaticamente da JAVA
 - `java.util.Vector`
 - `java.util.Hashtable`
 - synchronized collections
 - concurrent collections
 - introdotte in `java.util.concurrent`

COLLEZIONI NON THREAD SAFE

- ArrayList non è una classe threadsafe
 - add non è una 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 |
|---|
| <code>synchronizedCollection(coll)</code> |
| <code>synchronizedCollection(list)</code> |
| <code>synchronizedCollection(map)</code> |
| <code>synchronizedCollection(map)</code> |



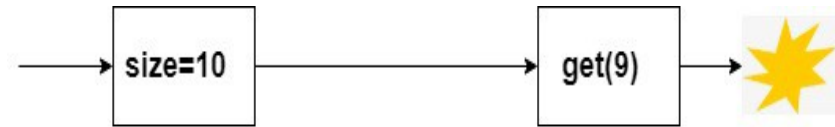
SYNCHRONIZED COLLECTIONS: VALUTAZIONE

```
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));}}
```

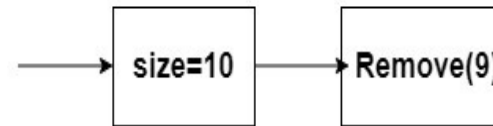
ArrayList time 50677689
SynchronizedArrayList time 62055651

CLASSI CONDITIONALLY THREAD SAFE

```
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); } }
```



- la thread safety garantisce che la safety delle singole operazioni operazioni sulla collezione, ma...
- funzioni che **coinvolgono più di una operazione** possono non essere thread-safe
- Vector è una collezione thread-safe, però perchè in caso di accessi concorrenti, questo programma genera `ArrayIndexOutOfBoundsException`?

CLASSI CONDITIONALLY THREAD SAFE

```
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 è vuota e viene descheduled 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 risultarlo.

USO DEI 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

USO DEI BLOCCHI SINCRONIZZATI

@ThreadSafe

```
public class UnsafeVector{  
    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); }  
    }  
}
```

ITERATORI E ECCEZIONI

- eccezione sollevata dagli iteratori su collezioni, se la collezione viene modificata prima che l'iterazione sia completata

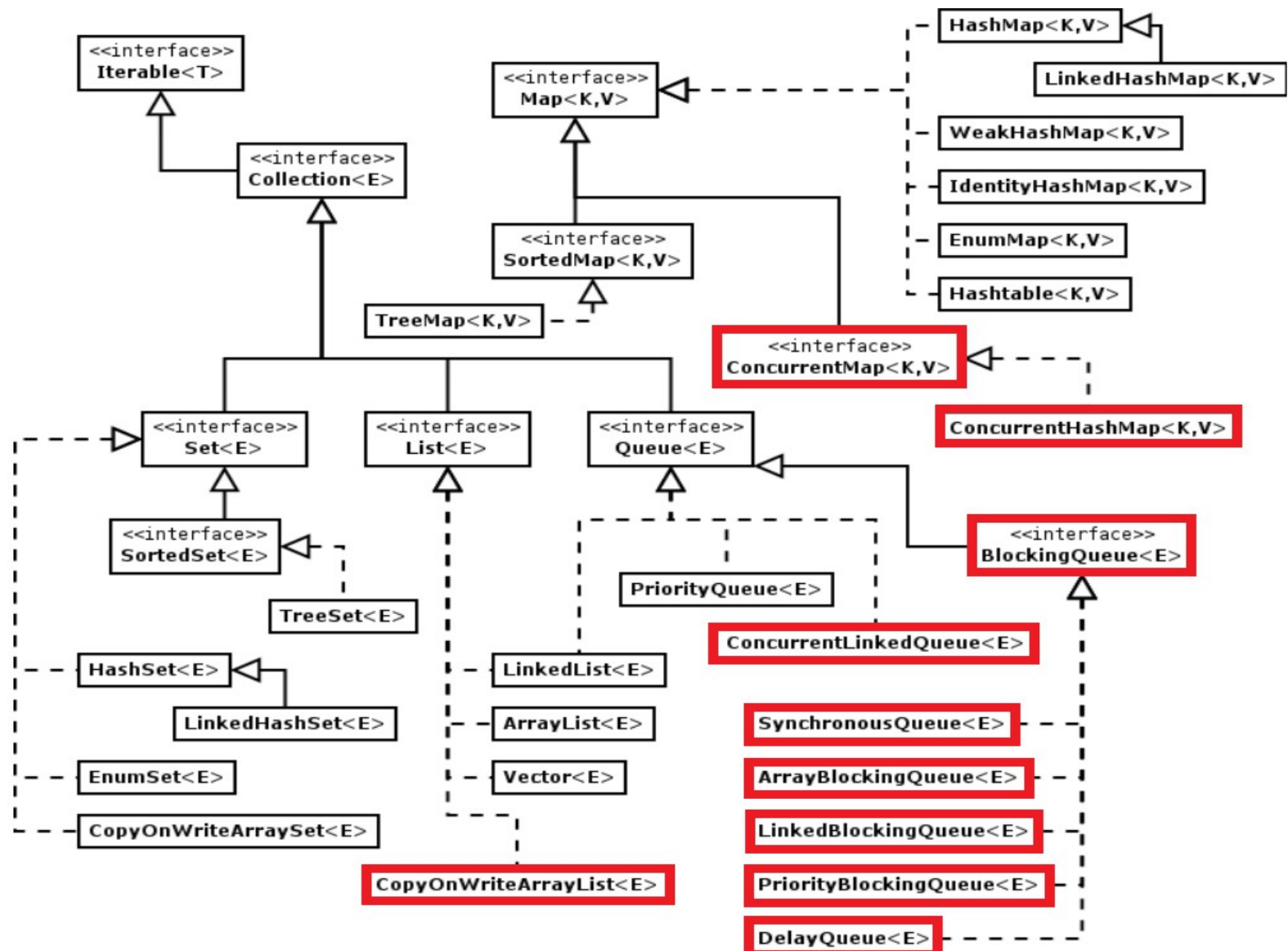
- può essere sollevata anche se il programma è sequenziale

```
for (E element: list)
    if (isBad(element))
        list.remove(element) //ConcurrentModificationException
```

- anche se la collezione è sincronizzata, l'iteratore su di essa può non esserlo

```
synchronized(syncList) {
    Iterator iterator = syncList.iterator( );
    // do stuff with the iterator here
}
```

CONCURRENT COLLECTIONS (IN ROSSO)



CONCURRENT COLLECTIONS

- evoluzione delle precedenti librerie basata sulla 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
- Blocking Queue è una concurrent collections
- la più utilizzata: ConcurrentHashMap $\langle K, V \rangle$, sincronizzazione ottimizzata, diverse operazioni atomiche
 - put-if-absent, remove-if-equal, replace-if-equal

HASH MAP E HASH TABLE

- ConcurrentHashMap
 - introdotta in JAVA 5, nella libreria `java.util.concurrent`
 - thread safe, è una evoluzione di `HashTable` ed `HashMap`
- collezioni che memorizzano coppie chiave/valore
 - usando la tecnica dell'hashing
- differenza principale
 - `HashTable` (da JAVA 1.0) è *threadsafe*
 - `HashMap` (da JAVA 1.2) non è *threadsafe*
 - perchè un'altra collezione thread safe?

CONCURRENT HASH MAP: MOTIVAZIONE

```
import java.util.Map;
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);
        evaluatingThePerformanceWithSynchronizedBlock
                                   (new HashMap<String,Integer>()),5);
        evaluatingThePerformance
                                   (new ConcurrentHashMap<String,Integer>(),5);}
```


CONCURRENT HASH MAP: MOTIVAZIONE

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

CONCURRENT HASH 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 parametri
- 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

CONCURRENT HASH MAP: MOTIVAZIONE

```
public static void performSynchronizedTest(Map<String,Integer>
                                           maptoEvaluateThePerformance) {

    for(int i=0; i<500000; i++) {
        Integer randomNumber = (int) Math.ceil(Math.random() * 550000);

        synchronized (maptoEvaluateThePerformance) {
            Integer Value =
                maptoEvaluateThePerformance.get(String.valueOf(randomNumber));
        }

        synchronized (maptoEvaluateThePerformance) {
            maptoEvaluateThePerformance.put
                (String.valueOf(randomNumber), randomNumber);
        }
    }
}
```

CONCURRENT HASH MAP: MOTIVAZIONE

- il metodo *evaluatingThePerformanceWithSynchronizedBlock* invoca *performSynchronizedTest*, invece di *performTest* (nel run della Runnable)
- per il resto è uguale al metodo della slide precedente

CONCURRENT HASH MAP: MOTIVAZIONE

500K entried added/retrieved by each thread in 1274 ms

500K entried added/retrieved by each thread in 1314 ms

500K entried added/retrieved by each thread in 1279 ms

.....

For class `java.util.Hashtable` the average time is 1283 ms

500K entried added/retrieved by each thread in 1288 ms

500K entried added/retrieved by each thread in 1157 ms

500K entried added/retrieved by each thread in 1249 ms

....

For class `java.util.Collections$SynchronizedMap` the average time is 1237 ms

500K entried added/retrieved in 1238 ms

500K entried added/retrieved in 1392 ms

500K entried added/retrieved in 1375 ms

.....

For class `java.util.HashMapWith SYNCHRONIZED BLOCKS` the average time is 1423 ms

500K entried added/retrieved by each thread in 550 ms

500K entried added/retrieved by each thread in 440 ms

500K entried added/retrieved by each thread in 458 ms

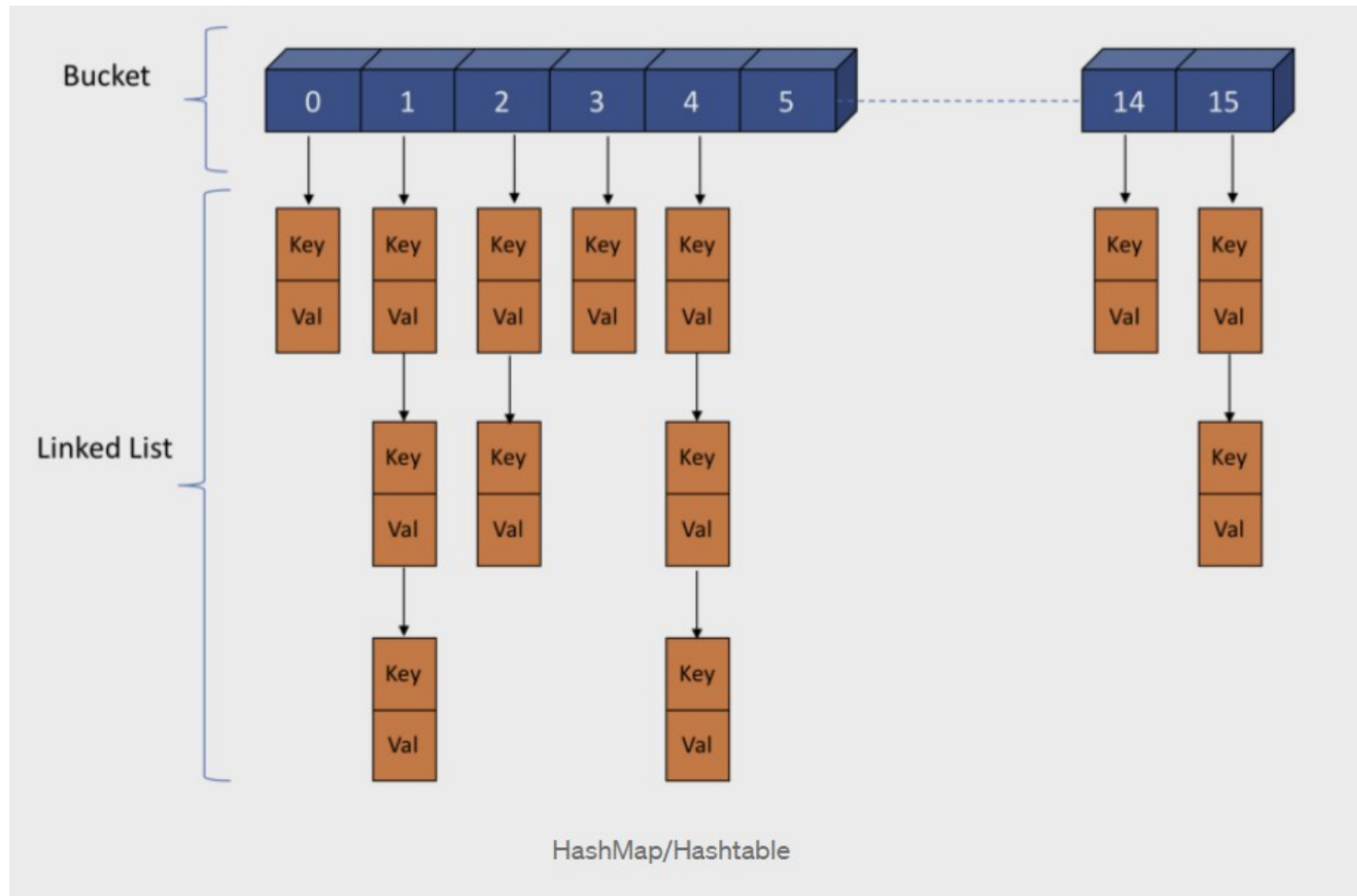
.....

For class `java.util.concurrent.ConcurrentHashMap` the average time is 524 ms

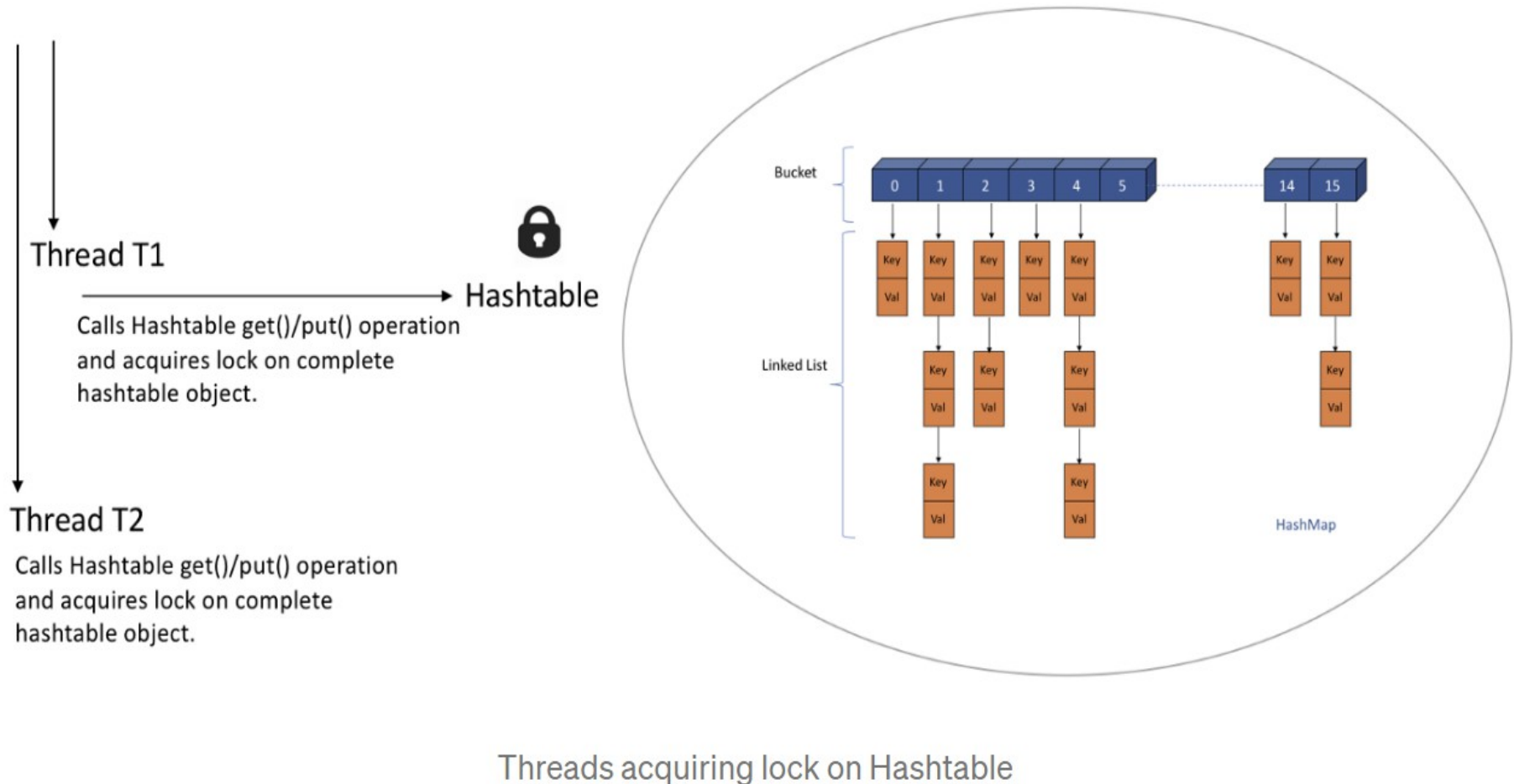
CONCURRENT HASH MAP

- 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

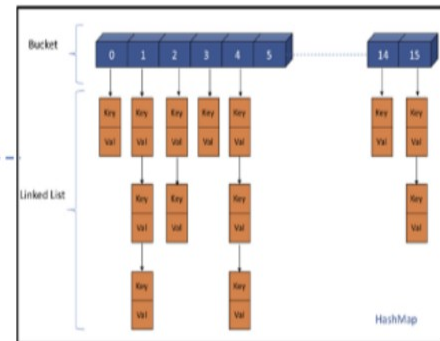
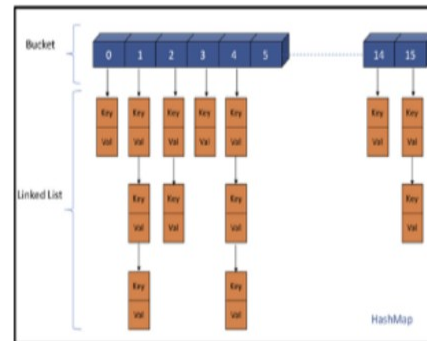
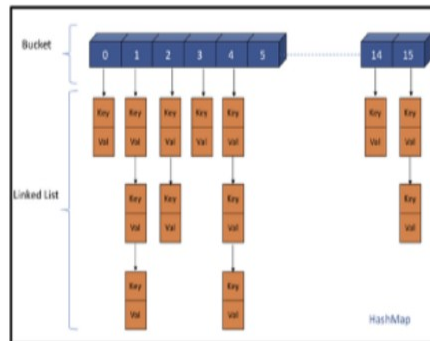
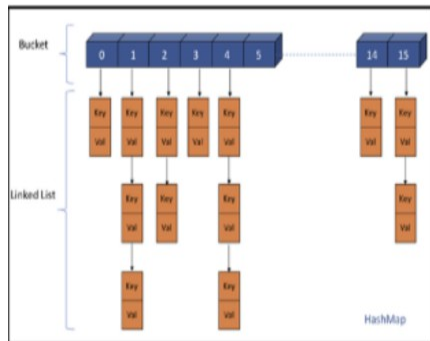
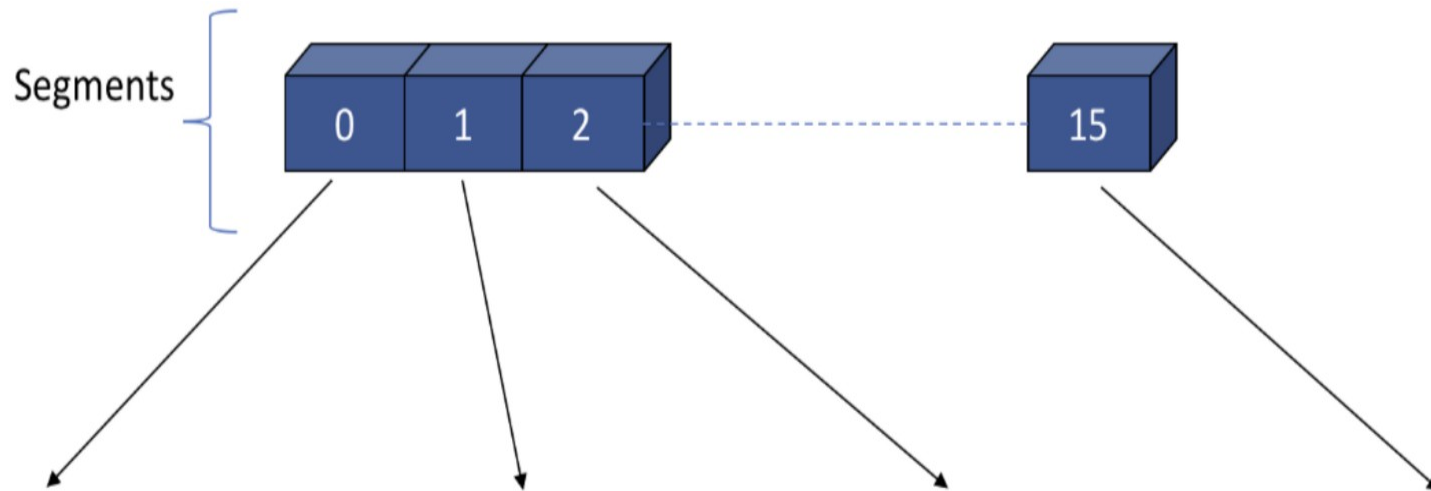
HASHMAP INTERNAL



HASHTABLE AND THREADS

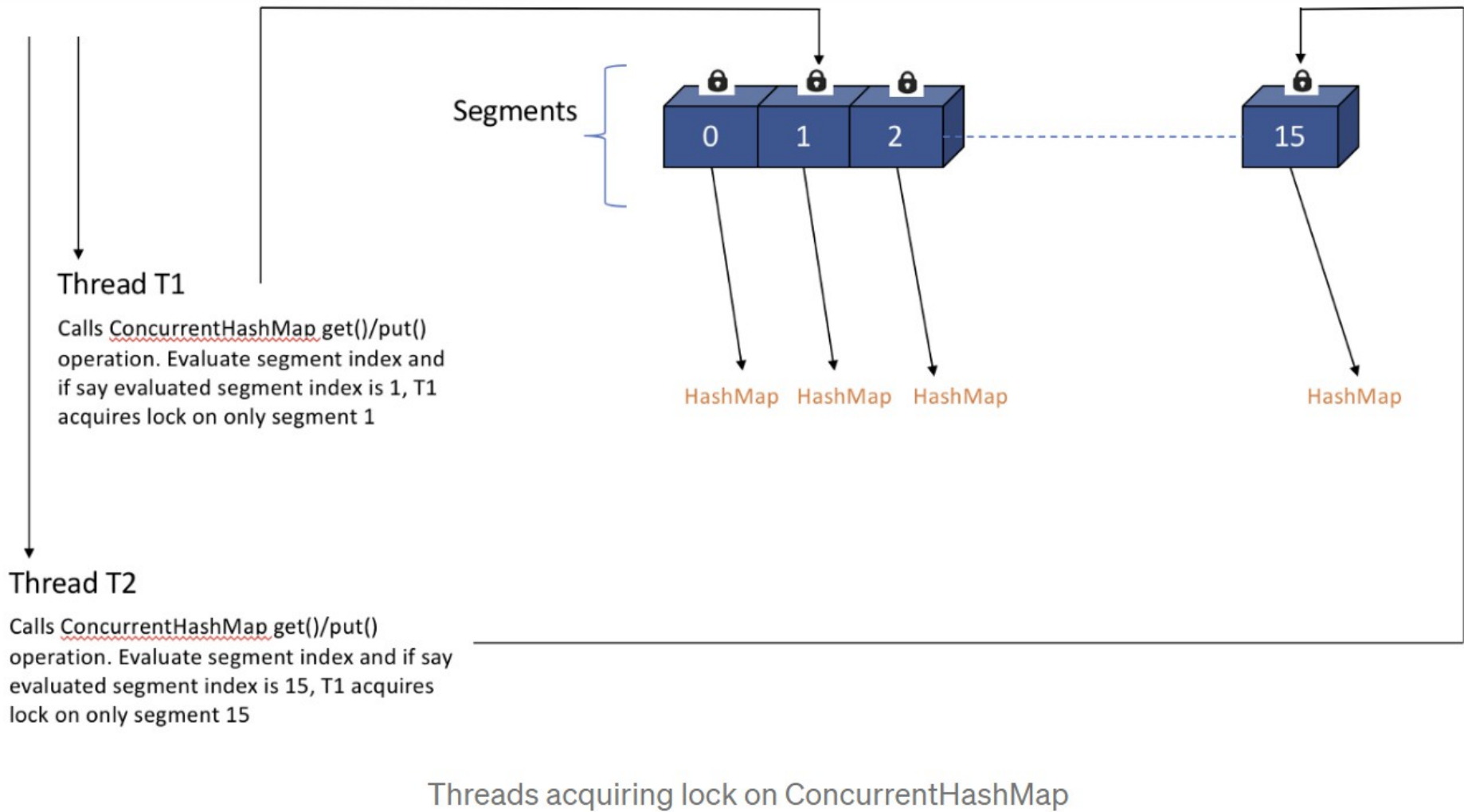


CONCURRENT HASHMAP INTERNAL



- struttura utilizzata fino a JAVA7
- da JAVA8 alberi bilanciati invece di linked lists

CONCURRENTHASHMAP INTERNAL



OPERAZIONI COMPOSTE ATOMICHE

le concurrent collections offrono inoltre un insieme di operazioni composte atomiche

- sequenze di operazioni di uso comune
- definite come una 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...”

OPERAZIONI COMPOSITE: ATOMICITA'

```
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();}}

• GetOrCreate non è atomica
```

```
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();
    }}
```

- t1 e t2 stampano due valori diversi, entrambi associati alla stessa chiave, 5

OPERAZIONI COMPOSITE ATOMICHE

- soluzione: utilizzare istruzioni atomiche composte
- funzioni del tipo “query-then-update”, “test-and-set”
- nel nostro caso 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 COMPOSITE: 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();}}
```

- 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 “non-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)

- **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 una 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

FAIL SAFE: COPYONWRITEARRAYLIST

- riprendiamo l'esempio mostrato nella lezione scorsa: divide et impera con multithreading
- calcolare la somma di tutti i numeri da 1 a n
 - individuare sottointervalli dell'intervallo 1-100
 - creare un task diverso per ogni intervallo: calcola la somma per quell'intervallo
 - sottomettere i task ad un threadpool, somme parziali in parallelo
 - raccogliere le somme parziali per calcolare la somma totale.
- soluzione diversa: le somme parziali non vengono raccolte sequenzialmente, partendo dalla prima somma, ma nell'ordine in cui esse sono disponibili.

FAIL SAFE: COPYONWRITEARRAYLIST

```
import java.util.ArrayList; import java.util.List; import java.util.concurrent.*;

public class Adder {

    public static void main(String[] args) throws ExecutionException, InterruptedException
    { ExecutorService executor = Executors.newFixedThreadPool(5);

        List<Future<Integer>> list = new CopyOnWriteArrayList<Future<Integer>>();

        for (int i = 0; i < 10; i=i+2) {
            Calculator c = new Calculator(i, i + 1);
            list.add(executor.submit(c)); }

        int s=0;

        while (!list.isEmpty() )
        for (Future<Integer> f : list) {
            if (f.isDone()) {System.out.println(f.get());
                s=s+f.get();
                list.remove(f);          } }

        System.out.println("la somma e' "+s);

        executor.shutdown();}}
```

ASSIGNMENT 4: CONTEGGIO OCCORRENZE

- scrivere un programma che conta le occorrenze dei caratteri alfabetici (lettere dalla “A” alla “Z”) in un insieme di file di testo. Il programma prende in input una serie di percorsi di file testuali e per ciascuno di essi conta le occorrenze dei caratteri, ignorando eventuali caratteri non alfabetici (come per esempio le cifre da 0 a 9). Per ogni file, il conteggio viene effettuato da un apposito task e tutti i task attivati vengono gestiti tramite un pool di thread. I task registrano i loro risultati parziali all’interno di una ConcurrentHashMap.

Prima di terminare, il programma stampa su un apposito file di output il numero di occorrenze di ogni carattere. Il file di output contiene una riga per ciascun carattere ed è formattato come segue:

```
<carattere1>,<numero di occorrenze>  
<carattere2>,<numero di occorrenze>  
...  
<caratteren>,<numero di occorrenze>
```

ASSIGNMENT 4: CONTEGGIO OCCORRENZE

- esempio di file di output:

a,1281

b,315

c,261

d,302

...

- si allega un archivio compresso contenente file testuali per effettuare i test