# Sincronizzazione
## Costrutti di alto livello, Monitor, Atomic, Volatile Synchronized Collections

05/10/2021

Laboratorio di Programmazione di Rete

Federica Paganelli

# LOCK IMPLICITE O INTRINSECHE

- JAVA associa a ciascun oggetto (istanza di una classe) una lock implicita ed una coda associata a tale lock.
- unico meccanismo di sincronizzazione prima di JAVA 5
- acquisizione della lock mediante:
  - metodi synchronized

```
public synchronized void deposito(float soldi){
    saldo += soldi;
}
```

  - blocchi di codice sincronizzato

```
public void  somemethod ( )
  synchronized(object) {
            // a thread that is executing this code section
            // has acquired the object intrinsic lock
            // only a single thread may execute this
            // code section at any given time
   }
}
```

# LOCK IMPLICITE O INTRINSECHE

```
public synchronized void someMethod() {
  // Do work
}
```

- quando il metodo synchronized viene invocato il metodo tenta di acquisire la lock intrinseca associata all'oggetto su cui esso è invocato
  - riferito dalla parola chiave `this`
- se l'oggetto è bloccato (= lock acquisita da un altro thread in un blocco sincronizzato o in un altro metodo synchronized)
  - il thread viene sospeso nella coda associata all'oggetto fino a che il thread che detiene la lock la rilascia
- la lock viene rilasciata al ritorno del metodo

# LOCK IMPLICITE O INTRINSECHE

```
public synchronized void someMethod() {
// Do work
 }
```

- NB la lock è associata ad una istanza dell'oggetto, non alla classe
  - diverse invocazioni di metodi `synchronized`, sullo stesso oggetto, non sono soggette ad interleaving
  - metodi su **istanze diverse** della stessa classe possono essere eseguiti in modo concorrente!

# LOCK IMPLICITE O INTRINSECHE

- I **costruttori non devono essere dichiarati synchronized**
  - Il compilatore solleva una eccezione
  - idea alla base: solo il thread che crea l'oggetto deve avere accesso ad esso mentre l'oggetto viene creato

- **synchronized è riferito all'implementazione non alla signature del metodo**, quindi:
  - non ha senso specificare synchronized nelle interfacce
  - Se una sottoclasse sovrascrive (override) un metodo synchronized della superclasse, il metodo della sottoclasse deve essere reso synchronized (altrimenti non lo è)
  - Se la sottoclasse eredita un metodo `synchronized` (e non lo sovrascrive) lo eredita come sincronizzato

# SYNCHRONIZED STATEMENTS

- sincronizzare blocchi di codice, invece di interi metodi

```
synchronized (objref) {
        // Java code block
                        }
```

  si passa il riferimento ad un oggetto `objref`

- un thread acquisisce la lock sull'oggetto riferito da `objref`, quando entra nel blocco sincronizzato e la rilascia quando lascia il blocco sincronizzato.

- un thread alla volta esegue il blocco di codice su quell'oggetto.

# SYNCHRONIZED STATEMENTS: UTILIZZO

- "lock scope reduction":
- permettono di sincronizzare parti di un metodo, piuttosto che l'intero metodo -> sezioni critiche di dimensione minore all'interno di metodi

```
Object mutex = new Object();
...

public void someMethod(){
  nonCriticalSection();

  synchronized (mutex){
    criticalSection();
  }

  nonCriticalSection();
}
```

mutex in questo caso è un campo privato della classe

criticalSection() indica qualsiasi parte (metodo) della classe che deve essere eseguita come sezione critica

# SINCRONIZZAZIONI SU SINGOLI CAMPI

- sincronizzazioni indipendenti su campi diversi di una classe
- se sincronizzassi gli interi metodi, essi non potrebbero essere eseguiti in parallelo, anche se modificano variabili diverse

```
public class test {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
            synchronized(lock1) {c1++;}
     }
    public void inc2() {
            synchronized(lock2) {c2++;}
    }
```

# SYNCHRONIZED STATEMENTS

```
class Program {
    public synchronized void f() {
                .........
    }
}
```

equivale a:

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

# Sincronizzazione di metodi statici

- possibile sincronizzare anche metodi statici
- acquisiscono la lock intrinseca associata alla classe, invece che all'oggetto
- nel frammento di codice seguente i metodi vengono eseguiti in mutua esclusione

```
public class SomeClass {
  public static synchronized void methodOne() {
  // Do work
  }
 public void methodTwo() {
  synchronized (SomeClass.class) {
 // Do work
   }
 }
}
```

NB. MyClass.class e this sono riferimenti a oggetti differenti.

- `this` -è un riferimento ad una particolare istanza della classe
- `MyClass.class` - è un riferimento all'oggetto che descrive la classe MyClass

# IL MONITOR

- Monitor (Per Brinch Hansen, Hoare 1974): meccanismo linguistico ad alto livello per la sincronizzazione, classe di oggetti utilizzabili concorrentemente in modo safe:
  - incapsula una struttura condivisa e le operazioni su di essa
  - risorsa è un oggetto passivo: le sue operazioni vengono invocate dalle entità attive (threads)
- sincronizzazione sullo stato della risorsa garantita esplicitamente
  - mutua esclusione sulla struttura garantita dalla lock implicita associata alla risorsa - un solo thread per volta si trova "all'interno del monitor"
  - meccanismi per la sospensione/risveglio sullo stato dell'oggetto condiviso simili a variabili di condizione: wait/notify

# IL MONITOR

- un oggetto con un insieme di metodi `synchronized` che incapsula lo stato di una risorsa condivisa
- due code gestite in modo implicito:

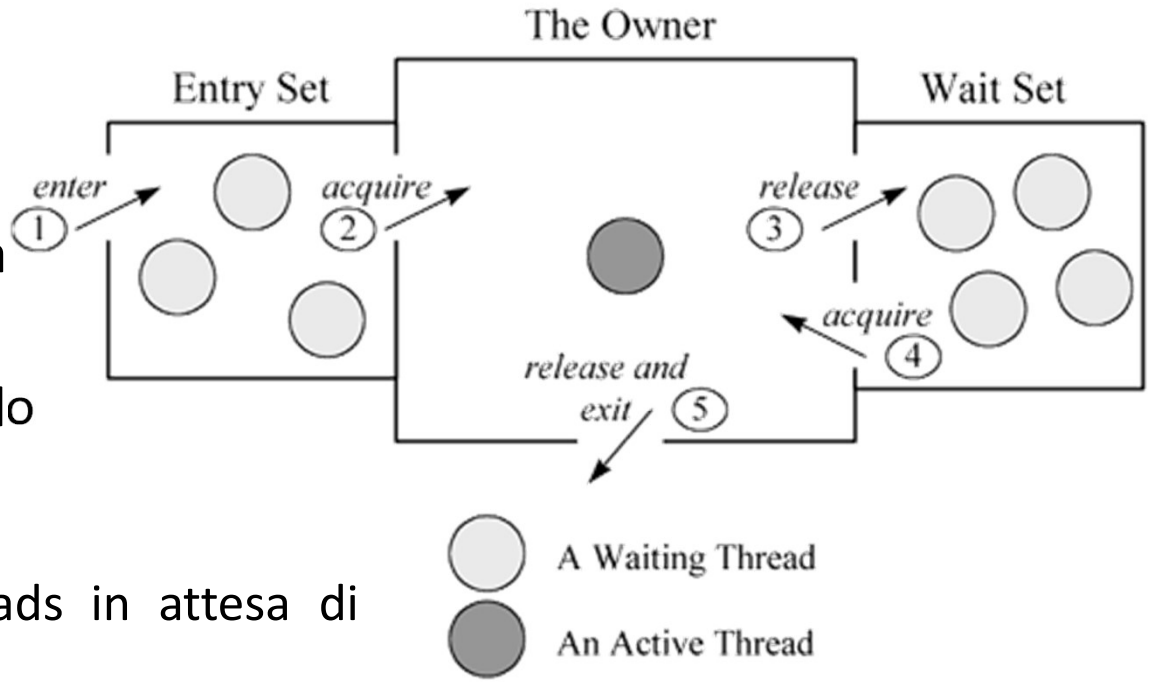



Figure 20-1. A Java monitor.

- **`Entry Set`**: contiene i threads in attesa di acquisire la lock.
- **`Wait Set`**: contiene i threads che hanno eseguito una `wait` e sono in attesa di una `notify`: inserzione/estrazione in questa coda in seguito ad invocazione esplicita di wait(), notify().

# I METODI WAIT/NOTIFY

- invocati su un oggetto:
  - appartengono alla classe **`Object`** (tutti le classi ereditano da `Object`,….)
  - per invocare questi metodi occorre aver acquisito precedentemente la lock sull'oggetto su cui sono invocati: devono quindi essere invocati all'interno di un metodo o di un blocco sincronizzato
  - se non compare riferimento esplicito all'oggetto, il riferimento implicito è `this`
- **`void wait()`** sospende il thread in attesa che sia verificata una condizione
- **`void wait (long timeout)`** sospende per al massimo timeout millisecondi
- **`void notify()`** notifica spedita ad un thread in attesa
- **`void notifyall()`** notifica spedita a tutti i threads in attesa

# WAIT E NOTIFY

- **`wait`**
  - rilascia la lock sull'oggetto (quella acquisita eseguendo il metodo sincronizzato) prima di sospendere il thread
    - a differenza di **`sleep( )`** e **`yield()`**
  - in seguito ad una notifica, può riacquisire la lock

- **`notify( )`**
  - risveglia uno dei thread nella coda di attesa di quell'oggetto
  - se viene invocato quando non vi è alcun thread sospeso la notifica viene «persa».
- **`notifyAll( )`**
  - risveglia tutti i threads in attesa
  - i thread risvegliati competono per l'acquisizione della lock
  - i thread verranno eseguiti uno alla volta, quando riusciranno a riacquisire la lock sull'oggetto

# WAIT E NOTIFY

- i metodi `wait`, `notify`, `notifyall` devono essere invocati all'interno di un metodo o blocco synchronized, altrimenti viene sollevata l'eccezione **`IllegalMonitorException( )`**

- il metodo `wait` permette di attendere un cambiamento su una condizione:
  - "fuori dal monitor"
  - in modo passivo evitando il controllo ripetuto di una condizione (*polling*)

- poichè esiste una unica coda implicita in cui vengono accodati i thread in attesa, non è possibile distinguere thread in attesa di condizioni diverse.
  - ogni thread deve ricontrollare se la condizione è verificata, dopo il suo risveglio.
  - differenza con le variabili di condizione.

# WAIT/NOTIFY: 'REGOLA D'ORO' PER L'UTILIZZO

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

- testare la condizione all'interno di un ciclo
  - poiché la coda di attesa è unica per tutte le condizioni, il thread potrebbe essere stato risvegliato in seguito al verificarsi di un'altra condizione
  - la condizione su cui il thread T è in attesa si è verificata però un altro thread l'ha resa di nuovo non valida dopo che T è stato risvegliato

# Wait e notify

- Struttura del codice:

```
synchronized(o){
    while(!condition){
       o.wait();
    }

    //make changes to o and other data
    o.notify();
}
```

Acquisisce il monitor

Rilascia il monitor

Acquisisce il monitor

Rilascia il monitor

# WAIT/NOTIFY E BLOCCHI SINCRONIZZATI

- attendere il verificarsi di una condizione su un oggetto condiviso

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

- modificare una condizione

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

# notify vs notifyALL

- differenze tra `notify()` e `notifyall()`
  - **`notify( )`** riattiva uno dei threads nella coda associata all'oggetto su cui si invoca la funzione.
  - **`notifyAll( )`** riattiva tutti i thread in attesa sull' oggetto, l'oggetto è quello su cui è stata invocata: i thread vengono messi in stato di pronto e competono, successivamente, per l'acquisizione della lock.

# MONITOR E LOCK ESPLICITE: CONFRONTI

- Vantaggi  delle Lock implicite:
  - imposta una disciplina di programmazione  per evitare errori dovuti alla complessità del programma concorrente: deadlocks, mancato rilascio di lock,....
  - definere costrutti “strutturati” per la gestione delle concorrenza
  - maggior robustezza
  - svantaggi: minore flessibilità rispetto a lock esplicite

- Vantaggi delle Lock esplicite
  - un numero maggiore di funzioni disponibili, maggiore  flessibilità
  - `tryLock()` consente di non sospendere il thread se un altro thread è in possesso della lock, restituendo un valore booleano
  - shared locks: multiple reader single writer
  - Condition Variables
  - migliori performance

# READERS/WRITERS CON MONITOR

- Problema dei thread lettori/scrittori.
  - Lettori: escludono gli scrittori, ma non gli altri lettori
  - Scrittori: escludono sia i lettori che gli scrittori

- Astrazione del problema dell'accesso ad una base di dati
  - un insieme di threads possono leggere dati in modo concorrente
  - per assicurare la consistenza dei dati, le scritture devono essere eseguite in mutua esclusione

- Analizziamo la soluzione con Monitor:
  - senza lock esplicite
  - senza ReadWriteLock
  - senza condition variables

# READERS/WRITERS CON MONITOR

```
public class ReadersWriters {
public static void main(String args[]) {
        RWMonitor RWM = new RWMonitor();
        for (int i=1; i<10; i++)
           {Reader r = new Reader(RWM,i);
            Writer w = new Writer(RWM,i);
            r.start();
            w.start();
            }
      }
   }
```

# READERS/WRITERS : WRITER STARVATION

```
public class Reader extends Thread {
  RWMonitor RWM;
  int i;
  public Reader (RWMonitor RWM, int i)
     { this.RWM=RWM; this.i=i;}
  public void run() {
     while (true) {
         RWM.startRead();
       try{
         Thread.sleep((int)Math.random() * 1000);
         System.out.println("Lettore"+i+"sta
                        leggendo");
       }
       catch (Exception e){};
       RWM.endRead();
     }
  }
}
```

# READERS/WRITERS : WRITER STARVATION

```
public class Writer extends Thread {
  RWMonitor RWM; int i;
  public Writer (RWMonitor RWM, int i) {
    this.RWM=RWM; this.i=i;
  }
  public void run() {
    while (true) {
      RWM.startWrite();
      try{
        Thread.sleep((int)Math.random() * 1000);
        System.out.println("Scittore"+i+"sta scrivendo");
      }
      catch (Exception e){};
      RWM.endWrite();
    }
   }
}
```

# READERS/WRITERS : WRITER STARVATION

```
class RWMonitor {
  int readers = 0;
  boolean writing = false;

  synchronized void startRead() {
    while (writing)
      try {
        wait();
      }
      catch (InterruptedException e) {}
    readers = readers + 1;
  }
  synchronized void endRead() {
    readers = readers - 1;
    if (readers == 0) notifyAll();
  }
…
```

# READERS/WRITERS : WRITER STARVATION

```
…
synchronized void startWrite() {
    while (writing || (readers != 0))
      try {
         wait();
      } catch (InterruptedException e) {}
    writing = true;
  }
  synchronized void endWrite() {
    writing = false;
    notifyAll();
  }
}
```

# READERS/WRITERS : WRITER STARVATION

- un lettore può accedere alla risorsa se ci sono altri lettori, i lettori possono accedere continuamente alla risorsa e non dare la possibilità di accesso agli scrittori
- se uno scrittore esce esegue una `notifyall()` che sveglia sia i lettori che gli scrittori: comportamento fair se è fair la strategia di schedulazione di JAVA.

  - *Lettore2 sta leggendo*
  - *Lettore9 sta leggendo*
  - *Lettore1 sta leggendo*
  - *Lettore1 sta leggendo*
  - *Lettore4 sta leggendo*
  - *Lettore7 sta leggendo*
  - *Lettore6 sta leggendo*
  - *Lettore0 sta leggendo*
  - *Lettore3 sta leggendo*

# Volatile & Atomic Variables

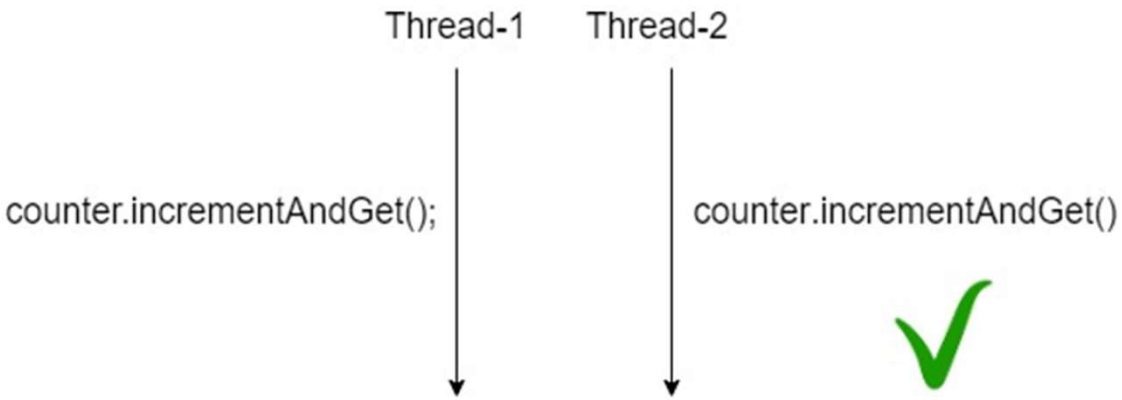
# Atomic variables

- Java Concurrency include il package `java.util.concurrent.atomic`
  - `AtomicBoolean`
  - `AtomicInteger`
  - `AtomicLong`
  - …
- Incapsulano variabili di tipo primitivo e garantiscono l'atomicità delle operazioni
- lock-free atomic operations
  - garantiscono l'atomicità delle operazioni per tipi di dato primitivi (integer, long,...) ed array senza usare sincronizzazioni esplicite o lock

# ATOMIC VARIABLES

- `AtomicInteger value = new AtomicInteger(1);`
- operazioni atomiche  senza usare sincronizzazioni esplicite o lock
  - `incrementAndGet()`: atomically increments by one
  - `decrementAndGet()`: atomically decrements by one
  - `compareAndSet(int expectedValue, int newValue)`
- altre classi
  - `AtomicLong`
  - `AtomicBoolean`

…



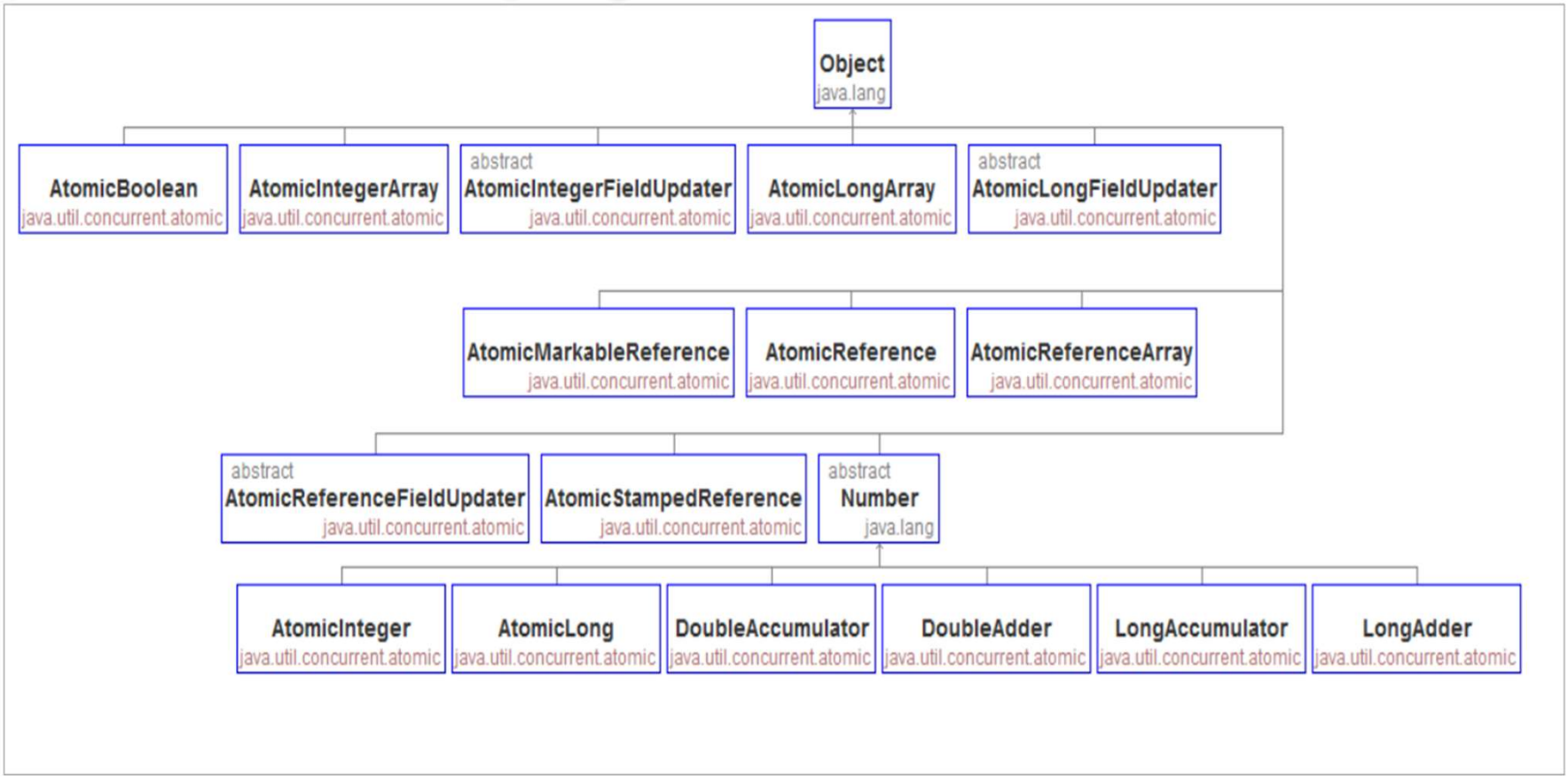
# Atomic variables - example

```
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++){
         executor.submit(new  CounterRunnable(atomicInt))
     }
     executor.shutdown();
 }
```

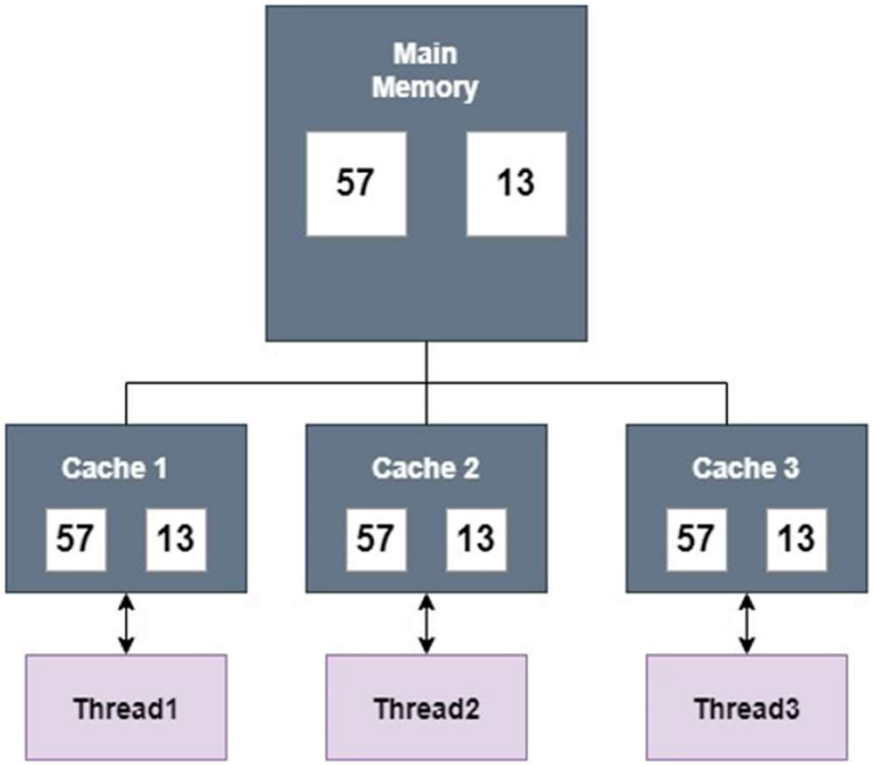
# Atomic Variables - example

```
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



# IL PROBLEMA DELLA VISIBILITA'



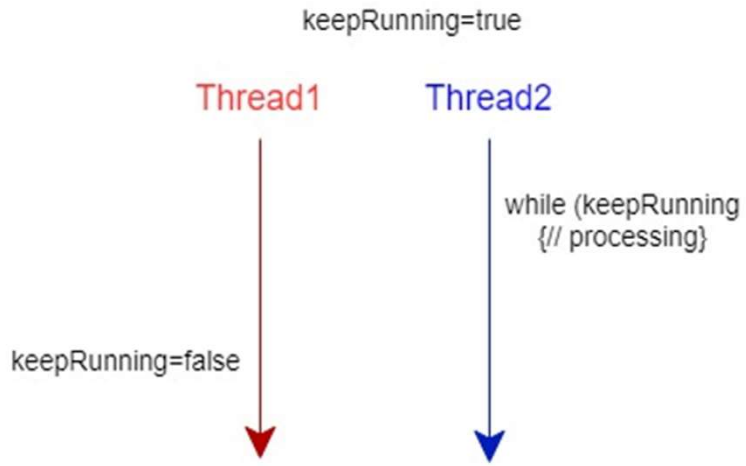

Architettura di riferimento

- ciascun thread può copiare le variabili dalla memoria principale in una cache della CPU per motivi di prestazioni.
- Ciò significa che ogni thread può copiare le variabili nella cache della CPU di diverse CPU.
- problema di "visibilità". Gli aggiornamenti di un thread non sono visibili agli altri thread.

# IL PROBLEMA DELLA VISIBILITA'

```
class Test extends Thread {
    boolean keepRunning = true;
    public void run() {
      while (keepRunning) {
       }
      System.out.println("Thread terminate

    public static void main(String[] args) throws InterruptedException {
        Test t = new Test();
        t.start();
        Thread.sleep(1000);
        t.keepRunning = false;
        System.out.println("keepRunning set to false.");
        }
      }
```




il programma non termina!

# IL PROBLEMA DELLA VISIBILITA': soluzione

- quando il Thread-1 aggiorna il flag KeepRunning, la modifica può non essere letta dal Thread-2: visibility problem
- modifichiamo la dichiarazione della variabile con la keyword volatile

```
volatile boolean keepRunning = true;
```

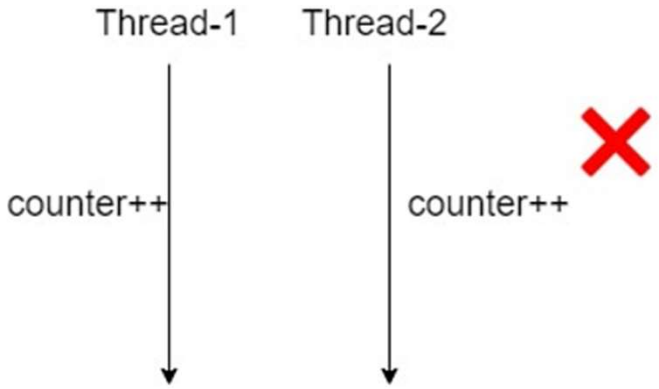
- From Java Specifications:
- *A field may be declared volatile, in which case the Java Memory Model ensures that all threads see a consistent value for the variable*.
- Nota bene (sempre dalle Java Specifications)
- *The memory model describes possible behaviors of a program. An implementation is free to produce any code it likes, as long as all resulting executions of a program produce a result that can be predicted by the memory model.*

# IL PROBLEMA DELLA VISIBILITA': SOLUZIONE

- Il campo volatile è necessario per assicurarsi che più thread vedano sempre il valore più recente, anche quando il sistema di cache o le ottimizzazioni del compilatore sono al lavoro.
- La lettura da una variabile volatile restituisce sempre l'ultimo valore scritto da questa variabile.
- E poiché il processore sincronizza l'intera cache, vediamo gli ultimi valori scritti di tutte le variabili

- Quando usare il modificatore volatile?
- Quando le operazioni di scrittura sulla variabile non dipendono dal suo valore corrente, oppure se è possibile assicurarsi che un solo un singolo thread aggiorni il valore
  - Es. flag, letture di sensori, ecc

# SINCRONIZZAZIONE SU SINGOLE VARIABILI




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

- ma... l'incremento di una variabile volatile non è atomico
- se più thread provano ad incrementare una variabile concorrentemente, un aggiornamento può andare perduto (anche se la variabile è volatile)
- Soluzioni:
  - `Atomic` variables
  - Lock o blocchi synchronized

# Synchronized Collections

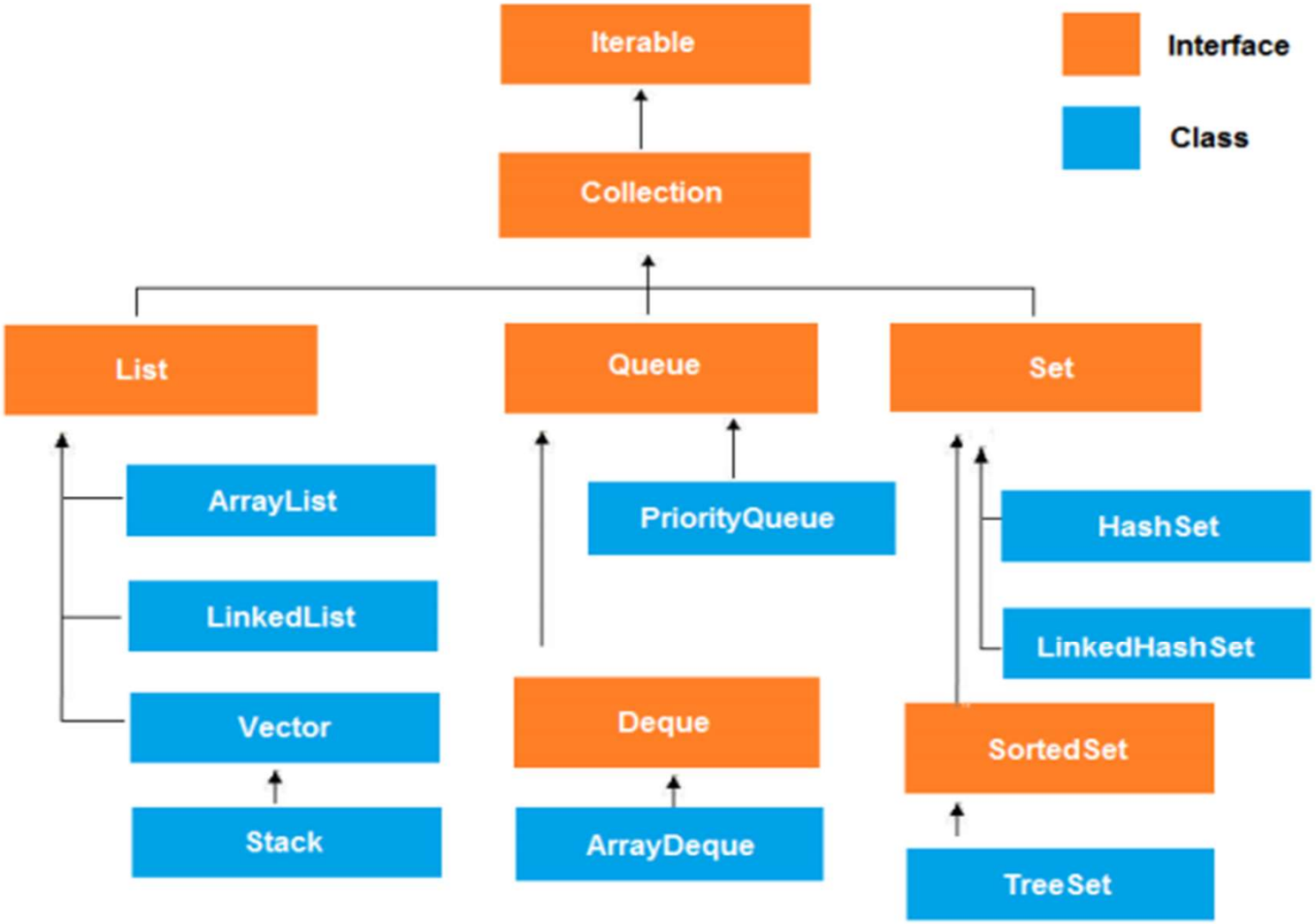
# JAVA COLLECTION FRAMEWORK: RIPASSO

- 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`**

- introdotte nel corso di Programmazione 2. Per il progetto rivedere le implementazioni delle collezioni più importanti con lo scopo di utilizzare nel progetto le strutture dati più adeguate

- vedremo:
  - collezioni ed iteratori: ripasso
  - **`synchronized collections`**
  - **`concurrent collections`** (lezione successiva)

# L'INTERFACCIA COLLECTION: RIPASSO



# JAVA COLLECTION FRAMEWORK: RIPASSO

- una ulteriore interfaccia: **`Map`**
  - **`HashMap`** (implementazione di Map) non è un'implementazione di **Collection**, ma è comunque una struttura dati molto usata dal **Java Collection Framework**
  - 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 wrapper di sincronizzazione ad una collezione

# JAVA COLLECTION FRAMEWORK: RIPASSO

- Iteratori:
  - oggetto di supporto usato per accedere agli elementi di una collezione, uno alla volta e in sequenza
  - associato ad un oggetto collezione (lavora su uno specifico insieme o lista, o map)
  - deve conoscere (e poter accedere) alla rappresentazione interna della classe che implementa la collezione (tabella hash, albero, array, lista puntata, ecc...)
- L'interfaccia **`Collection`** contiene il metodo **`iterator()`** che restituisce un iteratore per una collezione
  - le diverse implementazioni di **Collection** implementano il metodo **iterator()** restituendo un oggetto iteratore specifico per quel tipo di collezione
  - l'interfaccia **Iterator** prevede tutti i metodi necessari per usare un iteratore, senza conoscere alcun dettaglio implementativo

# JAVA COLLECTION FRAMEWORK: RIPASSO

- 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, l'iteratore è necessario crearne uno nuovo)
- schema generale per l'uso di un iteratore

```
// collezione di oggetti di tipo T che vogliamo scandire
Collection<T> c = ....
...

// iteratore specifico per la collezione c
Iterator<T> it = c.iterator()

// finche'non abbiamo raggiunto l'ultimo elemento
while (it.hasNext()) {
  // ottieni un riferiento all'oggeto corrente, ed avanza
  T e = it.next();
  ....      // usa l'oggetto corrente (anche rimuovendolo)
}
```

# JAVA COLLECTION FRAMEWORK: RIPASSO

```
HashSet<Integer> set = new HashSet<Integer>();

......

Iterator<Integer> it = set.iterator()

while (it.hasNext()) {
  Integer i = it.next();
  if (i % 2 == 0)
    it.remove();
  else
    System.out.println(i);
}
```

- ciclo for each: `for (String s : c)`
  - corrisponde a creare implicitamente un iteratore per la collezione v
  - Non permette di modificare la collezione (e.g. eliminare o modificare elementi

# JAVA COLLECTIONS E MULTITHREADING

- il supporto per gestire un accesso concorrente corretto agli elementi della collezione (thread safeness ) varia da classe a classe

- in generale, si possono distinguere tre tipi di collezioni
  - collezioni che non offrono alcun supporto per il multithreading
  - synchronized collections
  - concurrent collections (introdotte in **`java.util.concurrent`**)

# JAVA COLLECTIONS E SINCRONIZZAZIONE

- `Vector`:
  - contenitore elastico, “estensibile” ed “accorciabile”, non generico
  - una collezione thread safe conservative locking
  - Performance penalty

- `ArrayList`
  - come Vector, un vettore di dimensione variabile,
  - introdotto in Java1.2, prima di JDK5, può contenere solo elementi di tipo Object, dopo parametrico (generics) rispetto al tipo degli oggetti contenuti
    - Es. `List<String> v = new ArrayList<String>();`
  - gli elementi possono essere acceduti in modo diretto tramite l'indice
  - thread safety non fornita di default
    - nessuna sincronizzazione
    - maggior efficienza

# 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
```

# UNSYNCHRONIZED COLLECTIONS

- **Unsynchronized Collections: ArrayList**
  - un loro uso non controllato, in un programma multithreaded, può portare a risultati scorretti.

- consideriamo il seguente codice:

```
static List<String> testList = new ArrayList<String>();
testList.add("LaboratorioReti");
```

- l'implementazione della **add** non è atomica:
  - determina quanti elementi ci sono nella lista
  - determina il punto esatto in cui il nuovo elemento deve essere aggiunto
  - incrementa il numero di elementi della lista
  - esecuzioni concorrenti della **add** possono portare a interleaving scorretti

# UNSYNCHRONIZED COLLECTIONS

- Esempio di interleaving scorretto per la funzione `add:`
  - un thread A memorizza il numero di elementi della lista e poi viene deschedulato
  - un secondo thread B aggiunge un elemento nella lista ed incrementa il numero di elementi
  - A, quando viene di nuovo schedulato:
    - ha memorizzato un numero precedente (errato) di elementi della lista
    - lo incrementa di uno e sovrascrive il numero totale di elementi, che alla fine risulta errato

- in un ambiente concorrente, il programmatore deve coordinare esplicitamente i thread, mediante:
  - utilizzo di "wrapper" che aggiungono funzionalità di sincronizzazione alla collezione
  - meccanismi espliciti di tipo `lock` o `synchronized`

# Synchronized collections

# SYNCHRONIZED COLLECTIONS

- La classe **`Collections`** contiene metodi statici per l'elaborazione delle collezioni
- **`Factory methods`** per creare versioni sincronizzate di `lists/sets/map`
  - input: una collezione
  - output: la stessa collezione in cui le operazioni sono sincronizzate

```
List<String> synchList=
    Collections.synchronizedList(new ArrayList<String>())

    synchList.add("Laboratorio Reti");
```

- I metodi della collection risultante è protetta da lock, quindi thread-safe
- PS. **Nessun thread deve accedere all'oggetto originale (costruzione come nell'esempio in alto previene questo rischio**).
  - synchronizedList() produce un nuovo oggetto List che memorizza l'argomento in un campo privato
  - lock sull'intera collezione: degradazione di performance

# SYNCHRONIZED COLLECTIONS

```
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 start2 =System.nanoTime();
  addElements(Collections.synchronizedList(new ArrayList<Integer>()));
  final long end2=System.nanoTime();
  System.out.println("ArrayList time "+(start2-start1));
  System.out.println("SynchronizedArrayList time "+(end2-start2));}}
```

ArrayList time 50677689

SynchronizedArrayList time 62055651

# SYNCHRONIZED COLLECTIONS

- La thread safety garantisce che le invocazioni delle singole operazioni della collezione siano thread-safe

- Ma se si vogliono definire funzioni che coinvolgono più di una operazione base?

```
public static Object getLast (List<Object> l) {
                int lastIndex = l.size() - 1;
                return (l.get(lastIndex));
}
```

- può generare una **IndexOutOfBoundsException**!
  - tutti i thread calcolano, in modo concorrente, il valore di lastIndex
  - il primo thread rimuove l'elemento in quella posizione
  - il secondo thread prova a rimuovere, ma la posizione non risulta più valida

# SYNCHRONIZED COLLECTIONS

- un altro esempio:

```
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 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 si dicono **conditionally thread-safe** : le operazioni individuali sulle collezioni sono safe, ma funzioni composte da più di una operazione singola possono richiedere meccaninismi esterni di sincronizzazione.

# SYNCHRONIZED COLLECTIONS

- richiesta sincronizzazione esplicita da parte del programmatore

- per rendere atomica una operazione composta da più di una operazione individuale

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

- tipico esempio di utilizzo di blocchi sincronizzati
- notare che 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 da lock rientranti

# ITERATORI E CONCORRENZA

- anche l'iterazione su una collezione può essere vista come una operazione composta da tante operazioni elementari
- uno scenario che si verifica spesso:
  - un thread sta usando un iteratore su una collezione
  - un altro thread modifica la stessa collezione
  - al momento del reperimento dell'elemento successivo, l'iteratore solleva una **ConcurrentModificationException**
- Soluzione: sincronizzare l'intera struttura

```
Collection<Type> c =
 Collections.synchronizedCollection(myCollection);
   synchronized(c) {
        for (Type e : c)
               foo(e);}
```

- Mantiene consistente lo stato della collezione, ma riduce la concorrenza

# ITERATORI E CONCORRENZA

```
import java.util.*;
public class UseHashMap {
   private Map<String, Integer> scores = new HashMap<String,
                                                    Integer>();
   public void printScores(){
       for (Map.Entry<String,Integer> entry : scores.entrySet())
           {System.out.println(String.format("Score for name %s
is %d", entry.getKey(), entry.getValue()));
            try{Thread.sleep(1000);}
            catch (Exception ex){   }
           }
   }
   public void addScore(String name, int score)
       {scores.put(name, score);}
}
```

# ITERATORI E CONCORRENZA

```
public class PrintTask implements Runnable {
    UseHashMap u;
    public PrintTask(UseHashMap u) {
        this.u=u;
    }
    public void run() {
        u.printScores();
    }
}
```

# ITERATORI E CONCORRENZA

```
public class MainClass {
public static void main(String[] args) {
    UseHashMap useHashMap = new UseHashMap();
    Thread t= new Thread(new PrintTask(useHashMap));
    t.start();
    useHashMap.addScore("Sara",14);
    useHashMap.addScore("Jhon", 12);
    try {
        Thread.sleep(1000);}
    catch (Exception ex){}
    useHashMap.addScore("Bill",13);
    System.out.println("Added bill");
    }
}
```

# ITERATORI E CONCORRENZA

```
Score for name Sara is 14
Added bill
Exception in thread "Thread-0" java.util.ConcurrentModificationException
at java.util.HashMap$HashIterator.nextNode(Unknown Source)
at java.util.HashMap$EntryIterator.next(Unknown Source)
at java.util.HashMap$EntryIterator.next(Unknown Source)
at UseHashMap.printScores(UseHashMap.java:7)
at PrintThread.run(PrintThread.java:7)
at java.lang.Thread.run(Unknown Source)
```

- Attenzione:
- l'eccezione non viene generata se non c'è interleaving tra le add e la scansione effetuata dall'iteratore.
- affinchè questo si verifichi, occorre inserire le `sleep()` nei punti opportuni
- Sincronizzando le collezioni, si evita l'eccezione

# ITERATORI E CONCORRENZA

```
import java.util.*;
public class UseHashMap {
   private Map<String, Integer>scores = new HashMap<String, Integer>();
   public void printScores() {
     synchronized(scores){
        for (Map.Entry<String,Integer> entry : scores.entrySet()) {
            System.out.println(String.format("Score for name %s
            is %d", entry.getKey(), entry.getValue()));
            try {
                Thread.sleep(1000);} catch (Exception ex){}
           }
        }
     }
   public void addScore(String name, int score) {
     synchronized(scores) {
          scores.put(name, score);}
}}}
```