Laboratorio di Reti
Lezione 4
Lock intrinseche
Concurrent Collections

08/10/2020

Laura Ricci

LOCK INTRINSECHE: BLOCCHI SINCRONIZZATI

- ciascun oggetto JAVA (istanza di una classe) ha associato
  - una lock implicita
  - una coda associata alla lock.
- è stato l'unico meccanismo di sincronizzazione disponibile prima di JAVA 5
- blocco 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
       }}
```

- un thread
  - acquisisce la lock su object, quando entra nel blocco sincronizzato
  - la rilascia quando termina il blocco sincronizzato.

LOCK INTRINSECHE: METODI SINCRONIZZATI

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

metodo `synchronized` : quando viene invocato

- tenta di acquisire la lock intrinseca associata alla istanza dell'oggetto su cui esso è invocato
  - istanza riferita dalla parola chiave `this`
  - se l'oggetto è bloccato (= lock acquisita da un blocco sincronizzato o da un altro metodo `synchronized`)
    - il thread viene sospeso nella coda associata all'oggetto fino a che il thread che detine la lock la rilascia
- la lock viene rilasciata al ritorno del metodo
  - normale
  - eccezionale, ad esempio con una uncaught exception.

LOCK INTRINSECHE: METODI SINCRONIZZATI

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

equivale a:

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

METODI SYNCHRONIZED: MISCELLANEA

- i costruttori non devono essere dichiarati `synchronized`
  - il compilatore solleva una eccezione
  - solo il thread che crea l'oggetto deve avere accesso ad esso mentre l'oggetto viene creato
- 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, non lo è per default
- la lock è associata ad una istanza dell'oggetto, non alla classe, metodi su oggetti che sono istanze diverse della stessa classe possono essere eseguiti in modo concorrente!

LOCK SCOPE REDUCTION

- sincronizzare parti di un metodo, piuttosto che l'intero metodo
- sezioni critiche di dimensione minore all'interno di metodi
- utizzano oggetti "di pura sincronizzazione"
  - oggetto mutex: realizzato acquisendo la `lock()` intrinseca di un generico Oggetto

```
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

LOCK SCOPE REDUCTION

```
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++;}
         }   }
```

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

SINCRONIZZAZIONE DI METODI STATICI

- possibile sincronizzare anche metodi statici
- acquisiscono la lock intrinseca associata alla classe, invece che all'istanza di un oggetto
- nel fragmento 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
} } }
```

IL MONITOR

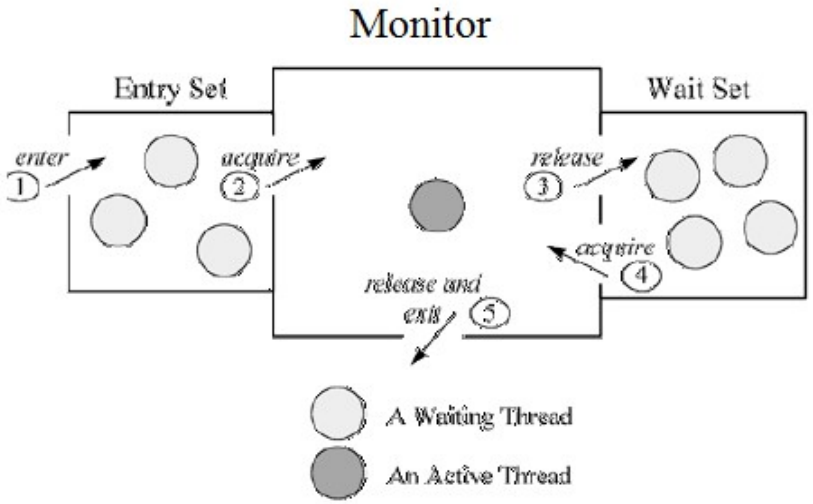
Monitor (`Per Brinch Hansen, Hoare 1974`): meccanismo linguistico ad alto livello per la sincronizzazione: classe di oggetti utilizzabili concorrentemente in modo safe:

- incapsula una struttura condivisa e le operazioni su di essa
- risorsa è un oggetto passivo: le sue operazioni vengono invocate dalle entità attive (threads)
- mutua esclusione sulla struttura garantita la lock implicita associta alla risorsa
  - un solo thread per volta si trova "all'interno del monitor"
- sincronizzazzione sullo stato della risorsa garantita esplicitamente
  - 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:
  - Entry Set: contiene i threads in attesa di acquisire la lock.
    inserzione/estrazione associate ad invocazione/terminazione del metodo
  - 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

- void wait( ) sospende il thread in attesa che sia verificata una condizione
  - permette di attendere "fuori dal monitor" un cambiamento su una condizione
  - attesa passiva evita il controllo ripetuto di una condizione (polling)
- void wait (long timeout)
  - sospende per al massimo timeout millisecondi
- void notify( ) notifica di una condizione spedita ad un thread in attesa
- void notifyall( ) notifica di una condizione spedita a tutti i threads in attesa

- sono invocati su un oggetto, quindi appartengono alla classe `Object`
- per invocarli, occorre aver acquisito precedentemente la lock
  - invocati all'interno di un metodo o di un blocco sincronizzato
    - altrimenti viene sollevate l'eccezione IllegalMonitorException( )
- se non compare riferimento esplicito all'oggetto, il riferimento implicito è `this`

WAIT E NOTIFY

- `wait ()`
  - rilascia la lock sull'oggetto 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 sulla lock di quell'oggetto
  - se viene invocato quando non vi è alcun thread sospeso la notifica viene persa.

- `notifyAll( )`
  - risveglia tutti i threads in attesa, che passano in stato di pronto
    - i thread risvegliati competono per l'acquisizione della lock
    - i thread verranno eseguiti uno alla volta, quando riusciranno a riacquisire la lock sull'oggetto

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, non usiamo
  - lock esplicite
  - ReadWriteLock
  - 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 si 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

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

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

testare sempre la condizione all'interno di un ciclo

- poichè 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 una altro thread la ha resa di nuovo invalida, dopo che T è stato risvegliato
- il ciclo può essere evitato solo in casi particolari

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()}
```

MONITOR E LOCK ESPLICITE: CONFRONTI

- lock implicite
  - vantaggi:
    - costrutti strutturati. Evitare errori dovuti alla complessità del programma concorrente: deadlocks, mancato rilascio di lock,....
    - maggior manutenibilità del software
  - svantaggi: poco flessibili
- lock esplicite
  - vantaggi: 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
    - migliori performance
  - svantaggi: codice poco leggibile, se usate in modo non strutturato

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:
  - `java.util.concurrent`
    - `Executor, concurrent collections, semaphores,...`
  - `java.util.concurrent.atomic`
    - `AtomicBoolean, AtomicInteger,...`
  - `java.util.concurrent.locks`
    - `Condition`
    - `Lock`
    - `ReadWriteLock`

UTILITY E LORO FUNZIONI

- `Executors, Thread pools and Futures`
  - execution frameworks per asynchronous tasking
- `Concurrent Collections:`
  - queues, blocking queues, concurrent hash map, …
  - strutture dati disegnate per ambienti concorrenti
- `Locks and Conditions`
  - Meccansimi di sincronizzazione più flessibili
  - Read/write locks
- `Synchronizers: Semaphore, Latch, Barrier, Exchanger`
  - tools per il coordinamento dei thread
- `Atomic variables`
  - atomicità di variabili di tipo semplice

LOCK FREE ATOMIC OPERATIONS

- Java Concurrency include il package java.util.concurrent.atomic
  - `AtomicBoolean`
  - `AtomicInteger`
  - `AtomicLong`

    ...
- classi che incapsulano variabili di tipo primitivo e garantiscono l'atomicità di operazioni, come ad esempio:
  - `incrementAndGet()`: atomically increments by one the current value.
  - `decrementAndGet()`: atomically decrements by one the current value.
- lock-free atomic operations
- garantiscono l'atomicità delle operazioni per tipi di dato primitivi (integer, long,...) ed array senza usare sincronizzazioni esplicite o lock

LOCK-FREE ATOMIC OPERATIONS

- operazioni per tipi di dato primitivi (integer, long,...) ed array che non richiedono l'utilizzo di sincronizzazioni esplicite o lock

```
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();
   CounterRunnable runnableTask = new  CounterRunnable(atomicInt);
   for(int i = 0; i < 10; i++){
        executor.submit(runnableTask);
    }
    executor.shutdown();
}
```

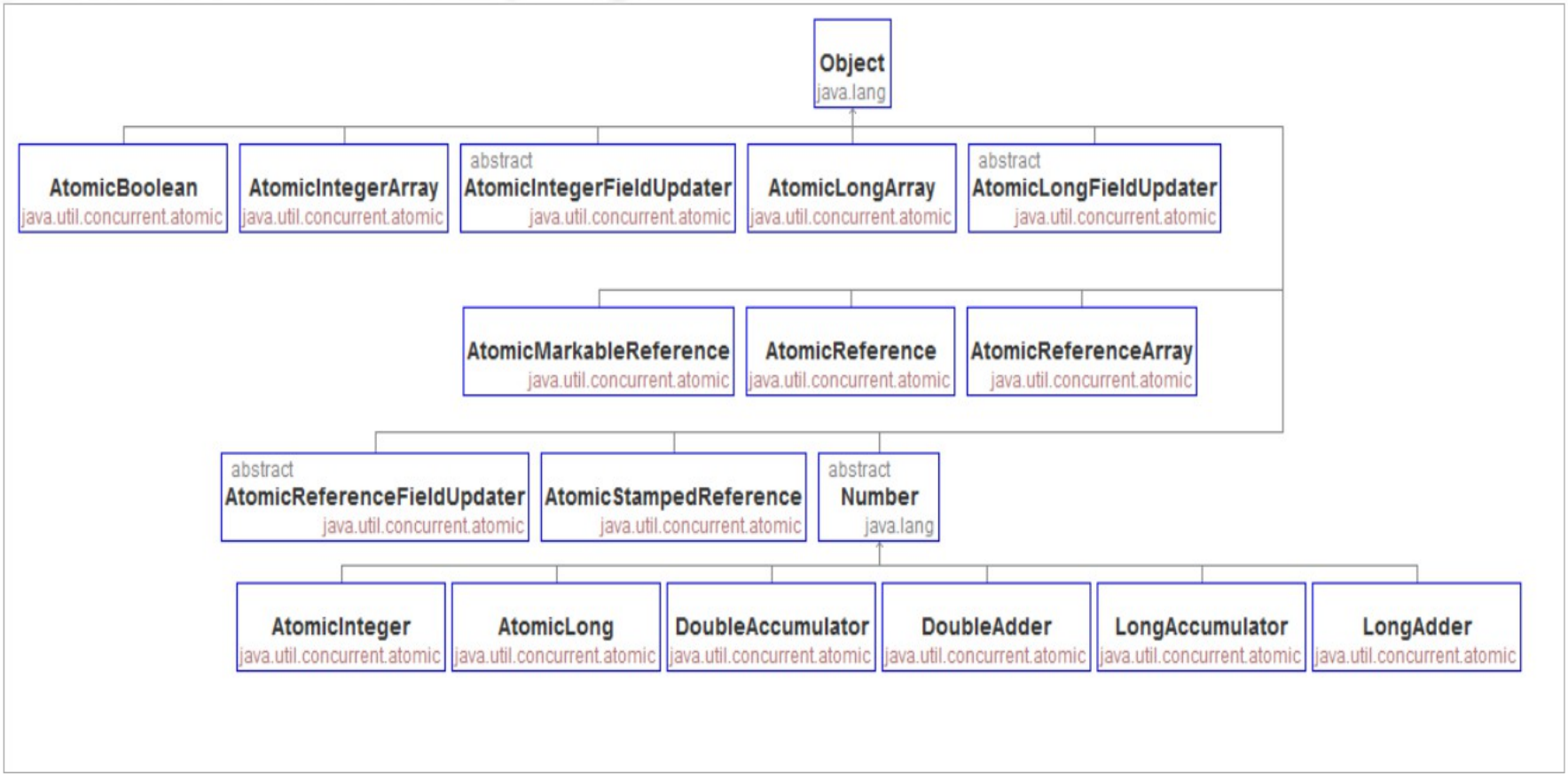
LOCK-FREE ATOMIC OPERATIONS

- garantiscono l'atomicità delle operazioni per tipi di dato primitivi (integer, long,...) ed array senza usare sincronizzazioni esplicite o lock
- lock-free atomic operations

```
class CounterRunnable implements Runnable{
      AtomicInteger atomicInt;
      CounterRunnable(AtomicInteger atomicInt){
                this.atomicInt = atomicInt;
       }
@Override
public void run() {
      System.out.println("Counter- " + atomicInt.incrementAndGet());
      }
}
```

- `incrementAndGet()` operazione atomica che incrementa il valore del contatore

JAVA.UTIL.CONCURRENT.ATOMIC



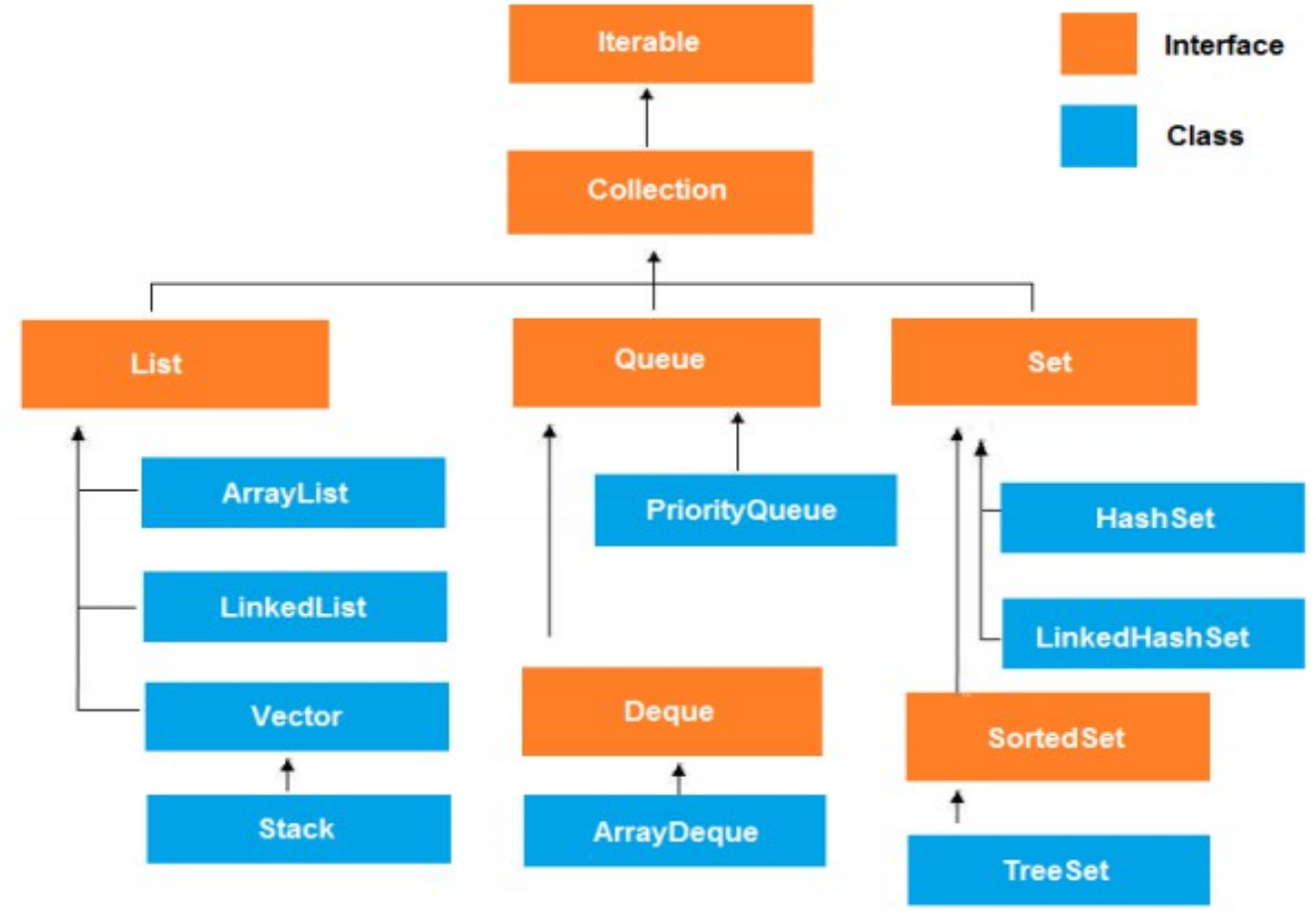
JAVA COLLECTION FRAMEWORK: RIPASSO

- 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.
  - rivedere le implementazioni delle collezioni più importanti con lo scopo di utilizzare nel progetto le strutture dati più adeguate

- in questa lezione:
  - collezioni ed iteratori: ripasso
  - `syncronized collections`
  - `concurrent collections`

DISTRICARSI NELLA GIUNGLA DELLE CLASSI



DISTRICARSI NELLA GIUNGLA DELLE CLASSI

- un'ulteriore 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 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
  - thread safe conservative locking
    - performance penalty

- JAVA 1.2: `ArrayList`
  - un vettore di dimensione variabile,
  - prima di JDK5, può contenere solo elementi di tipo `Object`, dopo parametrico (generic) rispetto al tipo degli oggetti contenuti
  - 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

- ArrayList: non thread safe
  - un loro uso non controllato, in un programma multithreaded, può portare a risultati scorretti.

```
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

IL METODO SynchronizedCollection( )

```
Collection <Integer> synchCollection=
        Collections.synchronizedCollection(new ArrayList<Integer>());
synchCollection.addAll(Arrays.asList(1,2,3,4,5,6));
System.out.println(synchCollection);
```

- il metodo restituisce una collezione thread-safe a partire da una collezione
  - garantita con lock intrinseche sull'intera collezione
  - ogni metodo sincronizzato sulla stessa lock
  - implementata mediante metodi wrapper che contengono costrutti di sincronizzazione
  - o qualcosa del genere
  - degradazione di performance: un singolo thread alla volta sulla intera collezione
- diversi metodi che implementano un "wrapped" con codice di sincronizzazione
  - **`synchronizedList, synchronizedMap, synchronizedSet,....`**

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 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
```

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`!
  - due 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

```
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`: conditionally thread-safe.
  - le operazioni individuali sulle collezioni sono safe, ma funzioni composte da più di una operazione singola possono risultarlo.

SYNCHRONIZED COLLECTIONS

- per rendere atomica una operazione composta da più di una operazione individuale richiesta una sincronizzazione esplicita da parte del programmatore

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

- tipico esempio di utilizzo di blocchi sincronizzati

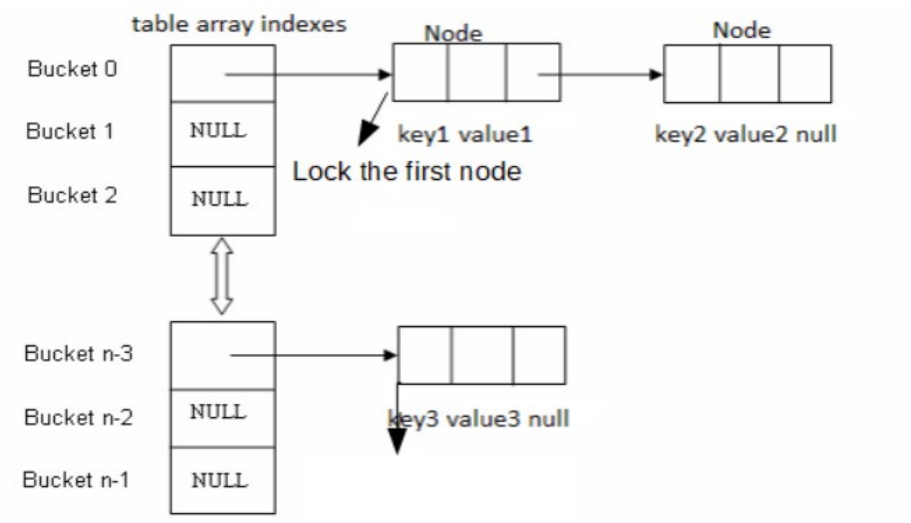
- il thread che esegue l'operazione composta acquisisce la lock sulla struttura synchList più di una volta:
  - quando esegue il blocco sincronizzato
  - quando esegue i metodi della collezione

ma...il comportamento corretto è garantito da lock rientranti

CONCURRENT COLLECTIONS

- introdotte a partire da JAVA5 in `java.util.concurrent`
- alternative thread-safe rispetto a `Collection`
- esempio:
  - `ConcurrentHashMap` versione thread safe di `HashMap`
- approccio alternativo rispetto a `Collections.synchronizedCollections()`
  - vantaggio:
    - ottimizzazione degli accessi, non una singola lock
    - thread-safeness con miglior performance
    - overlapping di operazioni di scrittura su parti diverse della struttura
    - letture in parallelo a modifiche della struttura
  - svantaggio
    - semantica "approssimata" di alcune operazioni `size()`, `isEmpty()`
    - weak consistency

CONCURRENT HASH MAP



- HashMap: per default 16 buckets (incremento se load factor è alto)
- ConcurrentHashMap
  - lock striping una lock per ogni bucket, associata al primo elemento
  - modifiche simultanee possibili, se modificano sezioni diverse della tabella
    - 16 o più threads possono operare in parallelo su bucket diversi
  - lettori possono accedere in parallelo a modifiche
    - richiede cambiamento della semantica degli iteratori

OPERAZIONI ATOMICHE

- le concurrent collections non permettono al programmatore di sincronizzare sequenze di operazioni su singoli elementi della struttura
- offrono però un insieme di operazioni atomiche
  - sequenze di operazioni di uso comune
  - definita 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..."

OPERARAZIONI ATOMICHE: PERCHE'?

```
import java.util.concurrent.ConcurrentHashMap;
public class CMRaceCondition {
    public static void main(String[] args) throws Exception
    { ConcurrentHashMap<String, String> chm = new ConcurrentHashMap<String, String>();
     chm.put("Roma","Italia");chm.put("Berlino","Germania");chm.put("Parigi",”Francia”);
     System.out.println("La mappa iniziale è: " + chm);
     Thread PutThread = new RaceThread(chm);
     Object value=chm.get("Londra");
     boolean notFound=false;
     if (value == null) {
           notfound=true;
           PutThread.start();
           Thread.sleep(10000); //il thread va in esecuzione e non trova Londra nella Map
           chm.put("Londra", "Great Britain"); }
           PutThread.join();
       if (notfound) {System.out.println("Sono il main ed ho inserito il valore" +
                                          "Londra=Great Britain");};
       System.out.println("E la mappa ora contiene "  + chm); } }
```

OPERARAZIONI ATOMICHE: PERCHE'?

```
import java.util.concurrent.ConcurrentHashMap;
public class RaceThread extends Thread {
        ConcurrentHashMap<String, String> chm;
        public RaceThread (ConcurrentHashMap<String, String> chm)
              {this.chm=chm; };
        public void run()
          {
           Object value=chm.get("Londra");
           if (value == null) {
           try {
                Thread.sleep(50000);
                // do la possibilità al Main di inserire nella Map il Valore Great Britain
                // ma quando mi risveglio, aggiorno Great Britain con UK
                }
           catch(Exception e) {   }
           chm.put("Londra", "UK");
         }
       }}
```

RACE CONDITION SU CONCURRENT MAP

- output restituito dal programma precedente

```
La mappa iniziale è: {Berlino=Germania, Roma=Italia, Parigi=Francia}
Sono il main ed ho inserito il valore Londra=Great Britain
E la mappa ora contiene {Berlino=Germania, Roma=Italia, Londra=UK,
Parigi=Francia}
```

- generata una race condition!
  - il main si è sospeso, dopo aver verificato che "Londra" non esiste
  - anche il thread si sospende dopo aver verificato che "Londra" non esiste
  - il main si è risveglia prima del thread ed inserisce "Londra-Gran Bretagna"
  - successivamente il thread ritorna in esecuzione ed inserisce "Londra-UK"
  - la stampa effettuata nel main mostra l'inconsistenza

OPERARAZIONI ATOMICHE

```
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}
```

- nuove funzioni per garantire atomiticità ad operazioni composte
- funzioni del tipo "`query-then-update`", "`test-and-set`"
- a cosa servono, se la `ConcurrentMap` garantisce la sincronizzazione?

ELIMINARE LA RACE CONDITION

```
import java.util.concurrent.ConcurrentHashMap;
public class PutIfAbsentExample {
    public static void main(String[] args) throws Exception
    { ConcurrentHashMap<String, String> chm =
                              new ConcurrentHashMap<String,  String>();
      chm.put("Roma", "Italia");
      chm.put("Berlino", "Germania");
      chm.put("Parigi", "Francia");
      System.out.println("La mappa iniziale è: " + chm);
      Thread PutThread = new RaceThread(chm);
      PutThread.start();
      Thread.sleep(10000);
      String returnedValue = chm.putIfAbsent ("Londra", "Great Britain");
      if (returnedValue==null) {System.out.println("Sono il main ed ho
                  inserito il valore" + "Londra=Great Britain");};
       System.out.println("Sono il Main e la mappa ora contiene "+chm); } }
```

ELIMINARE LA RACE CONDITION

```
public class RaceThread extends Thread {
   ConcurrentHashMap<String, String> chm;
   public RaceThread (ConcurrentHashMap<String, String> chm)
          {this.chm=chm; };
   public void run()
      {
       Object returnedValue = chm.putIfAbsent ("Londra", "UK");
       if (returnedValue==null)
         {System.out.println("Sono il Thread ed ho inserito il
        valore" + "Londra=Great UK");};
        System.out.println("Sono il Thread e la mappa ora contiene "
                           + chm);
        }
}
```

JAVA COLLECTION FRAMEWORK: ITERATORI

- iteratori:
  - oggetto di supporto usato per accedere agli elementi di una collezione, uno alla volta e in sequenza
  - associato ad un oggetto collezione
  - 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()` in modo diverso
  - l'interfaccia `Iterator` prevede tutti i metodi necessari per usare un iteratore, senza conoscere alcun dettaglio implementativo

COLLECTIONS: ITERATORI

- 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)
}
```

- 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)

COLLECTIONS: UN ESEMPIO CONCRETO

```
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 `foreach`: `for (String s : v)`
  corrisponde a creare implicitante un iteratore per la collezione `v`
  alcuni vincoli rispetto all'iteratore generico

FAIL FAST E FAIL SAFE OPERATORS

- Concurrent Modification
  - modificare una collezione mentre un altro thread sta iterando sulla collezione in modo concorrente
- Fail-fast iterators
  - sollevano una `ConcurrentModificationException` se c'è una modifica strutturale (inserzione, rimozione, aggiornamento) alla collezione su cui l'iteratore sta operando
  - iteratori su `ArrayList`, `HashMap`, ed altre collezioni
- Fail-safe iterators
  - non sollevano una `ConcurrentModificationException`
    - creano un clone della collection e lavorano su quello
  - iteratori su `ConcurrentHashMap`, `CopyOnWriteArrayList`, etc.

UN ITERATORE FAIL FAST

```
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)
```

UN ITERATORE FAIL SAFE SENZA COPIA

```
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
```

UN ITERATORE FAIL SAFE CON COPIA

```
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.Iterator;
class FailSafe {
    public static void main(String args[])
    { CopyOnWriteArrayList<Integer> list
                  = new CopyOnWriteArrayList<Integer>(new Integer[] { 1, 3, 5, 8 });
        Iterator <Integer>  itr = list.iterator();
        while (itr.hasNext()) {
            Integer no = (Integer)itr.next();
            System.out.println(no);
            if (no == 8)
                // This will not print,
                // hence it has created separate copy
                list.add(14);
        } } }
         // stampa 1 3 5 8
```

CONCURRENT HASH MAP

- un iteratore
  - se non clona, la collezione può catturare le modifiche effettute sulla collezione dopo la sua creazione
  - se clona, le modifiche possono non essere visibili all'iteratore

- alcuni metodi, `size()` e `isEmpty()`
  - possono restituire un valore "approssimato"
  - "weakly consistent behaviour"

- 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."

CONCURRENT HASH MAP: UN CASO D'USO

- simulazione di un magazzino rappresentato da una ConcurrentHashMap
  - chiave: nome della merce
  - valore: presenza o meno della merce in magazzino (in stock, sold out)

- simulazione effettuta mediante i seguenti threads:
  - HashMapStock: caricamento iniziale della merce nel magazzino
  - HashMapEmpty: simula uscita degli articoli dal magazzino
  - HashMapReplace: simula ricarico articoli in magazzino

CONCURRENT HASH MAP: UN CASO D'USO

```
import java.util.concurrent.*;
public class HashMapExample {
public static void main(String args[])
     {ConcurrentHashMap <String,String> concurrentMap = new
                          ConcurrentHashMap <String,String>();
      ThreadPoolExecutor executor  =
          (ThreadPoolExecutor) Executors.newCachedThreadPool();
      executor.execute(new HashMapStock(concurrentMap));
      executor.execute(new HashMapEmpty(concurrentMap));
      executor.execute(new HashMapReplace(concurrentMap));
      }}
```

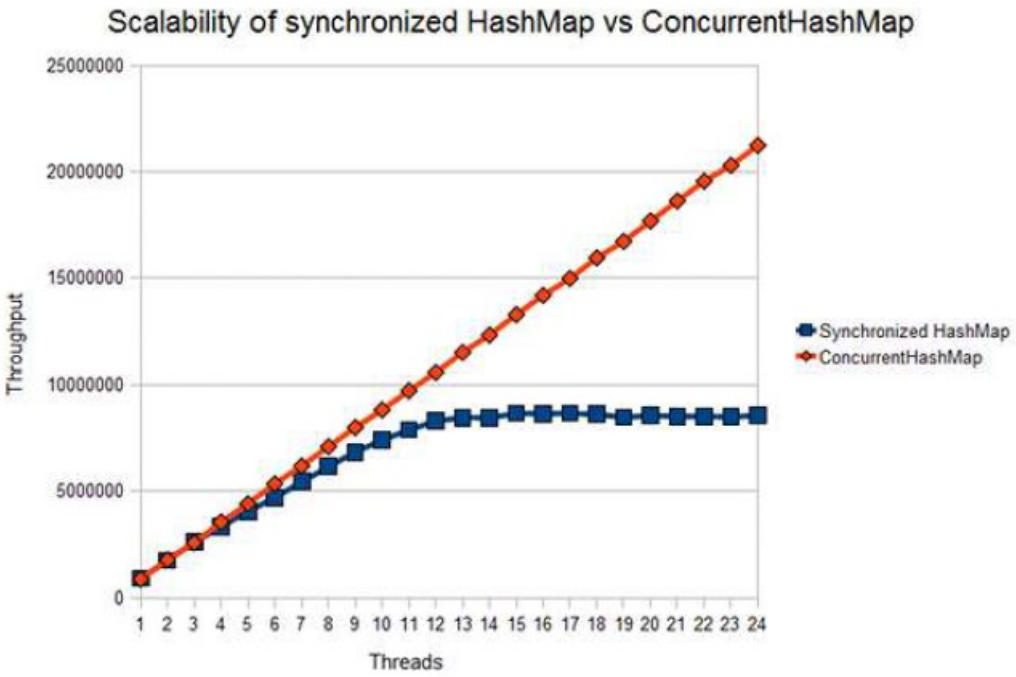
CONCURRENT HASH MAP: UN CASO D'USO

```
import java.util.concurrent.ConcurrentHashMap;
public class HashMapStock implements Runnable{
   ConcurrentHashMap <String,String> cMap = new ConcurrentHashMap
                                                    <String,String>();
   public HashMapStock (ConcurrentHashMap <String, String> cMap)
       {this.cMap= cMap;}
   public void run ()
       { cMap.putIfAbsent ("Socks", "in stock");
         System.out.println("Socks in stock");
         cMap.putIfAbsent ("Shirts", "in stock");
         System.out.println("Shirts in stock");
         cMap.putIfAbsent ("Pants", "in stock");
         System.out.println("Pants in stock");
         cMap.putIfAbsent ("Shoes", "in stock");
         System.out.println("Shoes in stock"); }}
```

CONCURRENT HASH MAP: UN CASO D'USO

```
import java.util.Iterator; import java.util.Random;
import java.util.concurrent.*;
public class HashMapEmpty implements Runnable{
    ConcurrentHashMap <String,String> cMap = new ConcurrentHashMap
                                                    <String,String>();
    public HashMapEmpty (ConcurrentHashMap <String, String> cMap)
       {this.cMap= cMap;}
    public void run ()
       {while(true)
         { Iterator <String> i =cMap.keySet().iterator();
           while(i.hasNext())
             {String s= i.next();
              if (cMap.replace(s,"in stock", "sold out"))
                     {System.out.println(s+"sold out"); }
           try { Thread.sleep(new Random().nextInt(500));
               } catch (InterruptedException e) {e.printStackTrace();}}}}}
```

HASHMAP E CONCURRENT HASH MAP



©https://www.javamex.com/tutorials/concurrenthashmap_scalability.shtml

throughput: the total number of accesses to the map performed by all threads together in two seconds.

Intel i7-720QM (Quad Core Hyperthreading) machine running Hotspot version 1.6.0_18 under Windows 7.