Laboratorio di Reti – A (matricole pari)

Autunno 2021,
instructor: Laura Ricci

laura.ricci@unipi.it

Lezione 4

Volatile, Atomic Variables, Monitor, Synchronized Collections

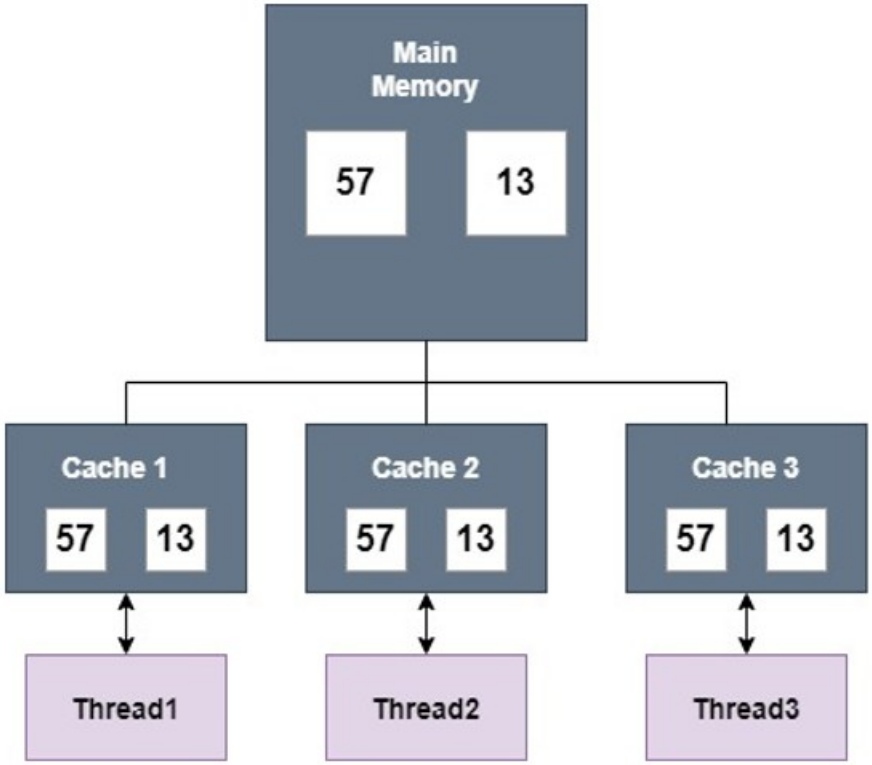
5/10/2021

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`

IL PROBLEMA DELLA VISIBILITA'

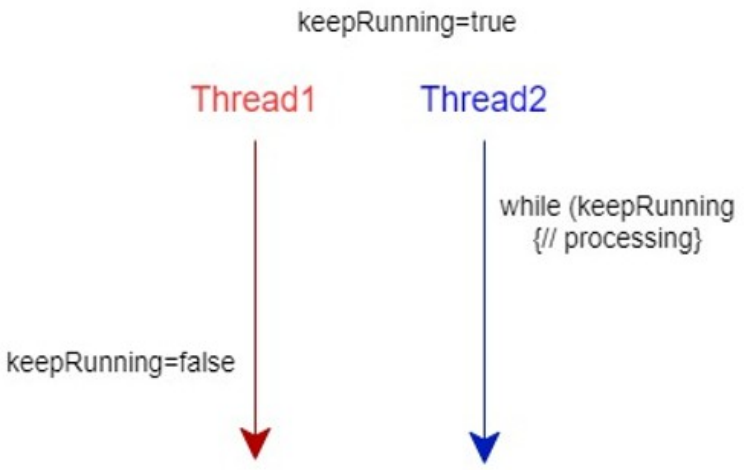


architettura di riferimento, utile per capire il problema della visibilità

non l'unica possibile

IL PROBLEMA DELLA VISIBILITA'

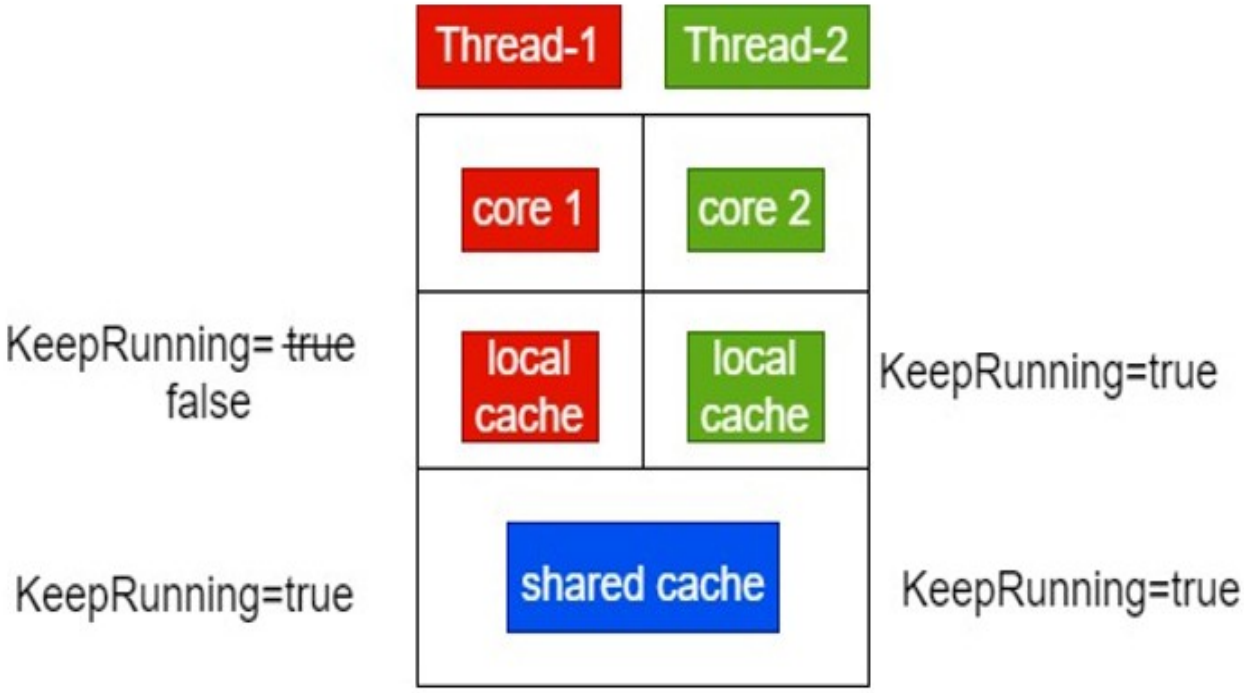
```
class Test extends Thread {
    boolean keepRunning = true;
    public void run() {
      while (keepRunning) { }
      System.out.println("Thread terminated.");
                        }
```




```
    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'



- quando il Thread-1 aggiorna `KeepRunning`, è possibile che la modifica non sia riportata nello shared cache
- il problema riguarda la "visibilità" della modifica, non la sincronizzazione.
  - read e write di un booleano sono atomiche

IL PROBLEMA DELLA VISIBILITA': SOLUZIONE

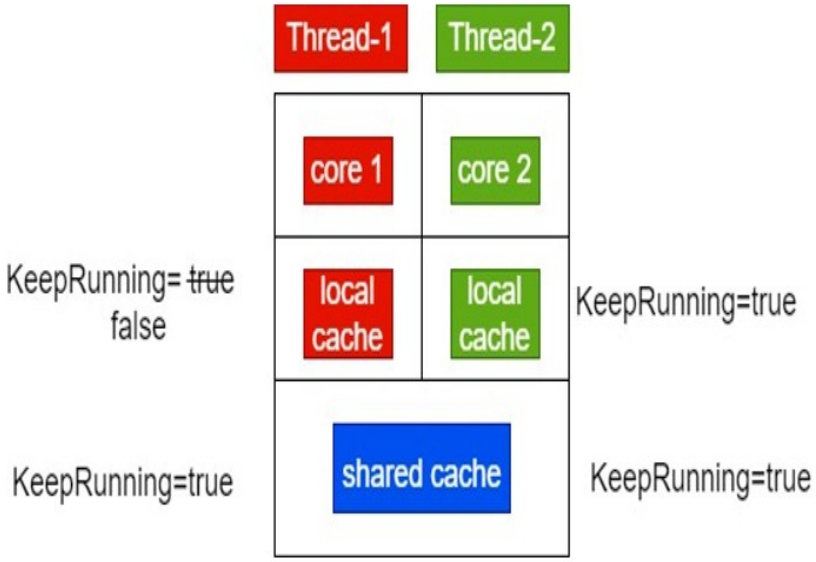
- modifichiamo la dichiarazione

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

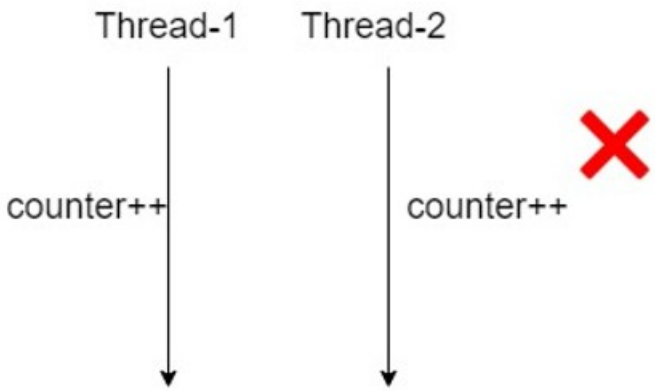
con la keyword `volatile`

  - l'aggiornamento ad una variabile `volatile` è sempre effettuato nella main memory
    - flush della cache
  - il valore della variabile `volatile` è sempre letto dalla memoria

- meccanismo che fornisce thread safeness nella lettura/scrittura di singole variabili
- usate anche per rendere atomici aggiornamenti a variabili di tipo long



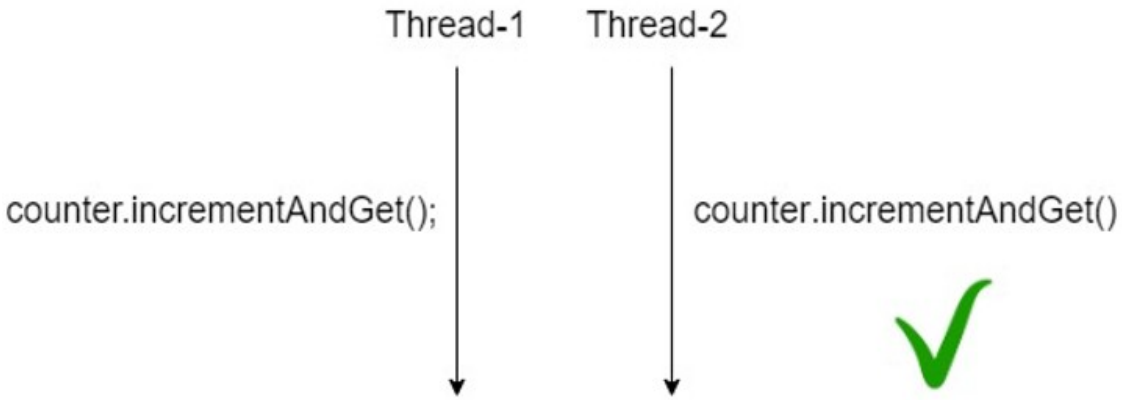
SINCRONIZZAZIONE SU SINGOLE VARIABILI



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 in modo concorrente, un aggiornamento può andare perduto (anche se la variabile è volatile)
- ovviamente il problema può essere risolto con le lock
- souzione alternativa: usare le variabili `Atomic`

ATOMIC VARIABLES



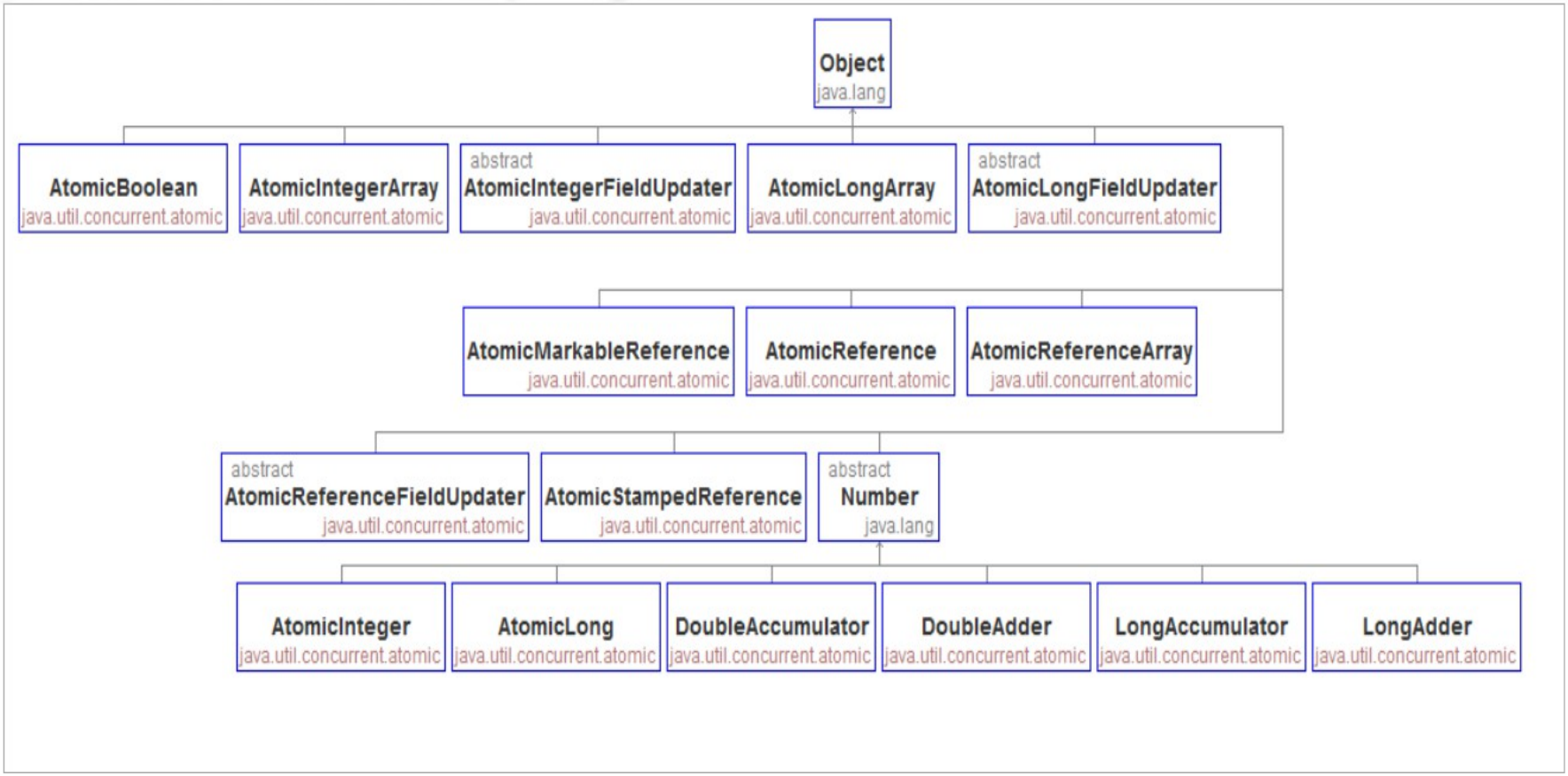

```
AtomicInteger value = new AtomicInteger(1);
```

- operazioni atomiche che non richiedono sincronizzazioni esplicite o lock: è la JVM che garantisce la atomicità
  - `incrementAndGet()`: atomically increments by one
  - `decrementAndGet()`: atomically decrements by one
  - `compareAndSet(int expectedValue, int newValue)`
- molte altre classi
  - `AtomicLong`
  - `AtomicBoolean`
  - ...

ATOMIC VARIABLES: UN ESEMPIO

```
import java.util.concurrent.*; import java.util.concurrent.atomic.*;
public class AtomicIntExample {
public static void main(String[] args) {
   ExecutorService executor = Executors.newFixedThreadPool(2);
   AtomicInteger atomicInt = new AtomicInteger();
   for(int i = 0; i < 10; i++){
        CounterRunnable runnableTask = new  CounterRunnable(atomicInt);
        executor.submit(runnableTask);
    }
    executor.shutdown(); }
 class CounterRunnable implements Runnable {
      AtomicInteger atomicInt;
      CounterRunnable(AtomicInteger atomicInt){this.atomicInt = atomicInt;}
      @Override
      public void run() {
        System.out.println("Counter- " + atomicInt.incrementAndGet());}}
```

JAVA.UTIL.CONCURRENT.ATOMIC



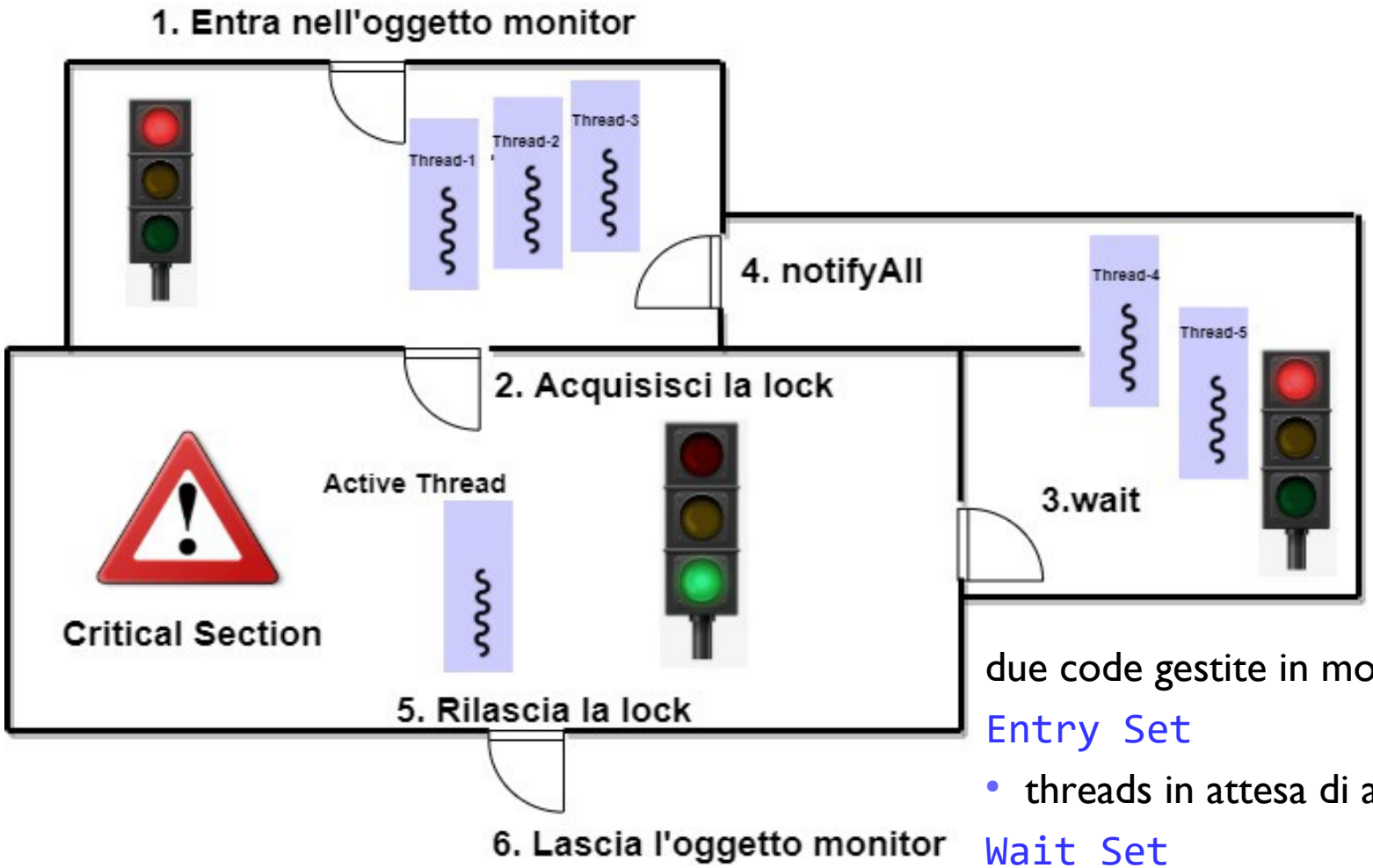
IL MONITOR

- meccanismo linguistico ad alto livello per la sincronizzazione
  - idea introdotta negli anni '70 safe (*Per Brinch Hansen, Hoare 1974*)
- incapsula un oggetto condiviso e le operazioni che vengono invocate dai threads su di esso, in modo concorrente
- funzionalità offerte dal monitor
  - mutua esclusione sulla struttura: lock implicite gestite dalla JVM: un solo thread per volta accede all'oggetto condiviso
  - coordinazione tra i thread
    - meccanismi per la sospensione sullo stato dell'oggetto condiviso, simili a variabili di condizione: `wait`
    - meccanismi per la notifica di una condizione ai thread sospesi su quella condizione + `notify/notifyall`

IL MONITOR

- JAVA built-in monitor: classe di oggetti utilizzabili concorrentemente in modo thread safe
  - meccanismi di sincronizzazione "ad alto livello"
- come viene implementato? ad ogni oggetto (non `int` o `long`, solo gli oggetti), cioè ad ogni istanza di una classe, viene associata
  - una "intrinsic lock" o lock implicita
    - acquisita con metodi o blocchi di codice `synchronized`. Garantisce la mutua esclusione nell'accesso all'oggetto
    - gestione automatica della coda di attesa, da parte della JVM
  - una "intrinsic condition variable" associata ad una "wait queue" gestita dalla JVM
    - `wait` (come await per condition variables)
    - `notify/notifyAll` (come signal per condition variables)

UN'OCCHIATA ALL'INTERNO DI UN MONITOR



due code gestite in modo implicito:

`Entry Set`

- threads in attesa di acquisire la lock

`Wait Set`

- threads che hanno eseguito una `wait` e sono in attesa di una `notifyAll`

METODI SINCRONIZZATI

- i metodi di un built-in monitor possono essere resi thread safe annotandoli con la parola chiave `synchronized`
- dalla lezione precedente: coda thread-safe, implementata con monitor

```
public class MessageQueue {

    public MessageQueue(int size)

    public synchronized void produce(Object x)

    public synchronized Object consume()
```

- l'esecuzione di un metodo synchronized richiede automaticamente l'acquisizione della la lock intrinseca associata all'oggetto
- l'intero codice del metodo sincronizzato viene serializzato rispetto agli altri metodi sincronizzati definiti per lo stesso oggetto
  - solo una thread alla volta può essere eseguire uno dei metodi synchronized del monitor sulla stessa istanza di una classe

LOCK INTRINSECHE: METODI SINCRONIZZATI

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

metodo `synchronized` : quando viene invocato

- tenta di acquisire la lock intrinseca associata all'istanza dell'oggetto su cui esso è invocato
  - istanza riferita dalla parola chiave `this`
  - se l'oggetto è bloccato 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
  - normale
  - eccezionale, ad esempio con una `uncaught exception`.

LOCK INTRINSECHE: METODI SINCRONIZZATI

- i costruttori non devono essere dichiarati `synchronized`
  - il compilatore solleva una eccezione
  - per default, solo il thread che crea l'oggetto accede ad esso mentre l'oggetto viene creato
- non ha senso specificare `synchronized` nelle interfacce
- `synchronized` non è ereditato da overriding
  - metodo nella sottoclasse deve essere esplicitamente definito `synchronized`, se necessario
- la lock è associata ad un'istanza dell'oggetto, non alla classe, metodi su oggetti che sono istanze diverse della stessa classe possono essere eseguiti in modo concorrente!

WAITING AND COORDINATION MECHANISMS

- JAVA fornisce 3 metodi di base per coordinare i thread
- invocati su un oggetto,

  appartengono alla classe `Object`
- occorre acquisire la lock intrinseca prima di invocarli, altrimenti viene sollevate l'eccezione `IllegalMonitorException()`
  - eseguiti all'interno di metodi sincronizzati
- se non si mette il riferimento ad un oggetto, il riferimento implicito è `this`

`void wait()`

- sospende il thread fino a che un altro thread invoca una `notify()` /`notifyAll()` sullo stesso oggetto.
- implementa una "attesa passiva" del verificarsi di una condizione
- rilascia la lock sull'oggetto

`void notify()`

- sveglia un singolo thread in attesa su questo oggetto
- nop se nessun thread è in attesa

`void notifyAll()`

- sveglia tutti i thread in attesa su questo oggetto, che competono per riacquisire della lock

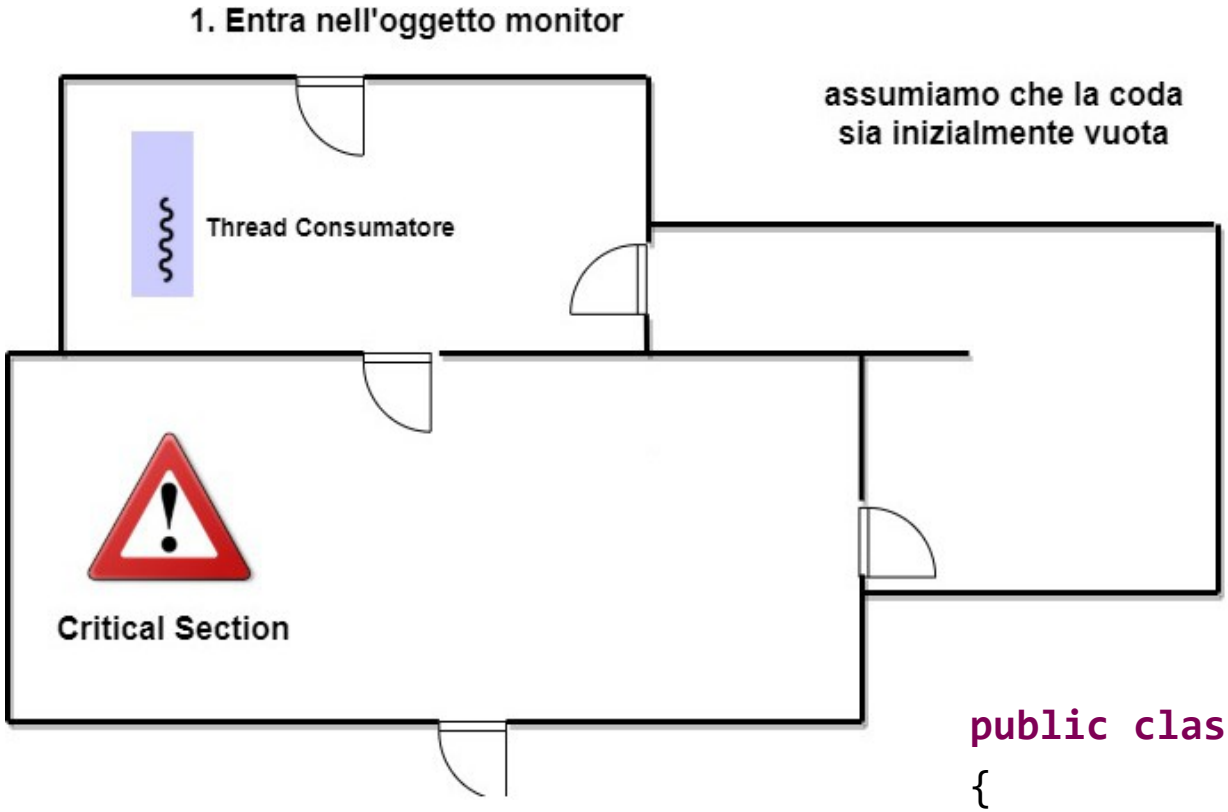
PRODUTTORE CONSUMATORE CON MONITOR

```
public class MessageQueue {
   int   putptr, takeptr, count;
   final Object[] items;
   public MessageQueue(int size){
      items = new Object[size];
      count=0;putptr=0;takeptr=0;}
   public synchronized void  produce(Object x)
     { while (count == items.length)
         try {
               wait();}
         catch(Exception e) {}
         // gestione puntatoricoda
         items[putptr] = x; putptr++;++count;
         if (putptr == items.length) putptr = 0;
         System.out.println("Message Produced"+x);
         notifyAll();}
```

PRODUTTORE CONSUMATORE CON MONITOR

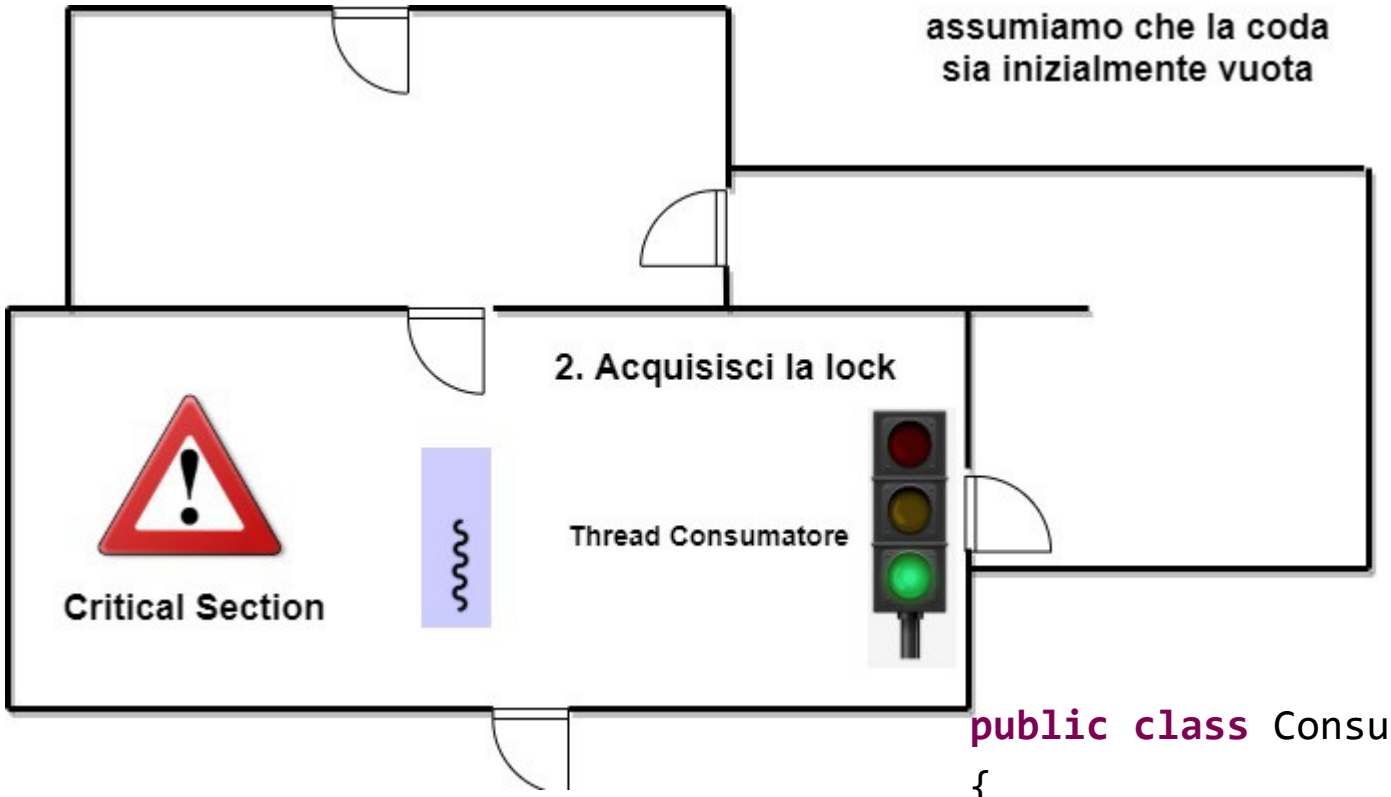
```
public synchronized Object consume() {
        while (count == 0)
         try {
              wait();}
            catch(InterruptedException e) {}
        // gestione puntatori coda
        Object data  = items[takeptr]; takeptr=takeptr+1; --count;
        if (takeptr == items.length) {takeptr = 0;};
        notifyAll();
        System.out.println("Message Consumed"+data);
        return data;
         }}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



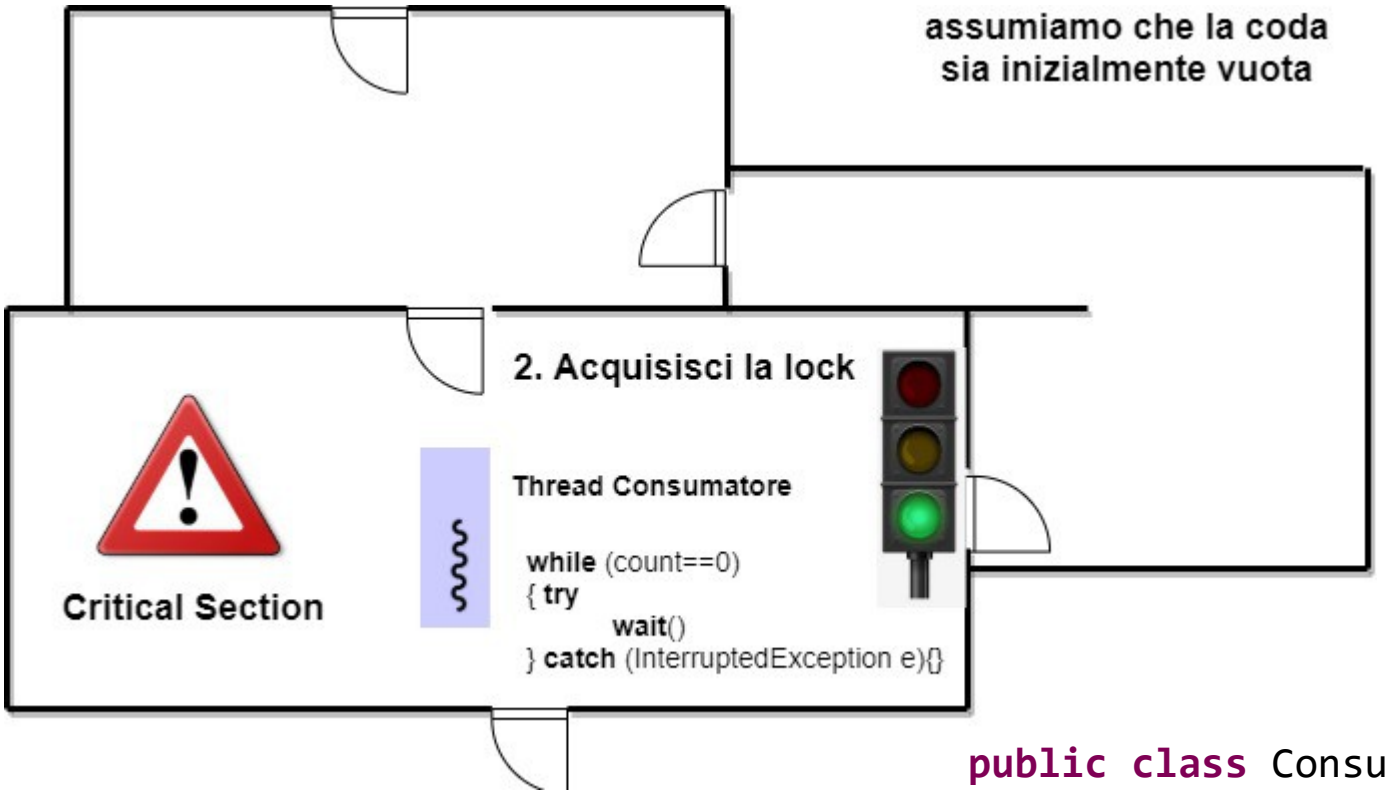

```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



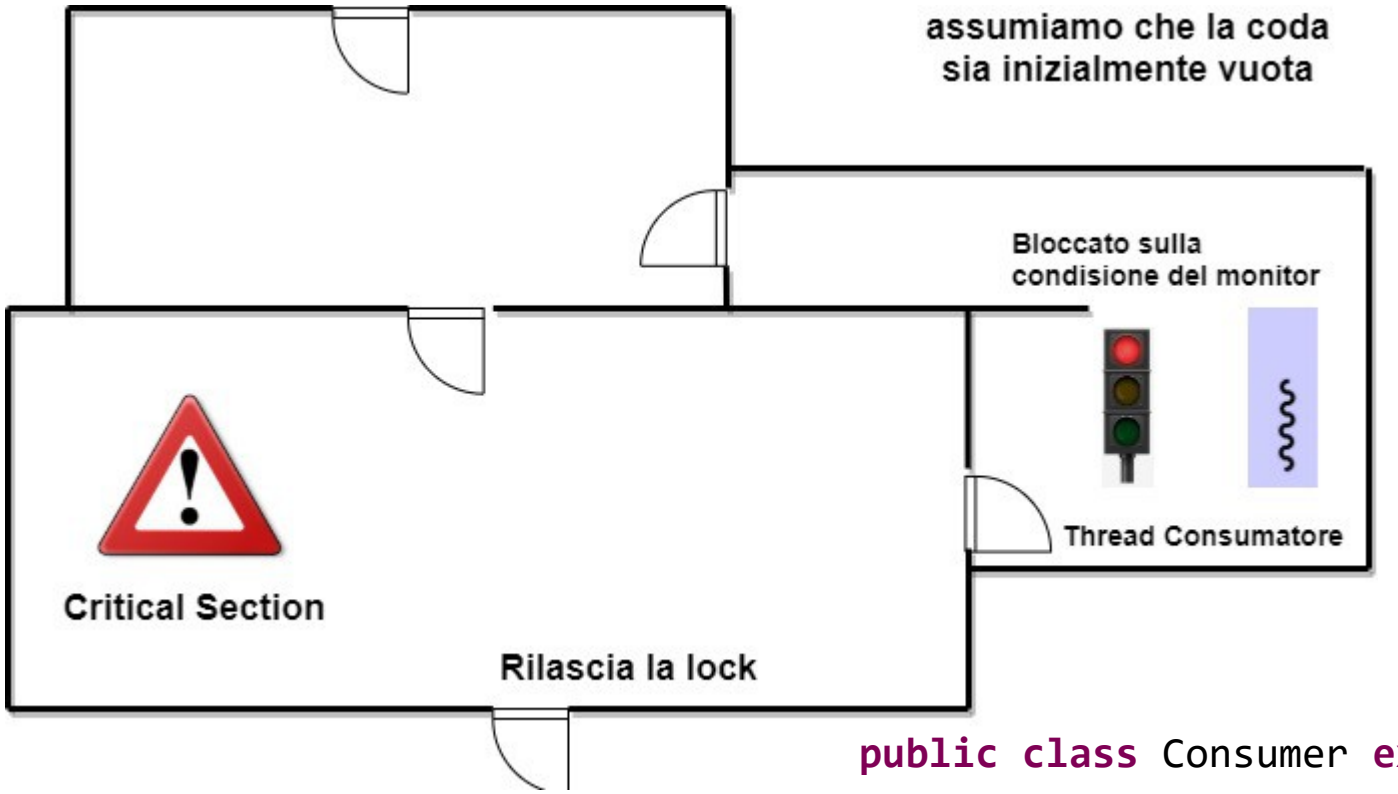

```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



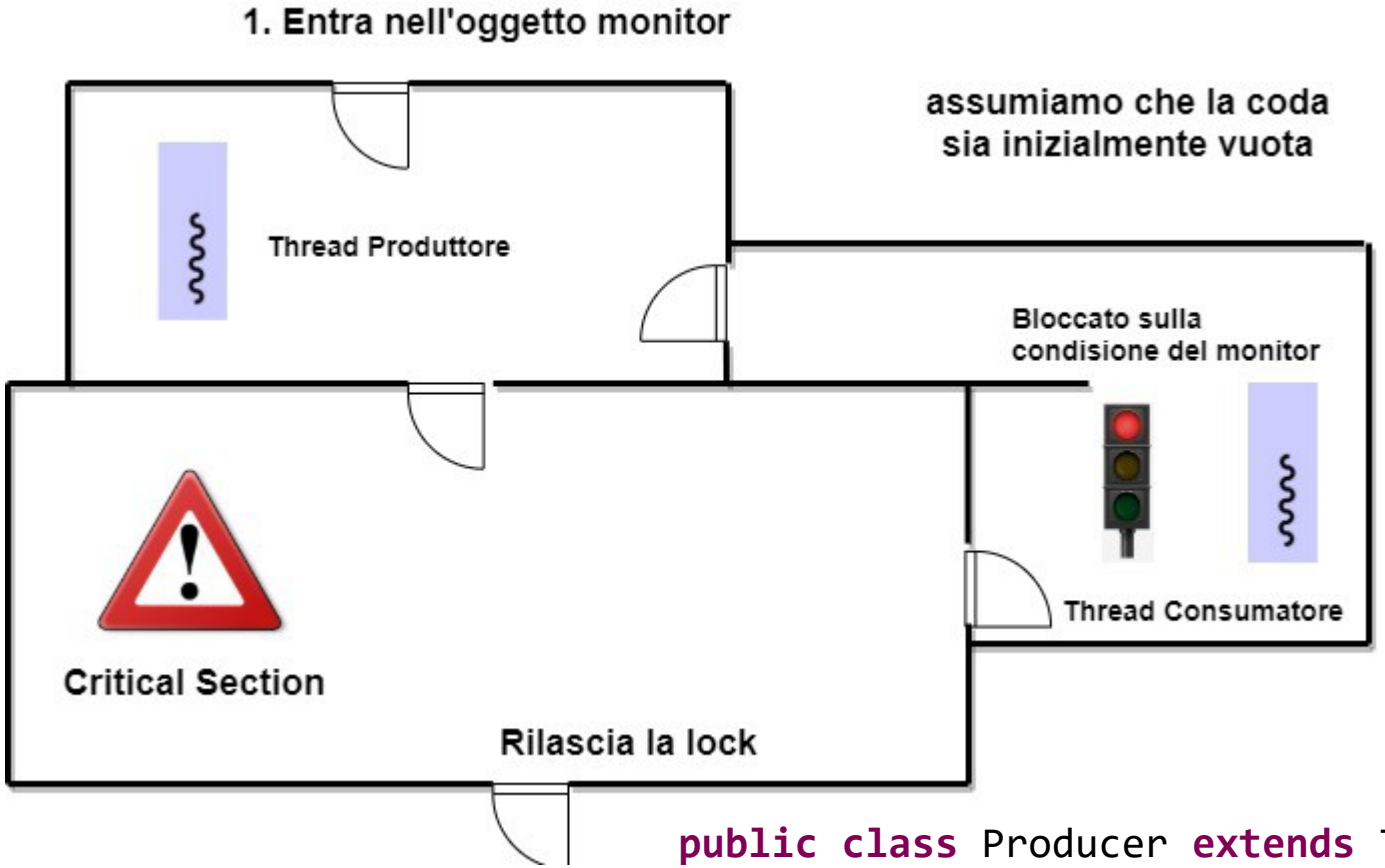

```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



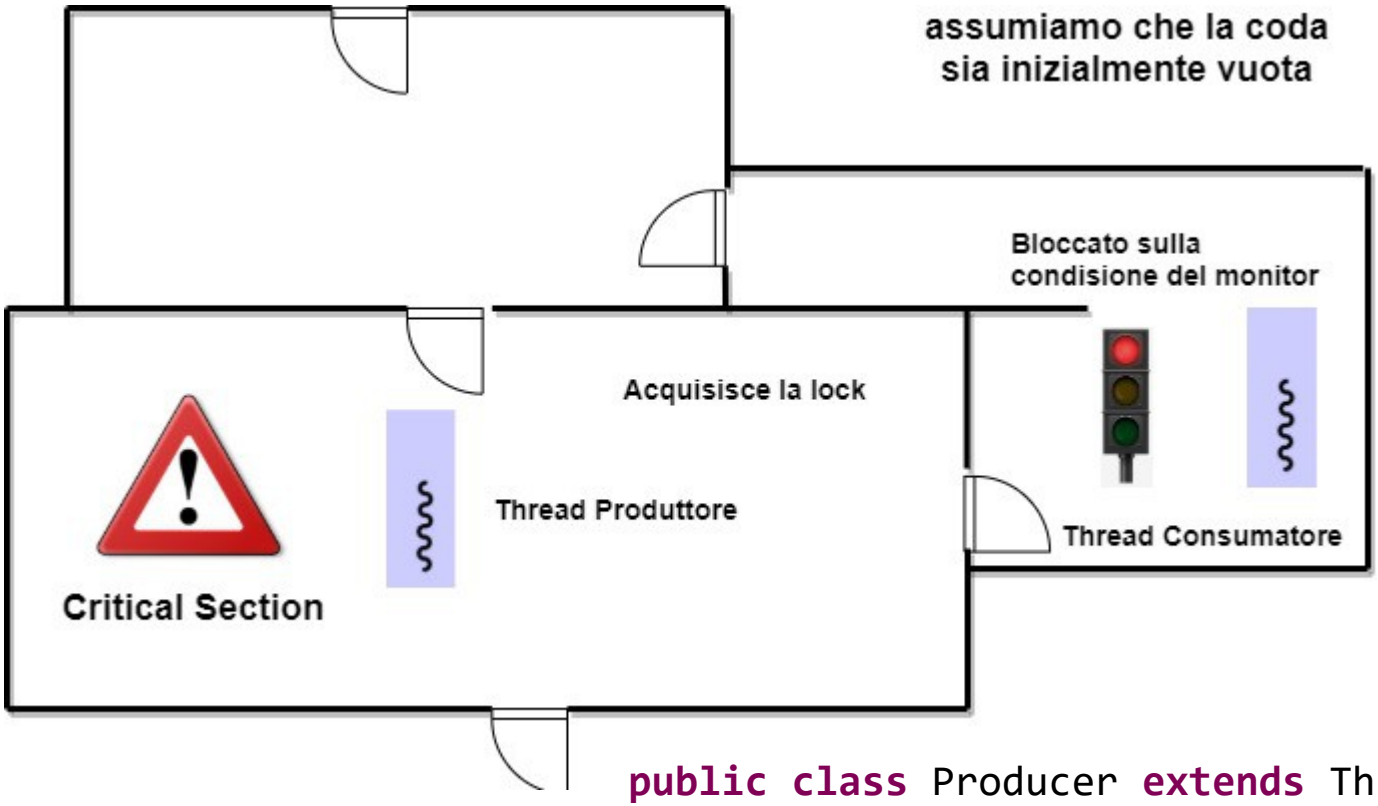

```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



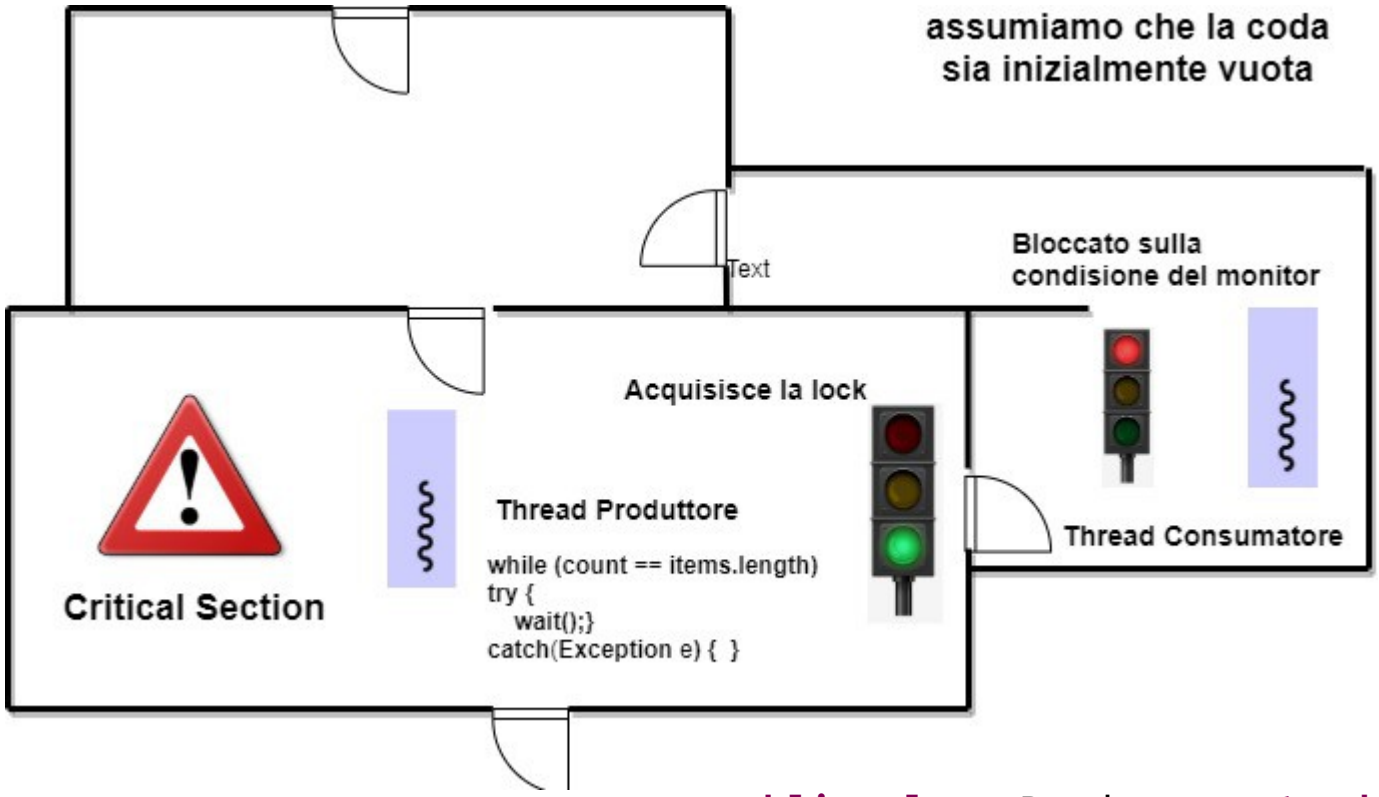

```
public class Producer extends Thread
{
public void run(){
 for(int i=0;i<10;i++)
{queue.produce("MSG#"+count+Thread.currentThread());
…….
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



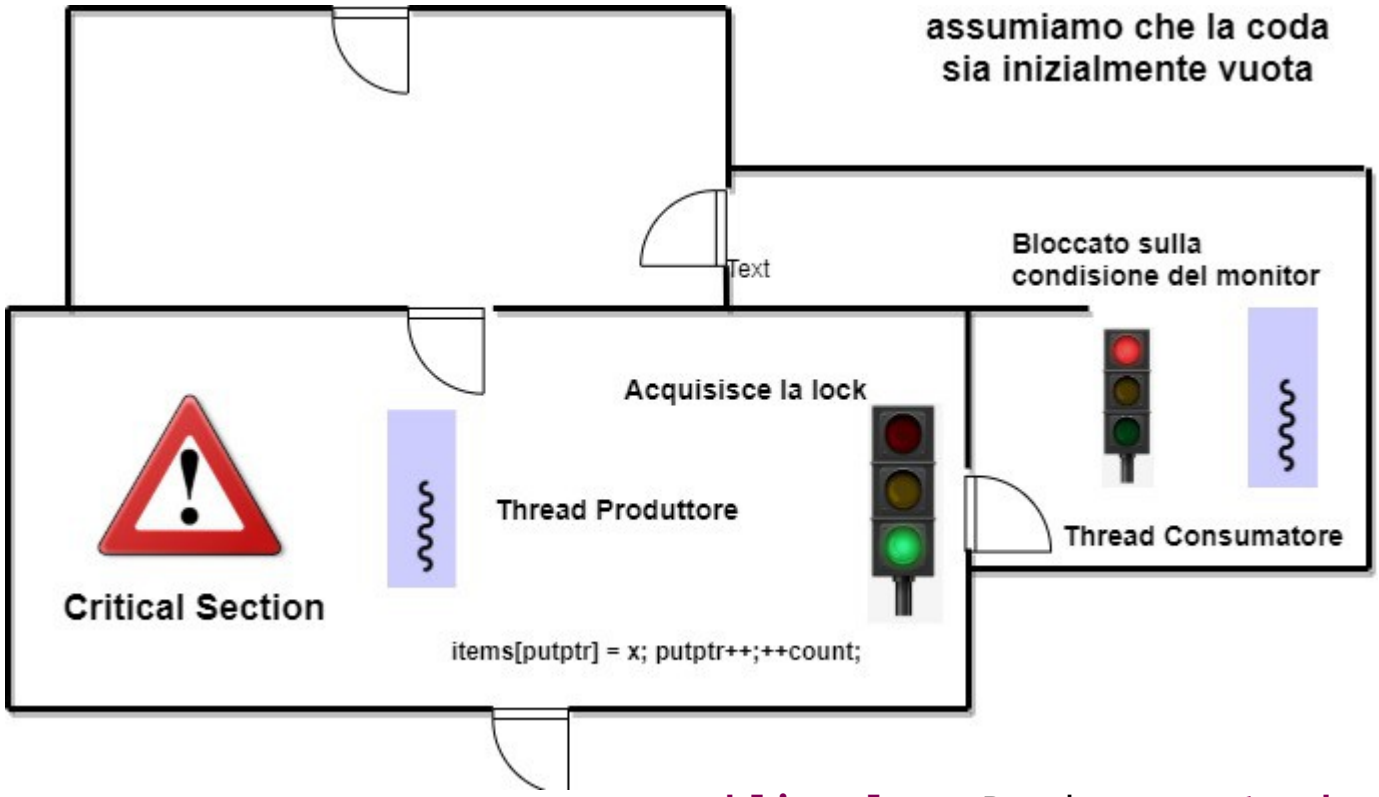

```
public class Producer extends Thread
{
public void run(){
 for(int i=0;i<10;i++)
{queue.produce("MSG#"+count+Thread.currentThread());
…….
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



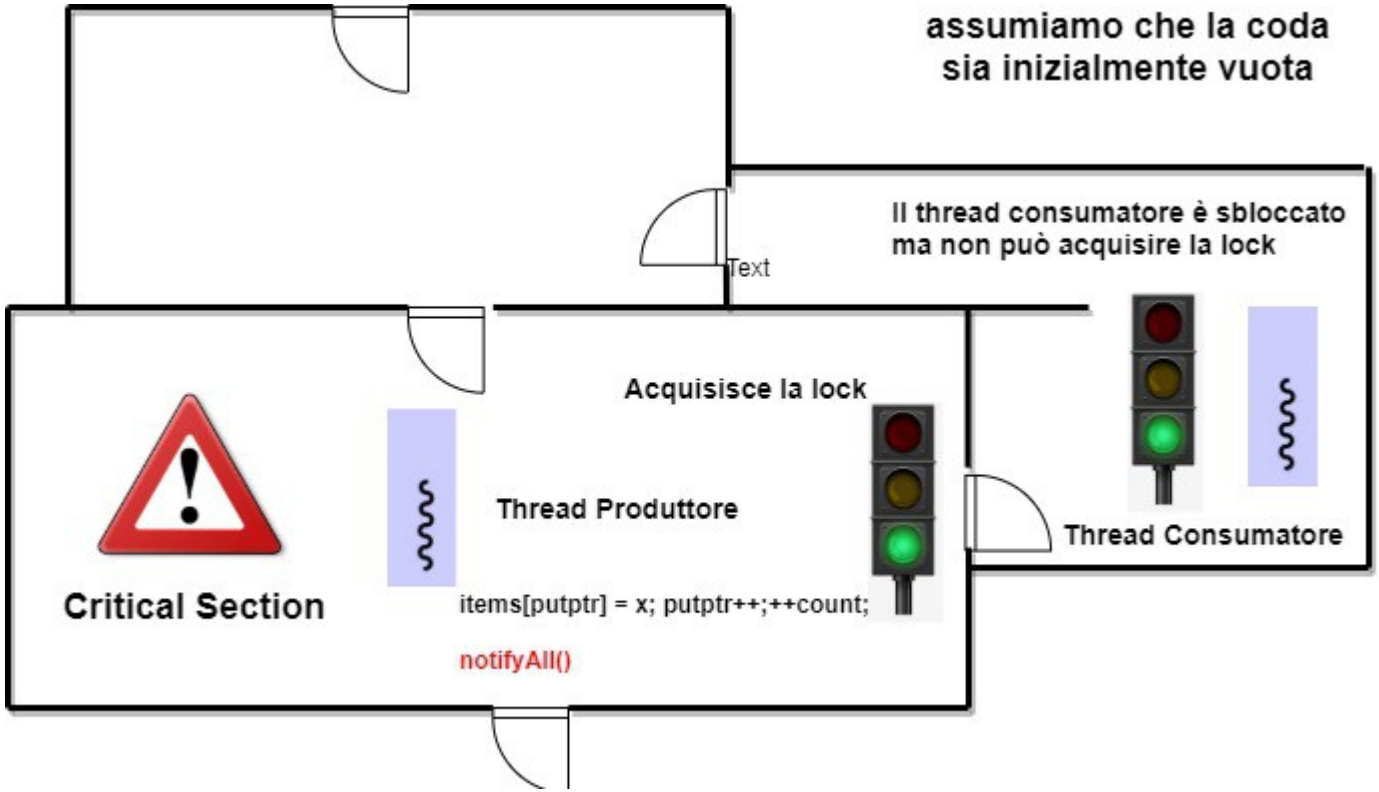

```
public class Producer extends Thread
{
public void run(){
 for(int i=0;i<10;i++)
{queue.produce("MSG#"+count+Thread.currentThread())
…….
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



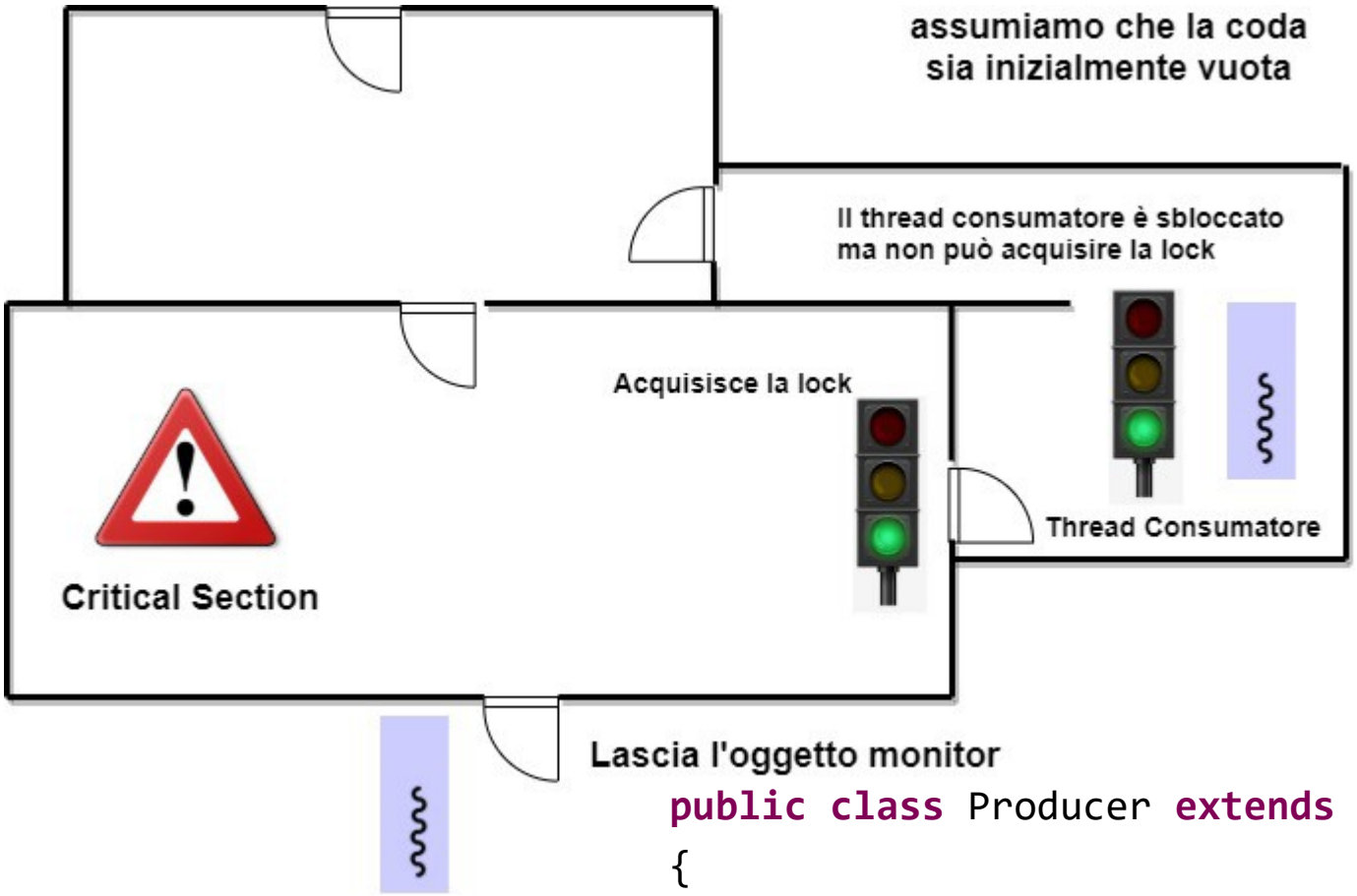

```
public class Producer extends Thread
{
public void run(){
 for(int i=0;i<10;i++)
{queue.produce("MSG#"+count+Thread.currentThread(
…….
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



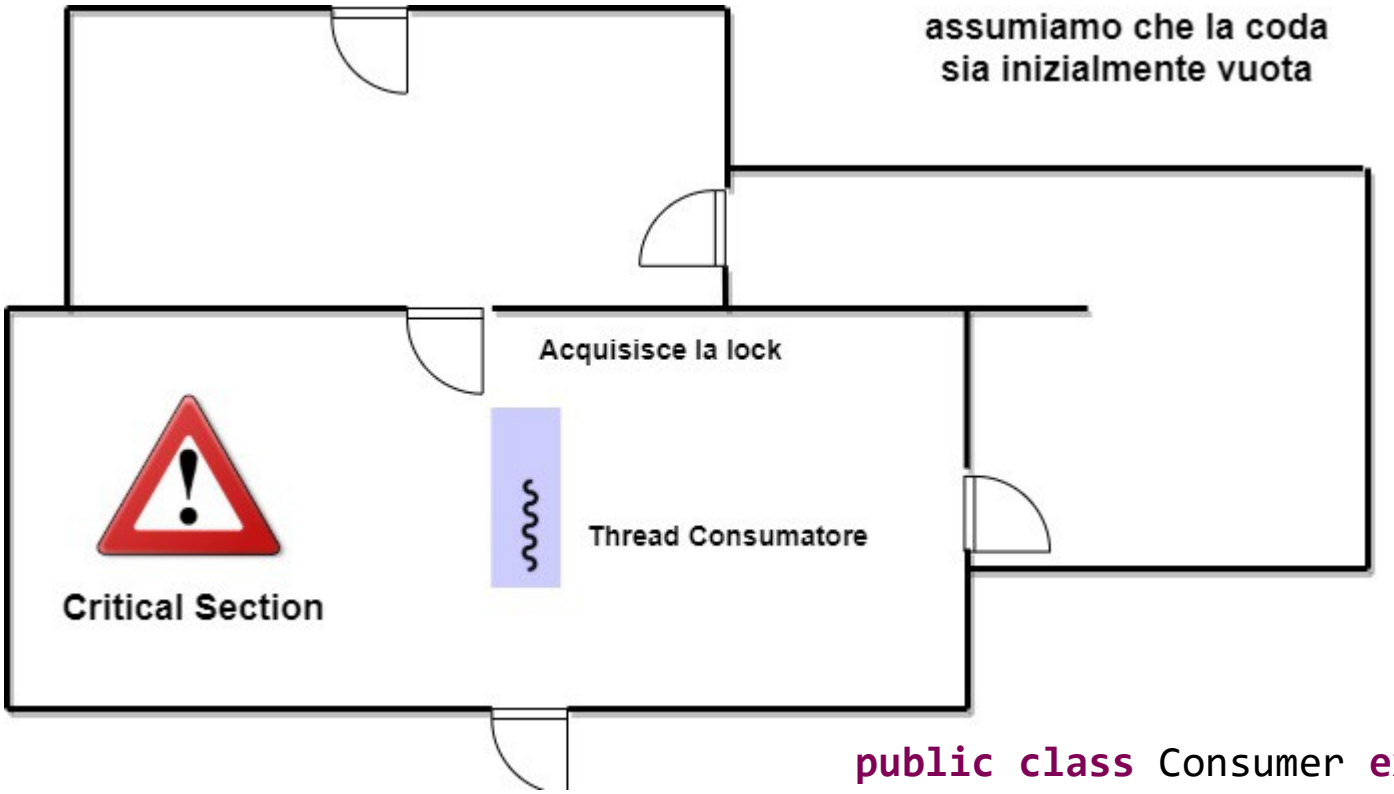

```
public class Producer extends Thread
{
public void run(){
 for(int i=0;i<10;i++)
{queue.produce("MSG#"+count+Thread.currentThread(
…….
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



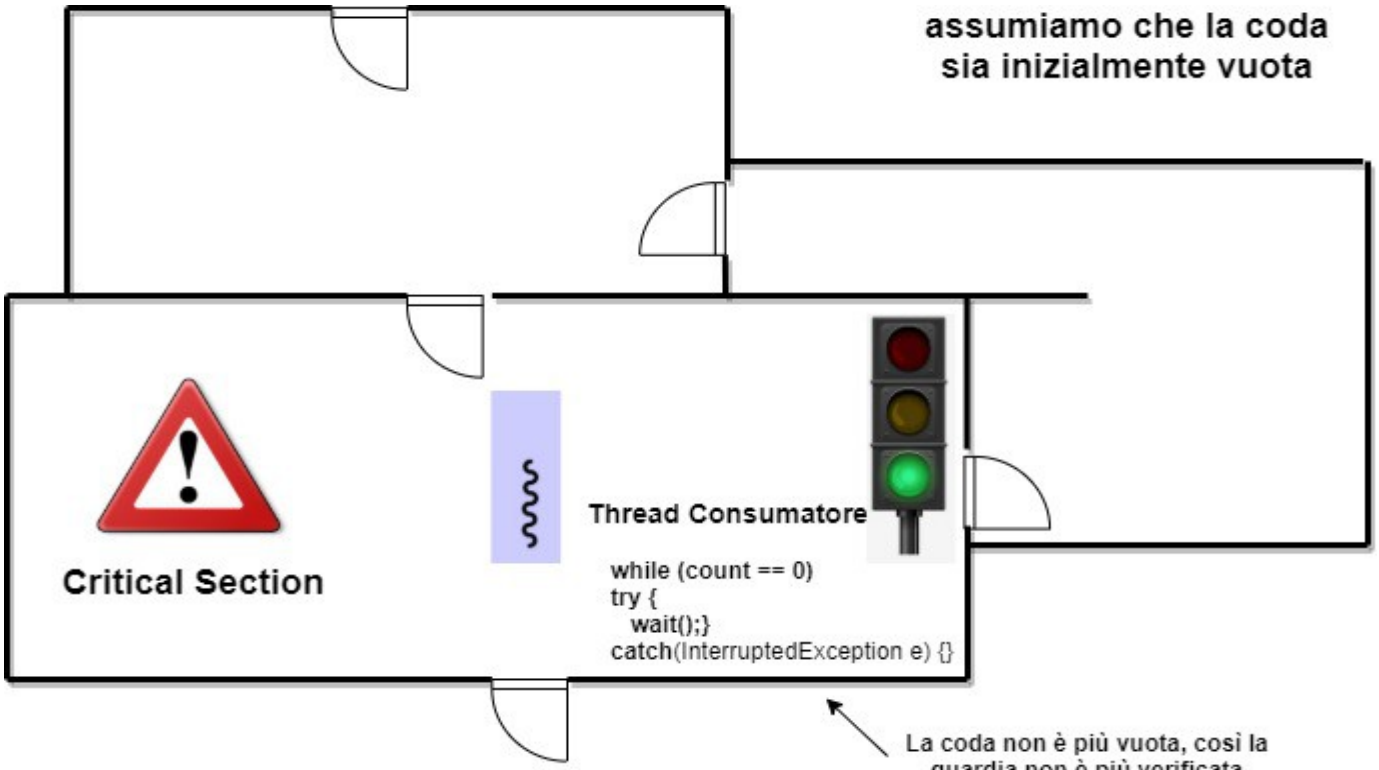

```
public class Producer extends Thread
{
public void run(){
 for(int i=0;i<10;i++)
{queue.produce("MSG#"+count+Thread.currentThread())
…….
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



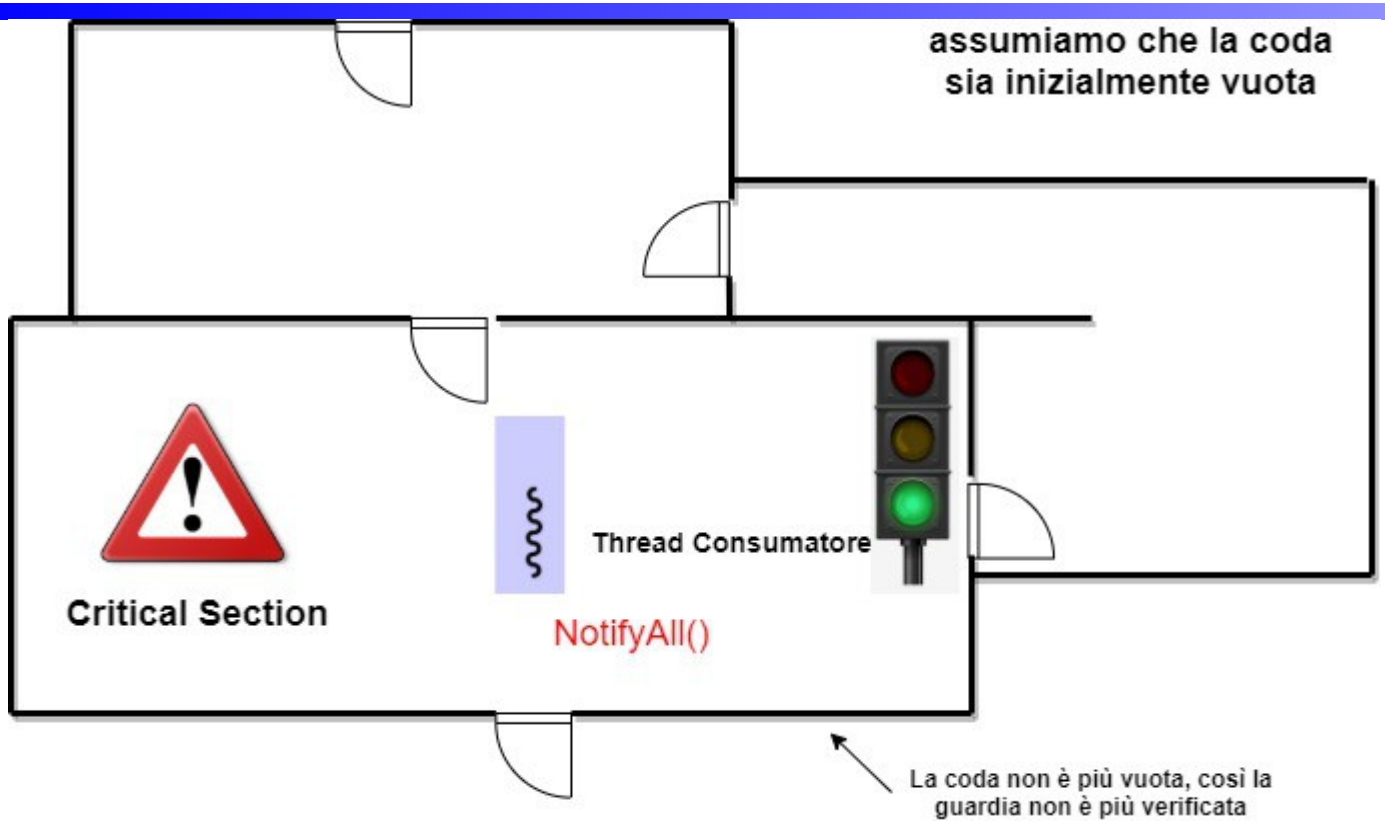

```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"



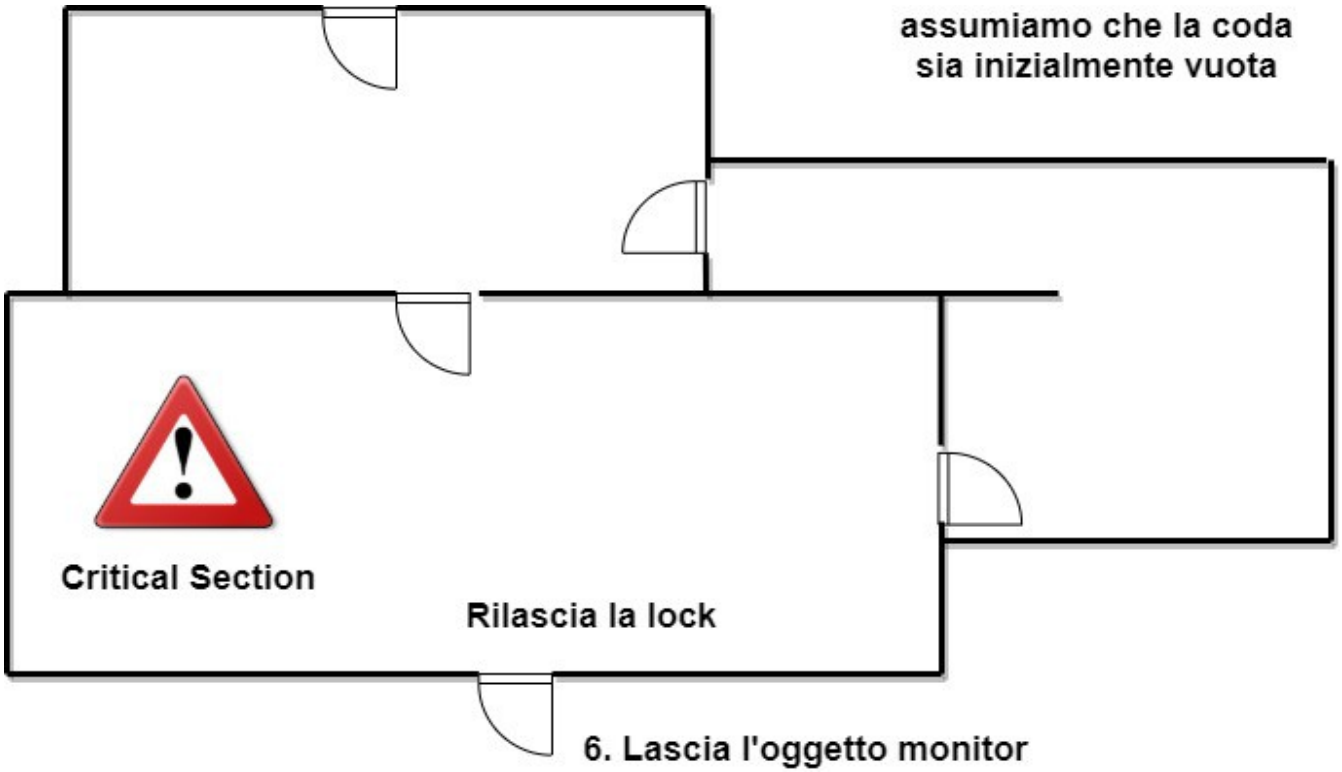

```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"




```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

PRODUTTORE CONSUMATORE "ILLUSTRATO"




```
public class Consumer extends Thread
{
public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
               ….}
```

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

LOCK INTRINSECHE: METODI 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){
            …} } }
```

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

LOCK SCOPE REDUCTION: UN ESEMPIO

- definiamo un object monitor che deve sincronizzare l'accesso a due liste di oggetti
- operazioni
  - aggiungere un elemento alla lista
  - stampare lo stato attuale della lista
- liste implementate come `ArrayList`, una struttura non thread-safe
- più thread accedono in modo concorrente, ma vogliamo implementare una sincronizzazione granulare
  - quando un thread inserisce in ad una lista, l'altro thread può inserire elementi nell'altra lista
  - al momento della stampa, sincronizzazione su entrambe le liste per avere uno "snapshot" dello stato delle liste al momento della stampa (no update concorrenti

JAVA COLLECTION FRAMEWORK: ITERATORI

- per la risoluzione di questo esercizio useremo gli 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

USARE GLI 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, è necessario creare uno nuovo iteratore

USARE GLI ITERATORI

```
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

LOCK SCOPE REDUCTION: UN ESEMPIO

```
import java.util.*;
public class MultipleLockDemo {
    private List<String> list1 = new ArrayList<>();
    private List<String> list2 = new ArrayList<>();
    public static void main (String[] args) {
        MultipleLockDemo obj = new MultipleLockDemo();
        Thread thread1 = new Thread() {public void run() {
                        for (int i = 0; i < 5; i++) {
                            obj.addToList1("thread1 list1 element=" + i);
                            obj.addToList2("thread1 list2 element=" + i);
                            obj.printLists(); };}};
        Thread thread2 = new Thread() { public void run() {
                        for (int i = 0; i < 5; i++) {
                             obj.addToList2("thread2 list2 element=" + i);
                            obj.addToList1("thread2 list1 element=" + i);
                            obj.printLists(); }}};
        thread1.start(); thread2.start();}
```

LOCK SCOPE REDUCTION: UN ESEMPIO

```
public void addToList1 (String s) {
     synchronized (list1) {
          list1.add(s); } }
 public void addToList2 (String s) {
     synchronized (list2) {
          list2.add(s); }  }
 public void printLists () {
     String name = Thread.currentThread().getName();
     synchronized (list1) {
         synchronized (list2) {
           Iterator <String> it1 = list1.iterator();
           while (it1.hasNext())
                 {System.out.println(name+"**"+it1.next()+"**"); }
            Iterator <String> it2 = list2.iterator();
            while (it2.hasNext())
                {System.out.println(name+"**"+it2.next()+"**"); }}}}}
```

cosa accade se elimino la sincronizzazione nella printList? Discuteremo in seguito questa eccezione

```
Exception in thread "Thread-1"
java.util.ConcurrentModificationException
```

LOCK SCOPE REDUCTION

- sincronizzare parti di un metodo, piuttosto che l'intero metodo
- sezioni critiche di dimensione minore all'interno di metodi
- possono usare oggetti "di pura sincronizzazione", oggetti mutex
  - realizzato acquisendo la `lock()` intrinseca di un generico `Object`
- oppure si sincronizzano su un oggetto specifico o su this

```
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

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()}
```

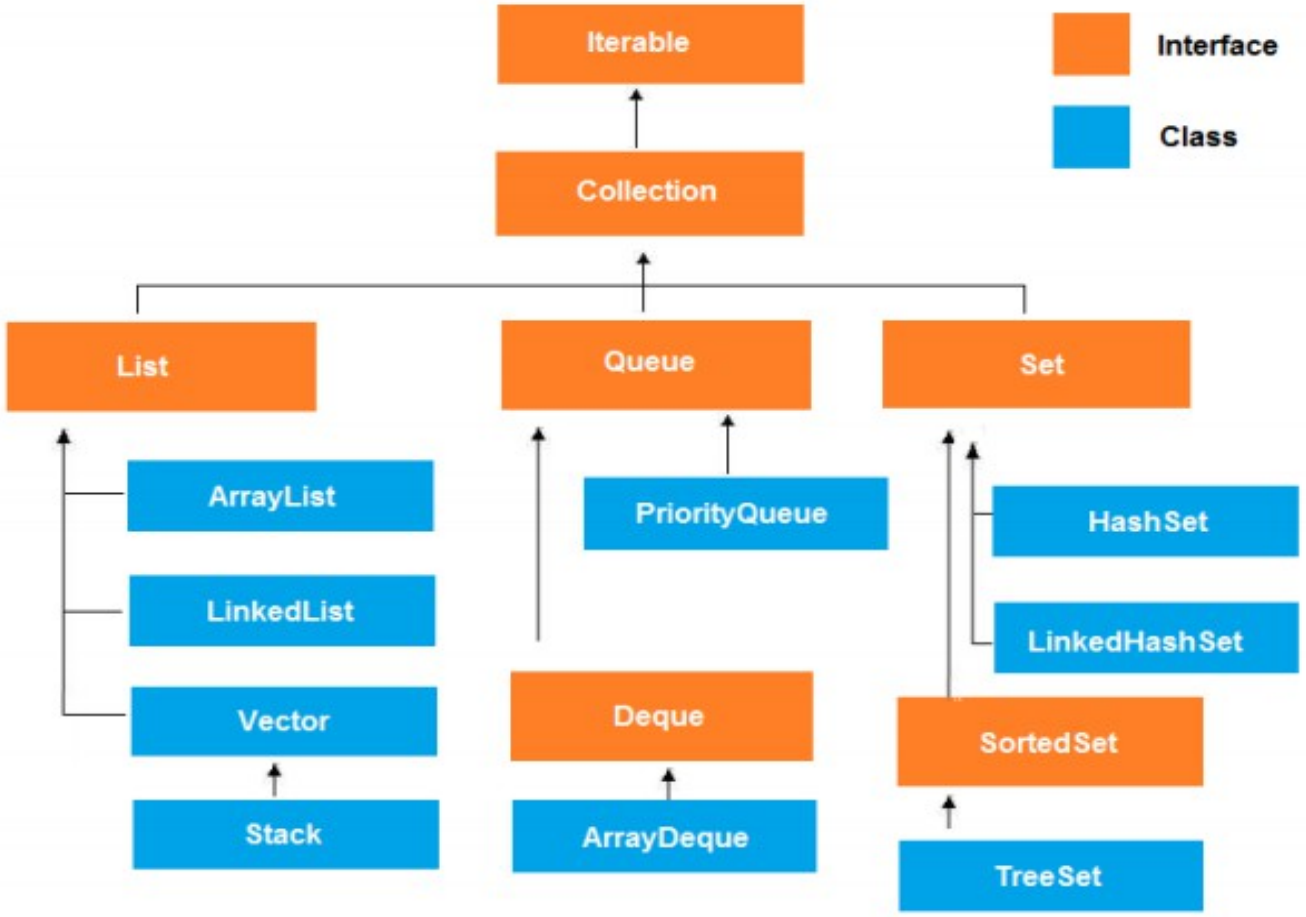
MONITOR E LOCK: CONFRONTI

- monitor, vantaggi
  - l'unità di sincronizzazione è il metodo: sincronizzazioni 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 e per method synchronization, può diminuire il livello di concorrenza
- lock esplicite, vantaggi:
  - maggior numero di funzioni disponibili, maggiore flessibilità
    - `tryLock()`
    - shared locks: multiple reader single writer
- lock esplicite, svantaggi: codice poco leggibile, se usate in modo non strutturato

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`
  - rivedere le implementazioni delle collezioni più importanti con lo scopo di utilizzare nel progetto le strutture dati più adeguate
- in questa lezione:
  - **`syncronized collections`**
- nella prossima lezione
  - **`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 THREAD SAFENESS

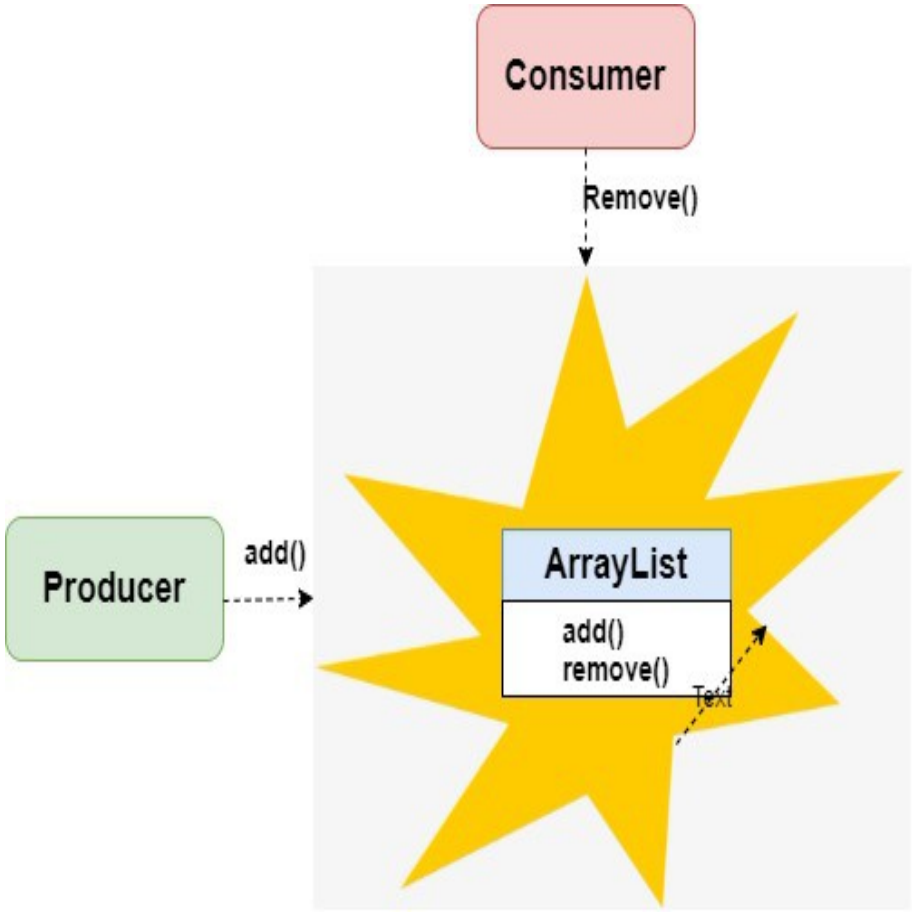
- in generale, si possono distinguere diversi tipi di collezioni riguardo alla therad safeness
  - collezioni thread safe, sincronizzate automaticamente da JAVA
    - `java.util.Vector`
    - `java.util.Hashtable`
  - collezioni che non offrono alcun supporto per il multithreading
    - `java.util.Map`
    - `Java.util,LinkedList`
    - `java.util.ArrayList`
  - synchronized collections
  - concurrent collections: introdotte in `java.util.concurrent`

COLLEZIONI NON THREAD SAFE

- `Arraylist`: se più thread accedono ad una stessa istanza ed almeno uno di essi la modifica, si possono creare inconsistenze

- add non atomica
  - determina quanti elementi ci sono nella lista
  - determina il punto esatto del nuovo elemento
  - incrementa il numero di elementi della lista

- analogamente per la `remove`



JAVA COLLECTIONS E THREAD SAFENESS

- `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
```

SYNCHRONIZED COLLECTIONS

- synchronized collection wrappers
  - "incapsulano" ogni metodo in uno blocco sincronizzato
- metodi definiti nella interfaccia `Collections`
  - trasformano una `Collection` non thread safe in una thread-safe
- utilizzano lock intrinseche gestite dalla JVM
- "conditionally thread safe" collections

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

SYNCHRONIZED COLLECTIONS

- synchronized collection wrapper
- utilizza un'unica "mutual exclusion lock"
  - se più thread tentono di accedere in modo concorrente
  - garantiscono un singolo thread alla volta sulla intera collezione

  - accessi concorrenti non consentiti
  - degradazione di perfomance: "high contention"

```
Map<Integer, String> mMap = new
HashMap<>();
    mMap =
        Collections.synchronizedMap(mMap);
    // Thread t1:
    mMap.put(1, "Italy");
    mMap.put(4, "France");
    mMap.put(7, "UK");
    mMap.put(12, "USA");
    mMap.put(13, "Sweden");
    mMap.put(18, "Norway");
    // Thread t2:
    String s1 = mMap.get(12);
    // Thread t3:
    String s2 = mMap.get(13);
    // Thread t4:
    String s3 = mMap.get(18);
```

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

COMPOSIZIONE NON THREAD SAFE DI OPERAZIONI

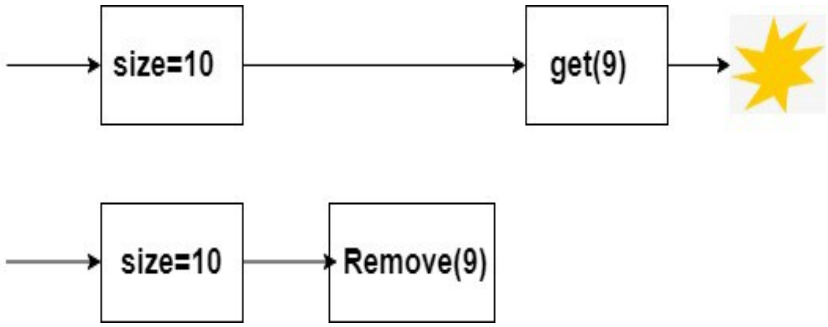
- la thread safety garantisce che le invocazioni delle singole operazioni della collezione siano thread-safe
- funzioni che coinvolgono più di una operazione possono non essere thread-safe

```
@NotThreadSafe
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); }
         }
```

COMPOSIZIONE NON THREAD SAFE DI OPERAZIONI

```
@NotThreadSafe
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); }
            }
```



- `Vector` è una collezione thread-safe
- tuttavia, in caso di accessi concorrenti, questo programma genera una

  `ArrayIndexOutOfBoundsException`

- per questo `Vector` è "conditionally thread safe"

COMPOSIZIONE NON THREAD SAFE DI OPERAZIONI

- soluzione: rendere atomiche sequenza di istruzioni con 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); }
         }}
```

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

- 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 da lock rientranti

CONCURRENT MODIFICATION EXCEPTION

- 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
            }
```

ASSIGNMENT

Risolvere il problema della simulazione del Laboratorio di informatica, assegnato nella lezione precedente, utilizzando il costrutto di Monitor