

Reti e Laboratorio III

Modulo Laboratorio III

AA. 2022-2023

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 3

**Callable, Scheduled Tasks,
Monitor**

29/9/2022

THREAD CHE RESTITUISCONO RISULTATI

- un oggetto di tipo Runnable
 - incapsula un'attività che viene eseguita in modo asincrono
 - il metodo run è un metodo asincrono, senza parametri e che non restituisce un valore di ritorno
- Interface Callable: consente di definire un task che può restituire un risultato e sollevare eccezioni
 - come “accedere” al risultato, in modo asincrono?
 - Future interface: contiene metodi per reperire, in modo asincrono, il risultato di una computazione asincrona. cioè
 - per controllare se la computazione è terminata
 - per attendere la terminazione di una computazione (eventualmente per un tempo limitato)
 - per cancellare una computazione,
- la classe FutureTask fornisce una implementazione della interfaccia Future.

L'INTERFACCIA CALLABLE

```
public interface Callable <V>
```

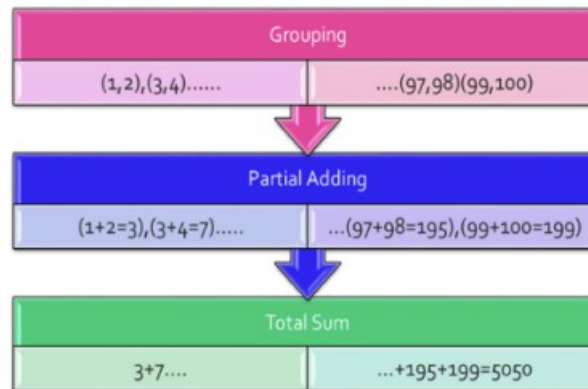
```
{ V call() throws Exception;}
```

- contiene il solo metodo `call()`, analogo al metodo `run()` dell'interfaccia `Runnable`
- il codice del task è implementato nel metodo `call()`
- a differenza del metodo `run()`, il metodo `call()` può
 - restituire un valore
 - sollevare eccezioni
- il parametro di tipo `<V>` indica il tipo del valore restituito
 - `Callable <Integer>` rappresenta una elaborazione asincrona che restituisce un valore di tipo `Integer`

DIVIDE ET IMPERA CON MULTITHREADING

calcolare la somma di tutti i numeri da 1 a n

- soluzione sequenziale: loop che itera da 1 a 100 e calcola la somma
- seguendo il pattern divide and conquer :
 - individuare sottointervalli dell'intervallo 1-100
 - creare un task diverso per ogni intervallo: calcola la somma per quell'intervallo
 - sottomettere i task ad un threadpool, somme parziali in parallelo
 - raccogliere le somme parziali per calcolare la somma totale.



THREAD CHE RESTITUISCONO RISULTATI

```
import java.util.concurrent.Callable;

public class Calculator implements Callable<Integer> {

    private int a;

    private int b;

    public Calculator(int a, int b) {

        this.a = a;

        this.b = b;

    }

    public Integer call() throws Exception {

        Thread.sleep((long)(Math.random() * 15000));

        return a + b;

    }

}
```

THREAD CHE RESTITUISCONO RISULTATI

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

public class Adder {

    public static void main(String[] args) throws ExecutionException,InterruptedException{

        // Create thread pool using Executor Framework

        ExecutorService executor = Executors.newFixedThreadPool(5);

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

        for (int i = 0; i < 10; i=i+2) { // Create new Calculator object

            Calculator c = new Calculator(i, i + 1);

            list.add(executor.submit(c));}

        int s=0;

        for (Future<Integer> f : list) {

            try { System.out.println(f.get());

                s=s+f.get();

            } catch (Exception e) {};}

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

        executor.shutdown(); }}
```

THREAD CHE RESTITUISCONO RISULTATI

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

public class Adder {

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

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

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

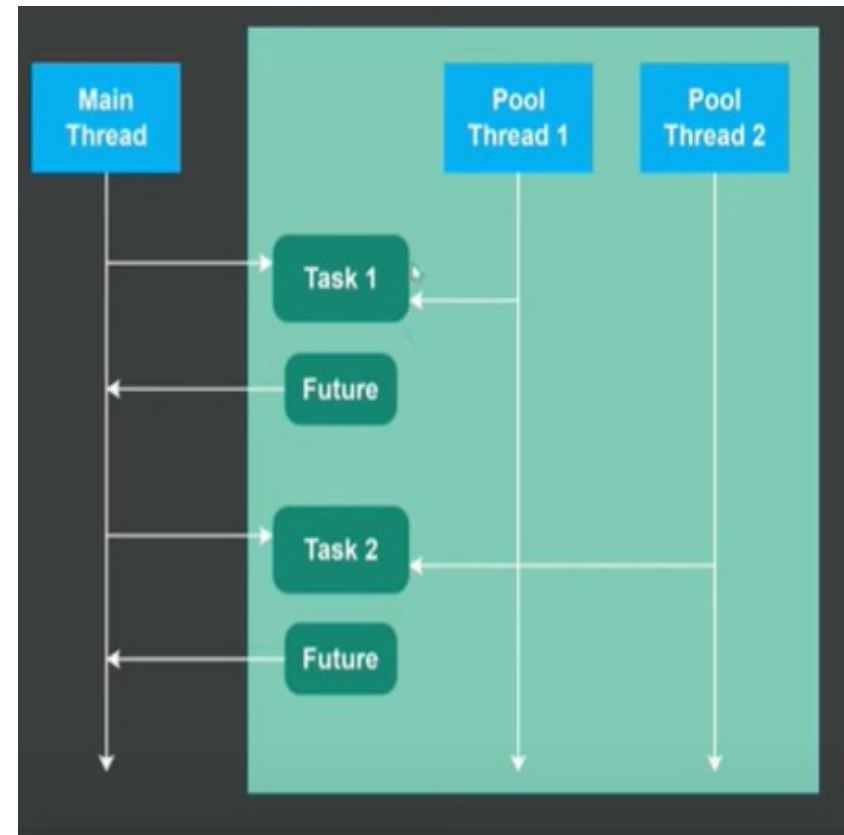
        int s=0;

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

        System.out.println("la somma e '" +s); executor.shutdown();}}
```

L'INTERFACCIA FUTURE

- sottomettere direttamente l'oggetto di tipo Callable al pool mediante il metodo **submit**
- la sottomissione restituisce un oggetto di tipo **Future**
- ogni oggetto Future è associato ad uno dei task sottomessi al ThreadPool
- è possibile applicare diversi metodi all'oggetto **Future**



L'INTERFACCIA FUTURE

```
public interface Future <V>
{
    V get( ) throws...;
    V get (long timeout, TimeUnit) throws...;
    void cancel (boolean mayInterrupt);
    boolean isCancelled( );
    boolean isDone( ); }

```

- metodo get()
 - si blocca fino a che il thread non ha prodotto il valore richiesto e restituisce il valore calcolato
- metodo get (long timeout, TimeUnit)
 - definisce un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una `TimeoutException`
- è possibile cancellare il task e verificare se la computazione è terminata oppure è stata cancellata

ALTRI TIPI DI THREAD POOL

- **Single Threaded Executor**
 - un singolo thread
 - equivalente ad invocare un `FixedThreadPool` di dimensione 1
 - utilizzo: assicurare che i task vengano eseguiti nell'ordine con cui si trovano in coda (sequenzialmente)
- **Scheduled Thread Pool**
 - distanziare esecuzione dei task con un certo delay
 - task periodici

SCHEDULED EXECUTOR SERVICE

- l'interfaccia `ScheduledExecutorService` dà la possibilità di schedulare un task
 - dopo un certo periodo di tempo (delay)
 - periodicamente
- `schedule(Runnable command, long delay, TimeUnit unit)`
 - esegue un task `Runnable` (o `Callable`) dopo un certo intervallo di tempo
- `scheduleAtFixedRate(Runnable command, long initialDelay, long delay, TimeUnit unit)`
 - esegue un task dopo un intervallo iniziale, poi lo ripete periodicamente.
 - se il tempo di esecuzione del task è maggiore del periodo specificato, le sue seguenti esecuzioni possono essere ritardate.
- `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`
 - esegue un task dopo un intervallo iniziale, poi lo ripete periodicamente con un intervallo dato tra la terminazione di una esecuzione e l'inizio della successiva

UN BEEP PERIODICO.....

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

import java.awt.*;

public class BeepClockS implements Runnable {

    public void run() {

        Toolkit.getDefaultToolkit().beep();

    }

    public static void main(String[] args) {

        ScheduledExecutorService scheduler

            = Executors.newSingleThreadScheduledExecutor();

        Runnable task = new BeepClockS();

        int initialDelay = 4;

        int periodicDelay = 2;

        scheduler.scheduleAtFixedRate(task, initialDelay, periodicDelay,

                                         TimeUnit.SECONDS);}}}
```

UN TASK “COUNTDOWN”

```
public class CountDownClock implements Runnable {  
    private String clockName;  
    public CountDownClock(String clockName) {  
        this.clockName = clockName;  
    }  
    public void run() {  
        String threadName = Thread.currentThread().getName();  
        for (int i = 5; i >= 0; i--) {  
            System.out.printf("%s -> %s: %d\n", threadName, clockName, i);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

COUNTDOWN SCAGLIONATI NEL TEMPO

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

public class ConcurrentScheduledTaskExample {

    public static void main(String[] args) {

        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(3);

        CountDownClock clock1 = new CountDownClock("A");
        CountDownClock clock2 = new CountDownClock("B");
        CountDownClock clock3 = new CountDownClock("C");

        scheduler.scheduleWithFixedDelay(clock1, 3, 10, TimeUnit.SECONDS);
        scheduler.scheduleWithFixedDelay(clock2, 3, 15, TimeUnit.SECONDS);
        scheduler.scheduleWithFixedDelay(clock3, 3, 20, TimeUnit.SECONDS);

    }

}
```

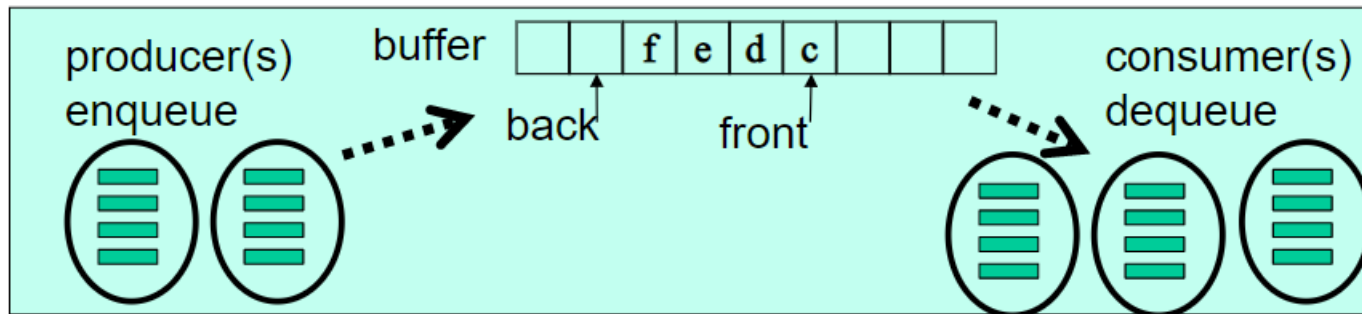
CONDIVIDERE RISORSE TRA THREADS

- un insieme di thread vogliono condividere una risorsa.
 - più thread accedono concorrentemente allo stesso file, alla stessa parte di un database o di una struttura di memoria
- l'accesso non controllato a risorse condivise può provocare situazioni di errore ed inconsistenze.
 - **race conditions**
- **sezione critica**: blocco di codice a cui si effettua l'accesso ad una risorsa condivisa e che deve essere eseguito da un thread per volta
- necessario implementare **classi thread safe**
 - il codice dei metodi della classe può essere utilizzato/condiviso in un ambiente concorrente senza provocare inconsistenze/comportamenti inaspettati

ALTERNATIVE PER DEFINIRE CLASSI THREAD SAFE

- alternative per definire classi thread safe: usare
 - classi thread safe predefinite
 - concurrent-aware interfaces
 - Interfaces: Blocking Queue, TransferQueue, Blocking Dequeue, ConcurrentMap, ConcurrentNavigable Map
 - concurrent-aware classes
 - LinkedBlockingQueue
 - ArrayBlockingQueue
 - PriorityBlockingQueue
 - DelayQueue
 - SynchronousQueue
 - CopyOnWriteArrayList
 - CopyOnWriteArraySet
 - ConcurrentHahsMap
 - i monitor
 - le lock a basso livello (non le vedremo)

IL PROBLEMA DEL PRODUTTORE CONSUMATORE



- un classico problema che descrive due (o più thread) che condividono un buffer, di dimensione fissata, usato come una coda
 - il produttore P produce un nuovo valore, lo inserisce nel buffer e torna a produrre valori
 - il consumatore C consuma il valore (lo rimuove dal buffer) e torna a richiedere valori
 - garantire che il produttore non provi ad aggiungere un dato nelle coda se è piena ed il consumatore non provi a rimuovere un dato da una coda vuota
- generalizzazione per più produttori e più consumatori

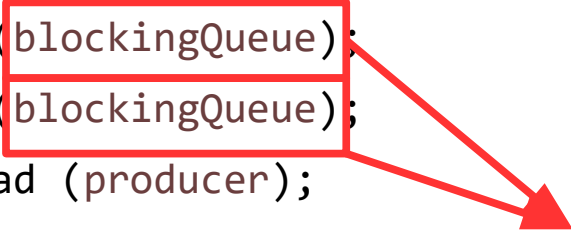
PRODUTTORE CONSUMATORE: SINCRONIZZAZIONE

- l'interazione esplicita tra threads avviene in JAVA mediante l'utilizzo di **oggetti condivisi**
 - la **coda** che memorizza i messaggi scambiati tra P e C è condivisa
- necessari costrutti per **sospendere** un thread T quando **una condizione** non è verificata e **riattivare** T quando diventa vera
 - il produttore si sospende se la coda è piena
 - si riattiva quando c'è una posizione libera
- due tipi di sincronizzazione:
 - **implicita**: la mutua esclusione sull'oggetto condiviso è garantita dall'uso di lock (implicite o esplicite)
 - **esplicita**: occorrono altri meccanismi

PRODUTTORE/CONSUMATORE CON BLOCKINGQUEUES

```
import java.util.concurrent.BlockingQueue;  
import java.util.concurrent.ArrayBlockingQueue;  
public class ProducerConsumerExample {
```

```
    public static void main(String[] args) {  
        BlockingQueue<String> blockingQueue =  
            new ArrayBlockingQueue<String>(3);  
        Producer producer = new Producer(blockingQueue);  
        Consumer consumer = new Consumer(blockingQueue);  
        Thread producerThread = new Thread (producer);  
        Thread consumerThread = new Thread(consumer);  
        producerThread.start();  
        consumerThread.start(); } }
```



il riferimento alla
struttura dati condivisa si
passa ad entrambi i thread

```
import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {
    BlockingQueue <String> blockingQueue = null;
    public Producer (BlockingQueue<String> queue) {
        this.blockingQueue = queue;    }
    public void run() {
        while (true) {
            long timeMillis = System.currentTimeMillis();
            try {
                this.blockingQueue.put("" + timeMillis);
            } catch (InterruptedException e) {
                System.out.println("Producer was interrupted"); }
            sleep(1000); }}
    private static void sleep(long timeMillis) {
        try { Thread.sleep(timeMillis);
        } catch (InterruptedException e) {e.printStackTrace()}} }
```

```
import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {
    BlockingQueue<String> blockingQueue = null;
    public Consumer (BlockingQueue <String> queue) {
        this.blockingQueue = queue; }
    public void run() {
        while (true) {
            try {
                String element =
                    this.blockingQueue.take();
                System.out.println("consumed: "+ element);
            } catch (InterruptedException e) {e.printStackTrace();}
        }
    }
}
```

IL MONITOR

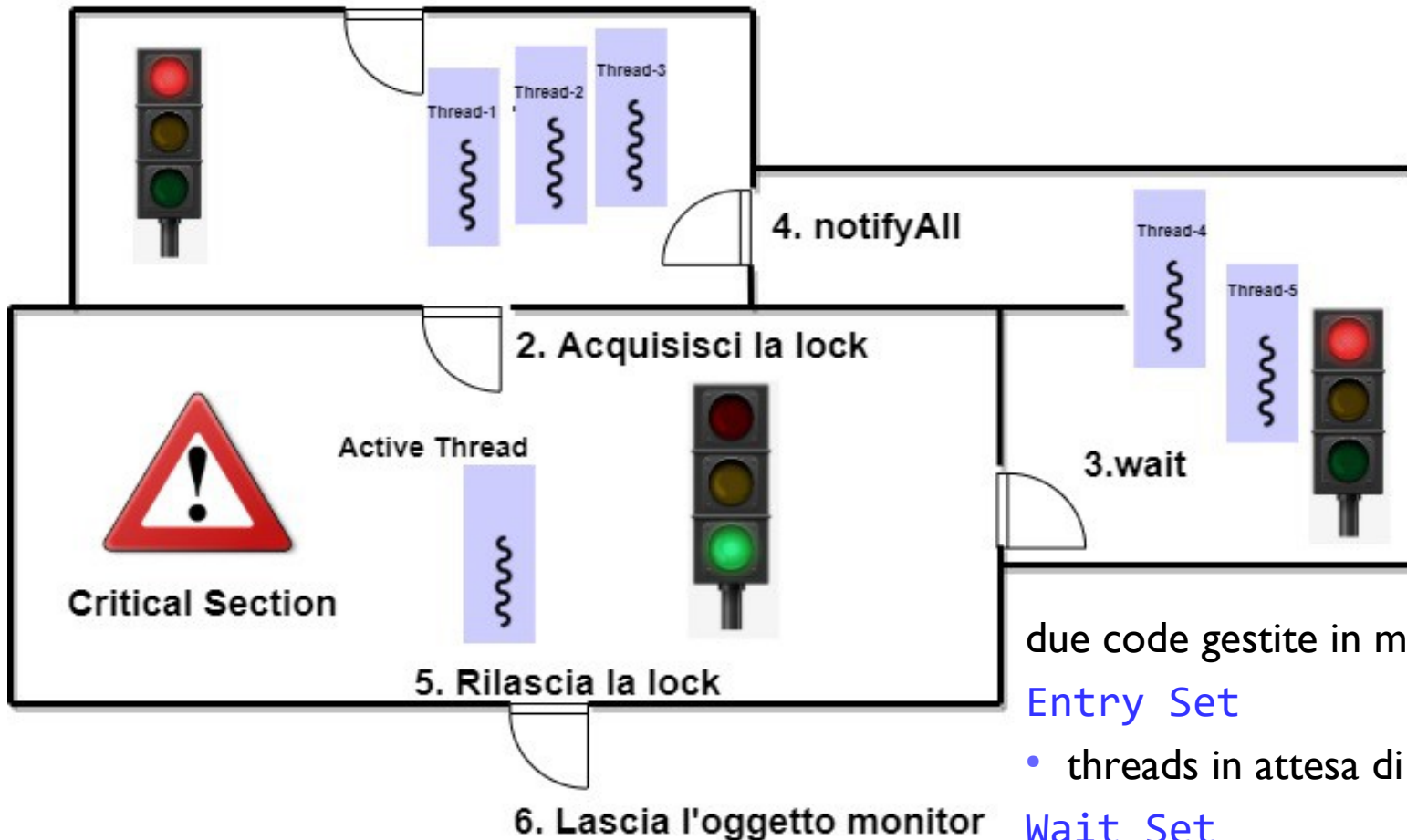
- meccanismo linguistico ad alto livello per la sincronizzazione
 - idea introdotta negli anni '70 safe (*Per Brinch Hansen, Hoare 1974*)
- incapsula un **oggetto condiviso** e le operazioni che vengono invocate dai threads su di esso, in modo concorrente
- funzionalità offerte dal monitor
 - **mutua 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 “wait queue” gestita dalla JVM
 - `wait`
 - `notify/notifyAll`

UN'OCCHIATA ALL'INTERNO DI UN MONITOR

1. Entra nell'oggetto 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`
- 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 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
 - 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 sollevata 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

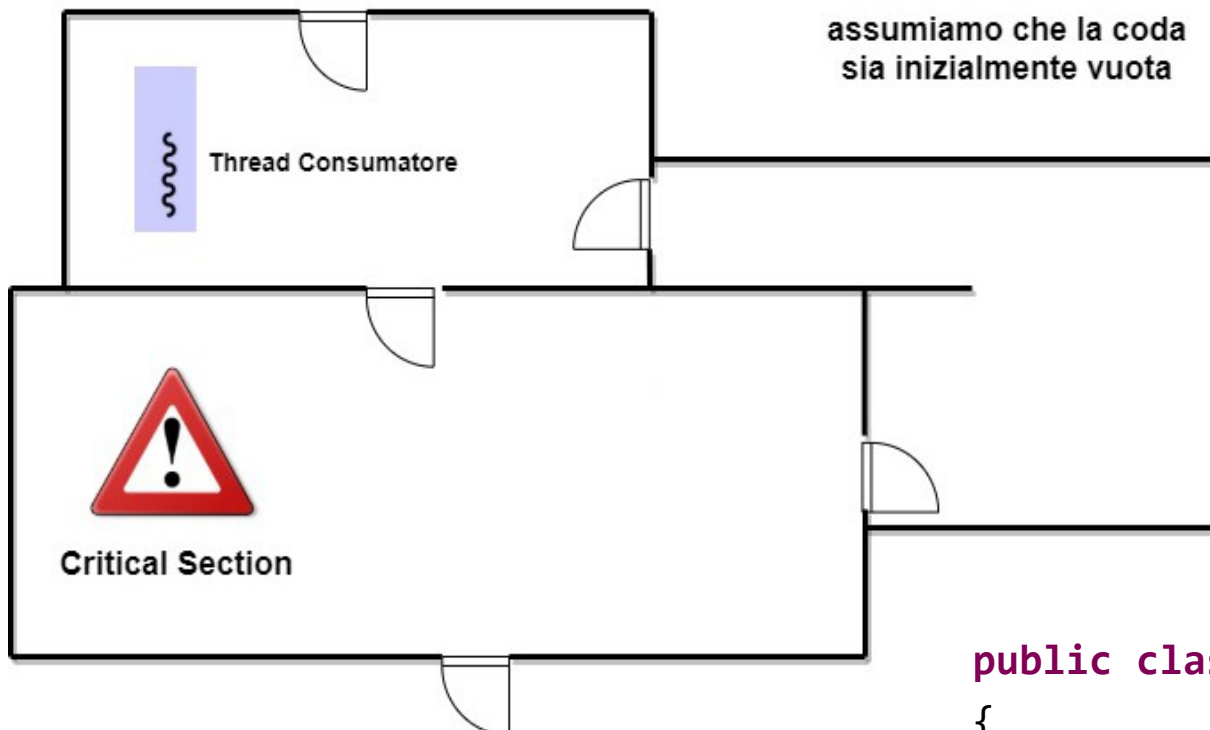
```
public class MessageQueue {
    int putptr, takeptr, count;
    final Object[] items;
    public MessageQueue(int size){
        items = new Object[size];
        count=0;putptr=0;takeptr=0;}
    public synchronized void produce(Object x)
    { while (count == items.length)
        try {
            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;  
}}
```

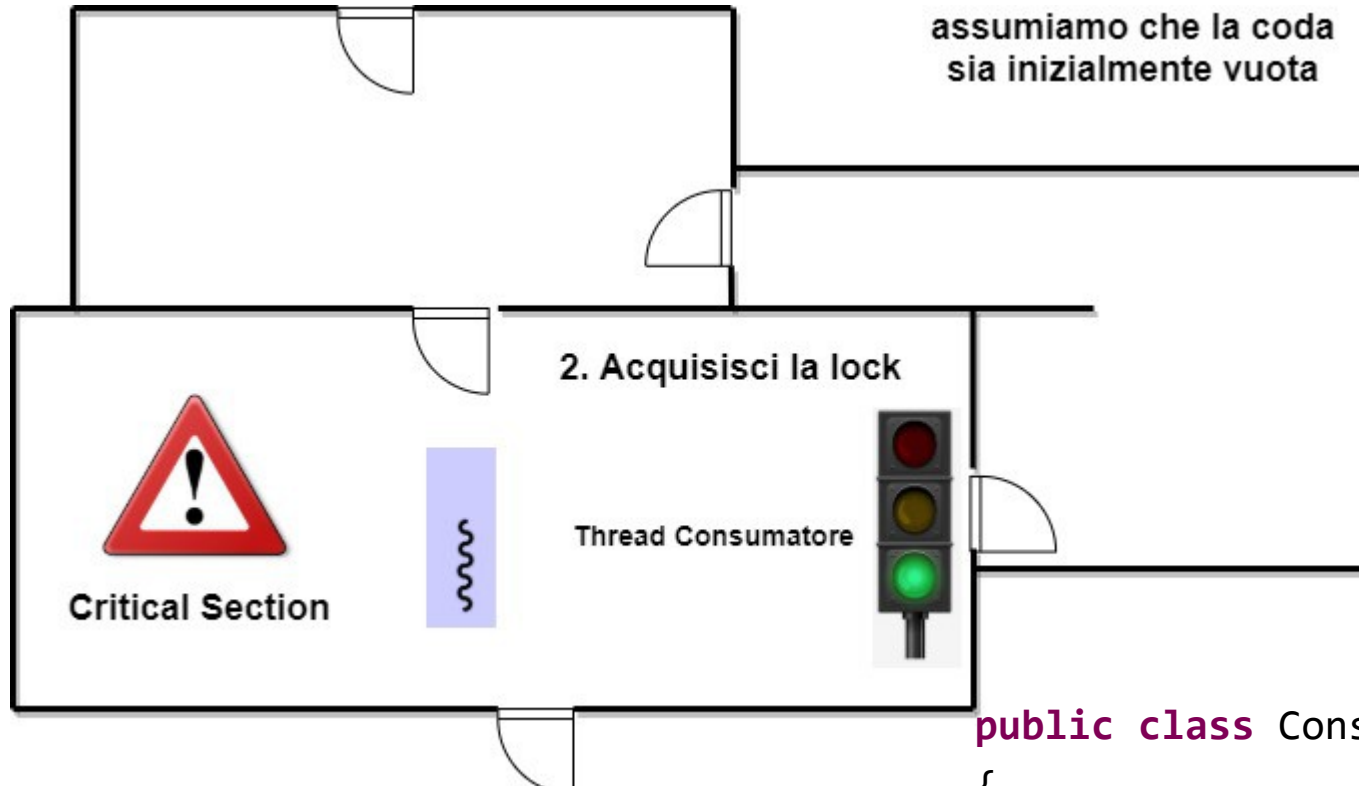
PRODUTTORE CONSUMATORE “ILLUSTRATO”

1. Entra nell'oggetto monitor



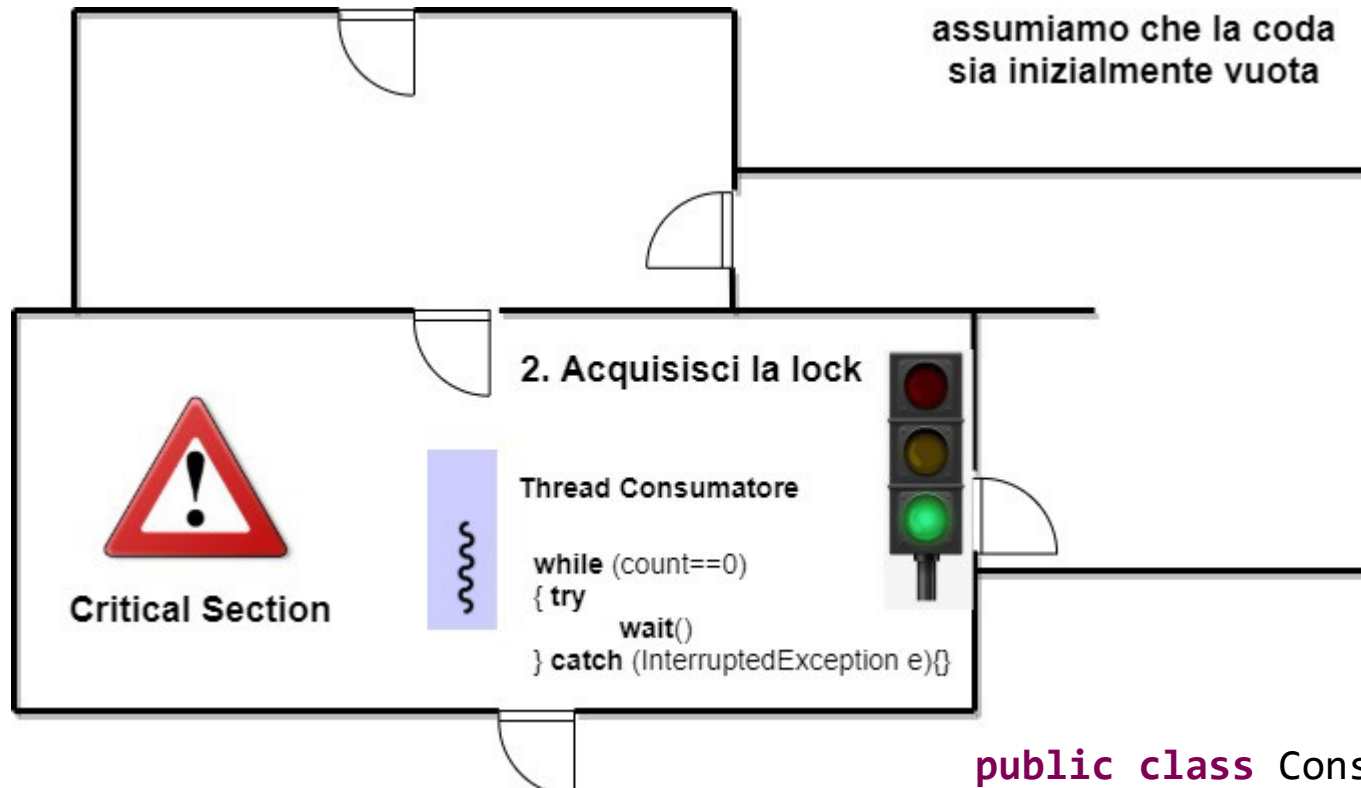
```
public class Consumer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++){
            Object o=queue.consume();
            ...}
    }
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



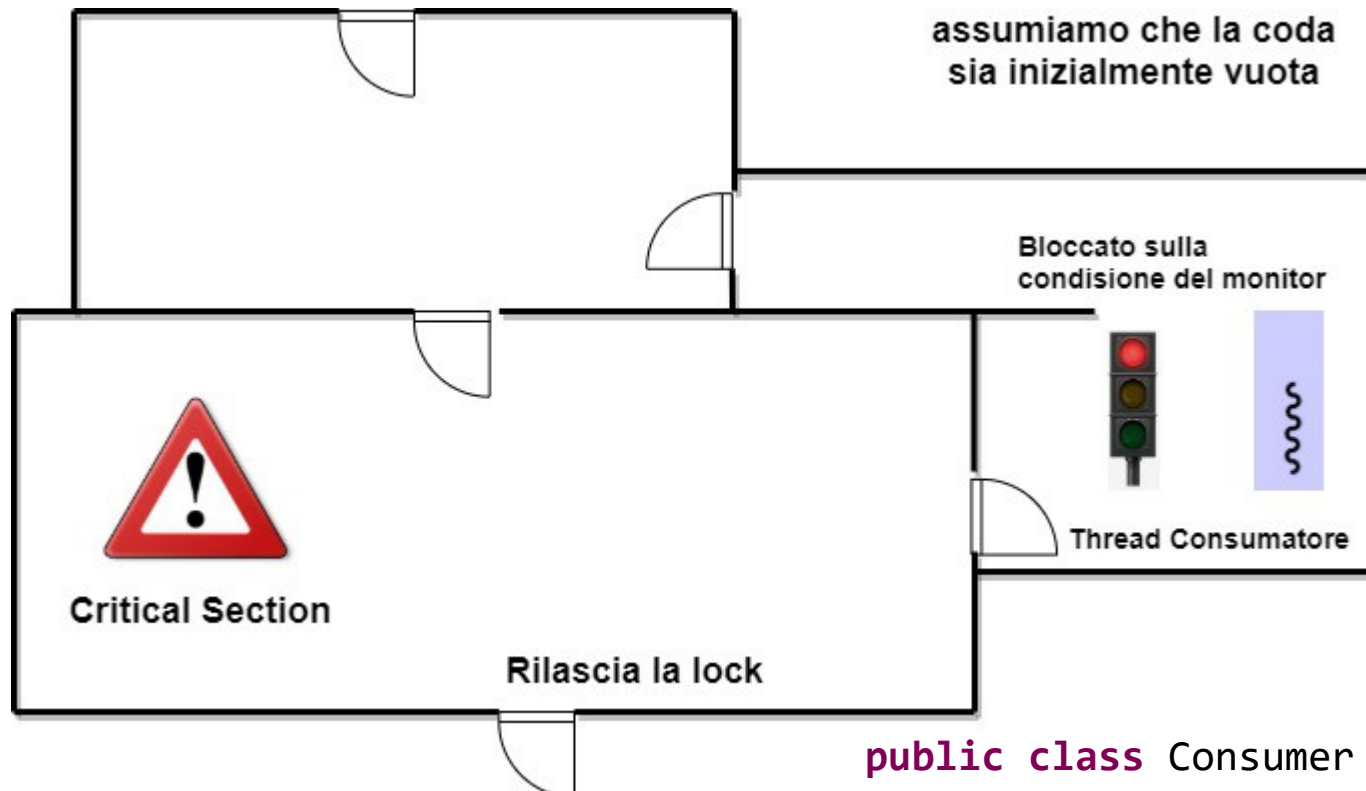
```
public class Consumer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++){
            Object o=queue.consume();
            ....}
    }
}
```


PRODUTTORE CONSUMATORE “ILLUSTRATO”



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

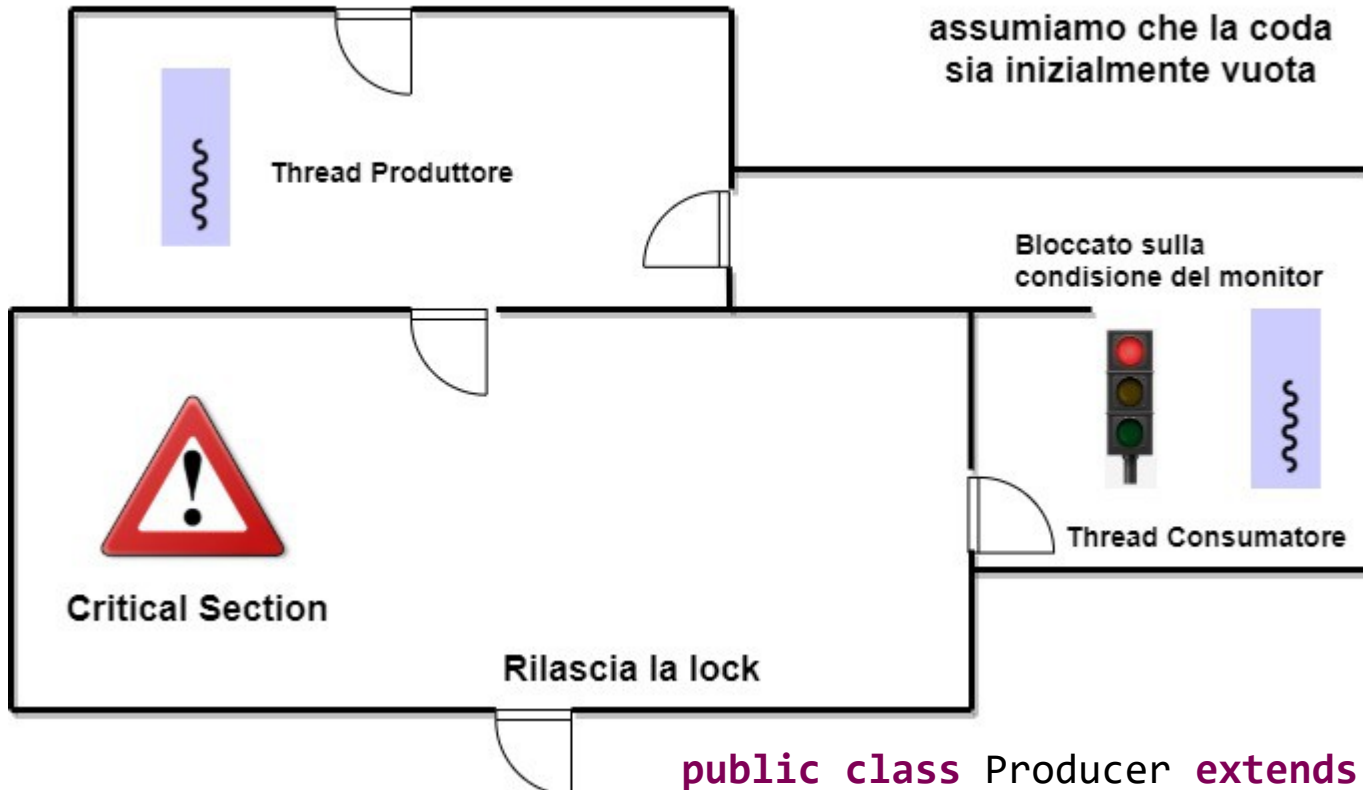
PRODUTTORE CONSUMATORE “ILLUSTRATO”



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

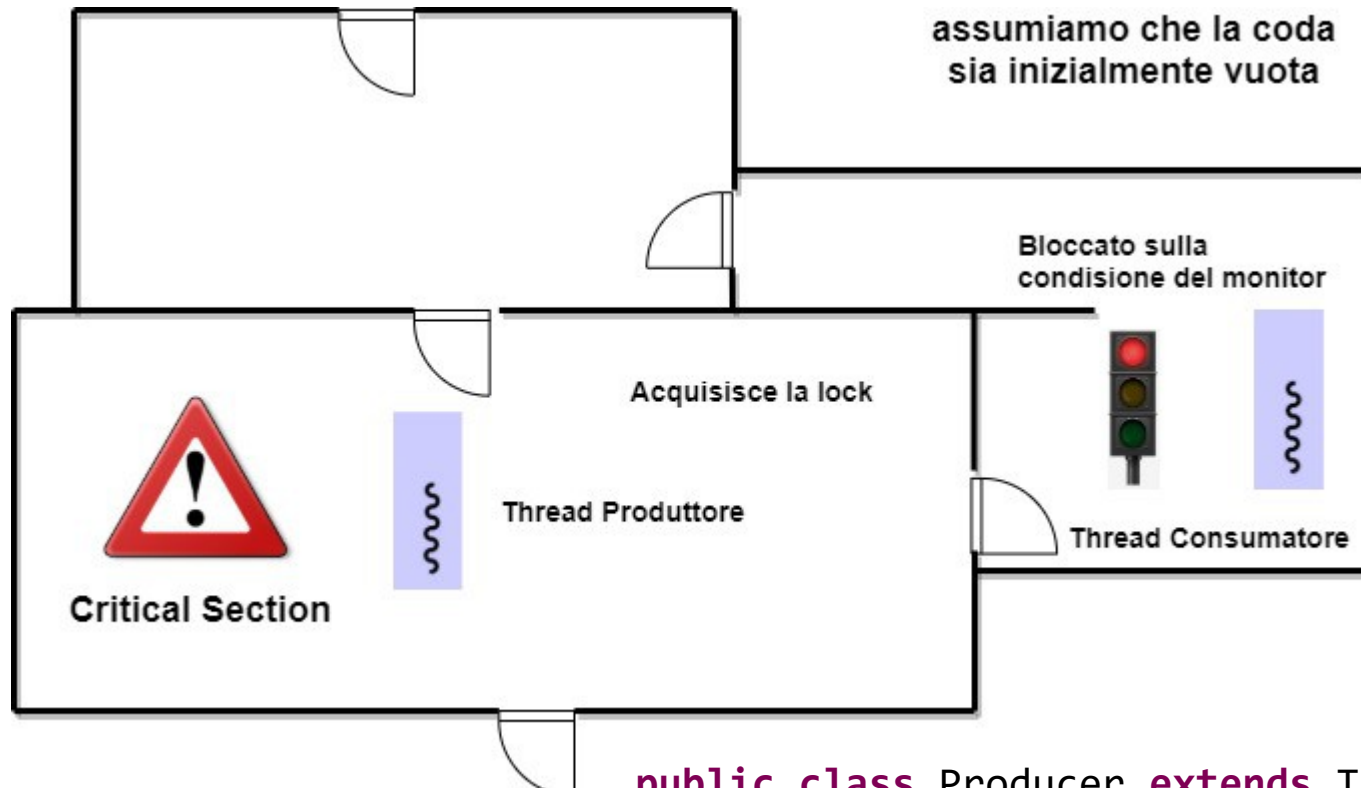
PRODUTTORE CONSUMATORE “ILLUSTRATO”

1. Entra nell'oggetto monitor



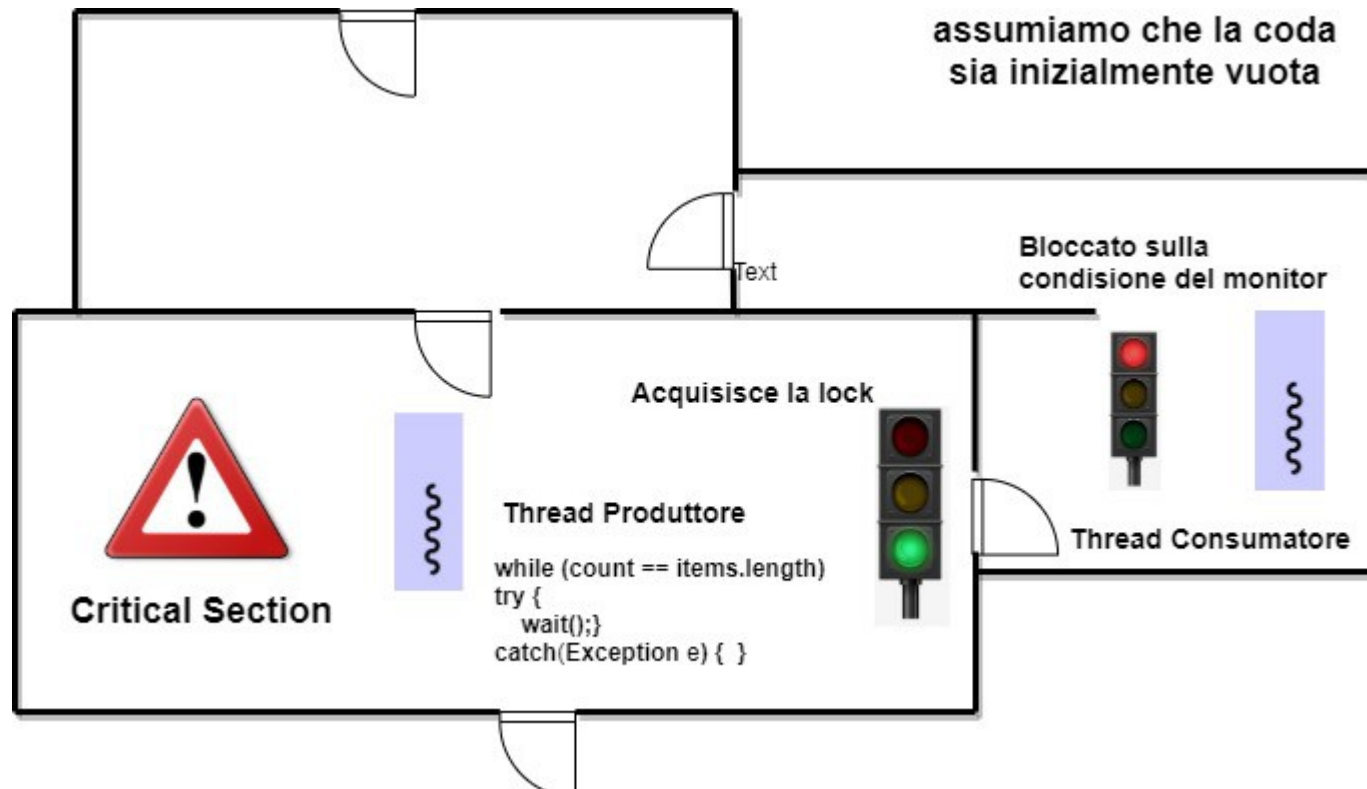
```
public class Producer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++)
        {queue.produce("MSG#" + count + Thread.currentThread());}
        ....
    }
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



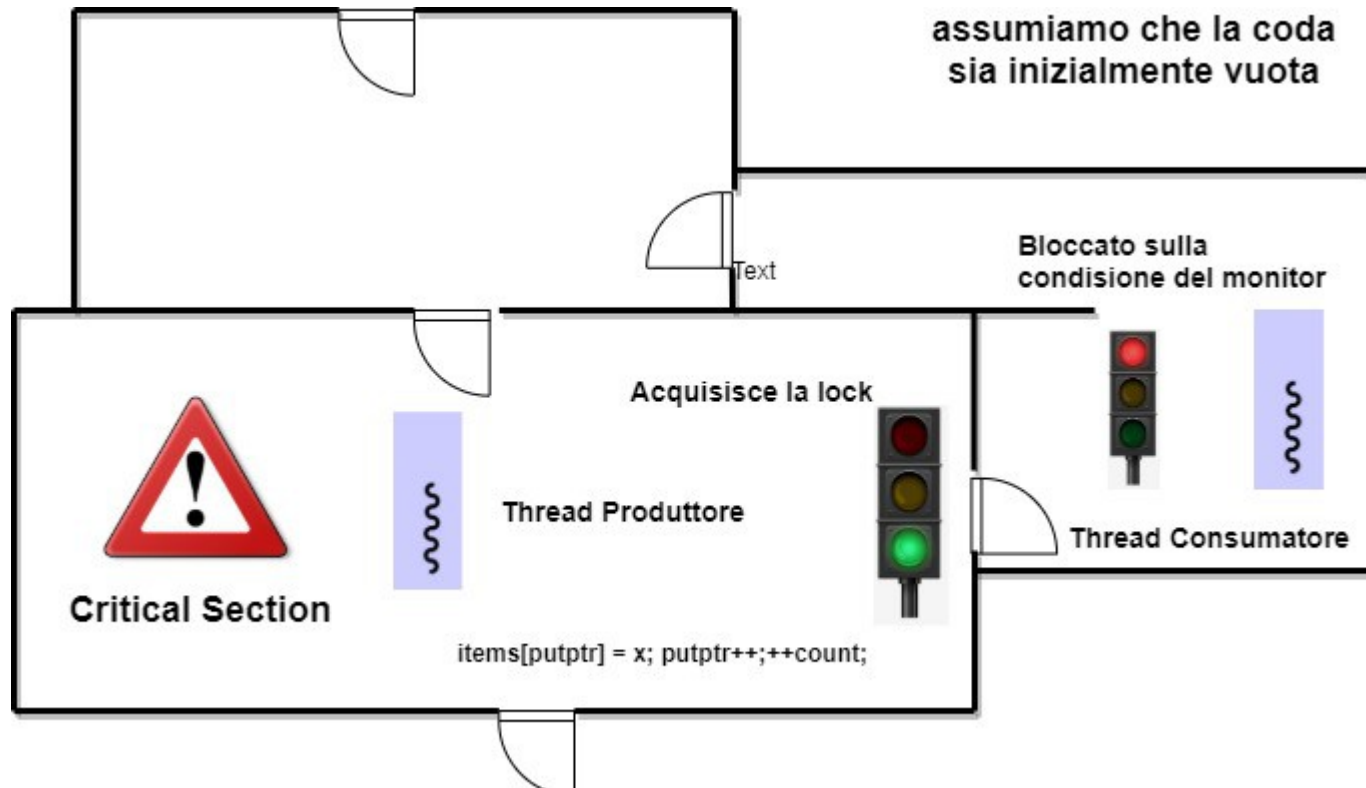
```
public class Producer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++)
        {queue.produce("MSG#"+count+Thread.currentThread());}
        ....
    }
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



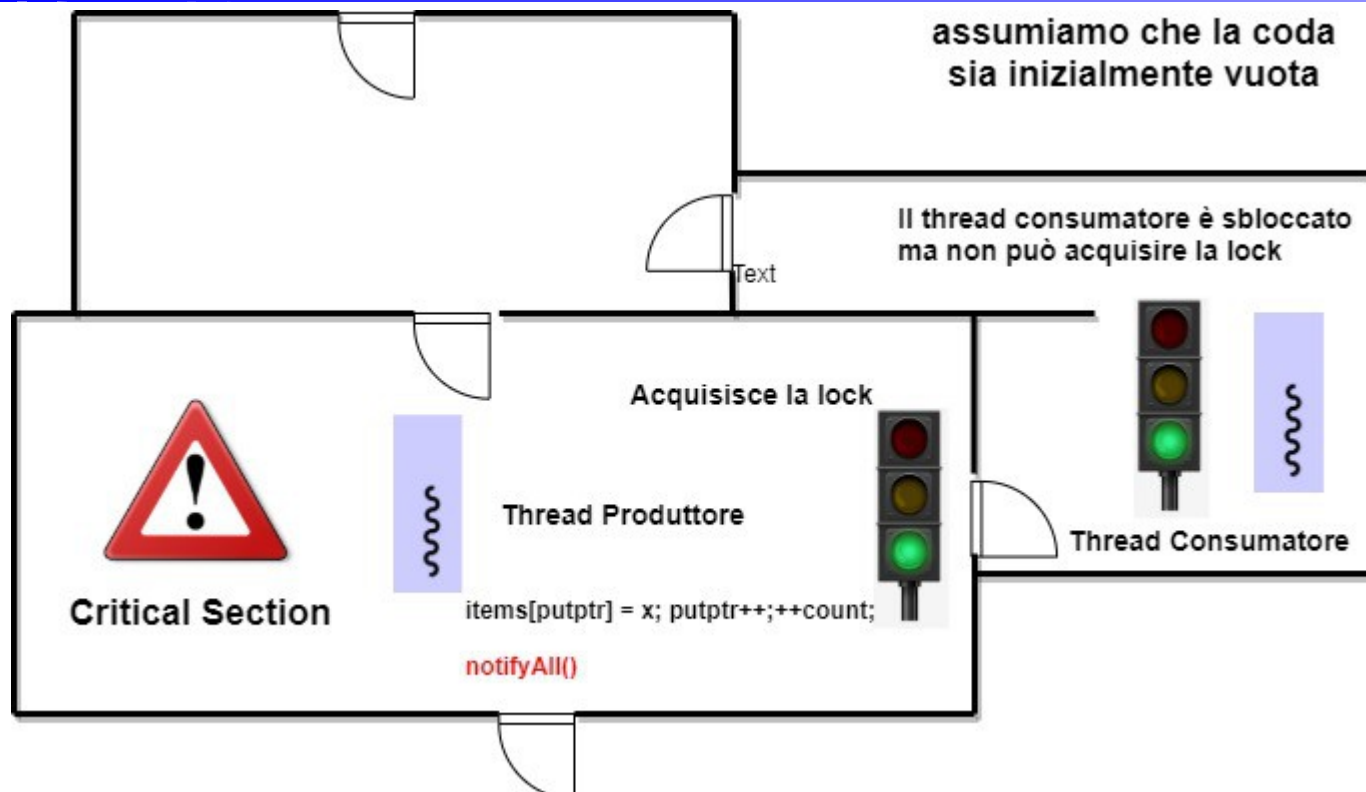
```
public class Producer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++)
        {queue.produce("MSG#" + count + Thread.currentThread())}
        ....
    }
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



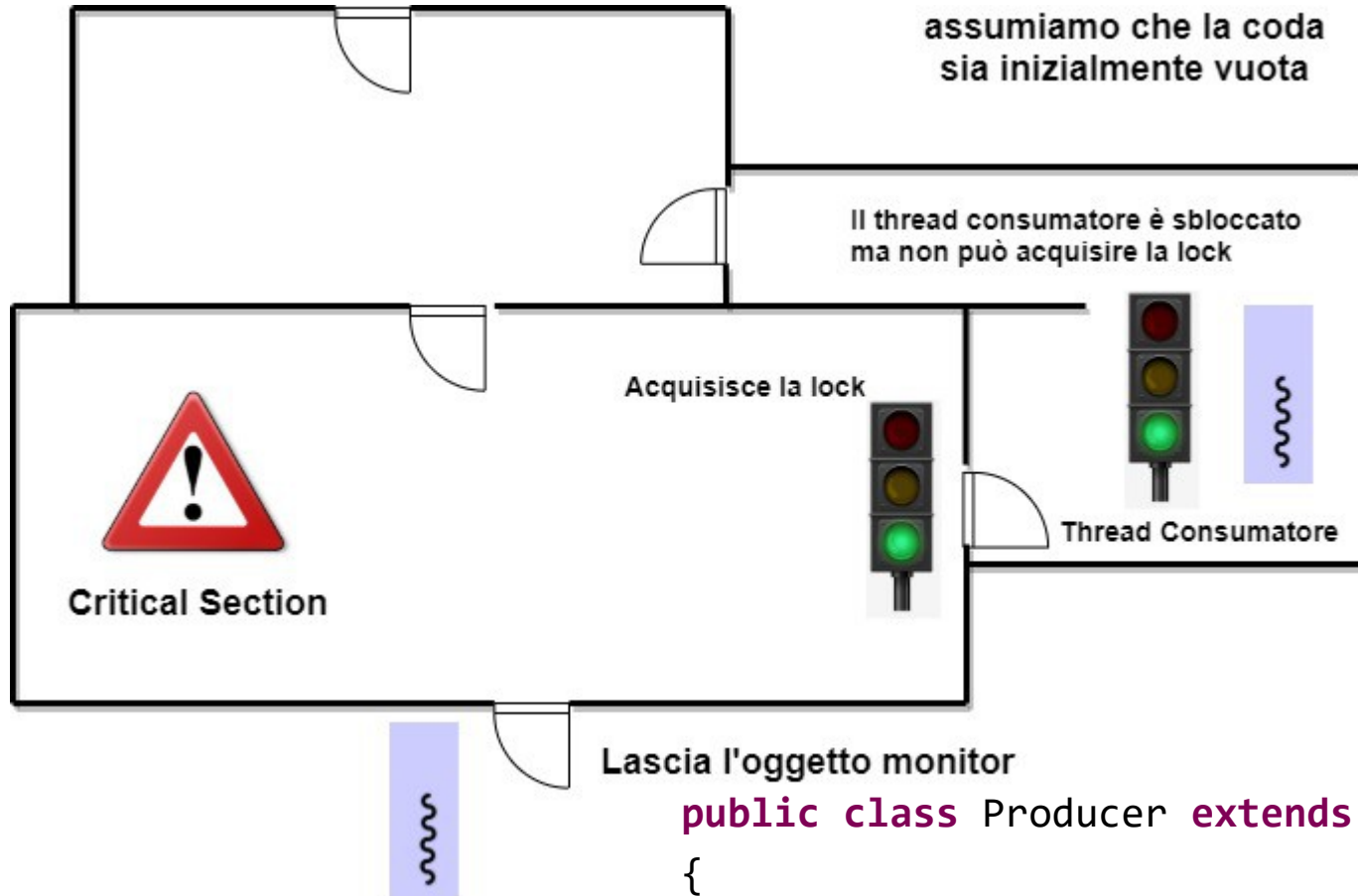
```
public class Producer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++)
        {queue.produce("MSG#" + count + Thread.currentThread().getName() + " ");
          ....
        }
    }
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



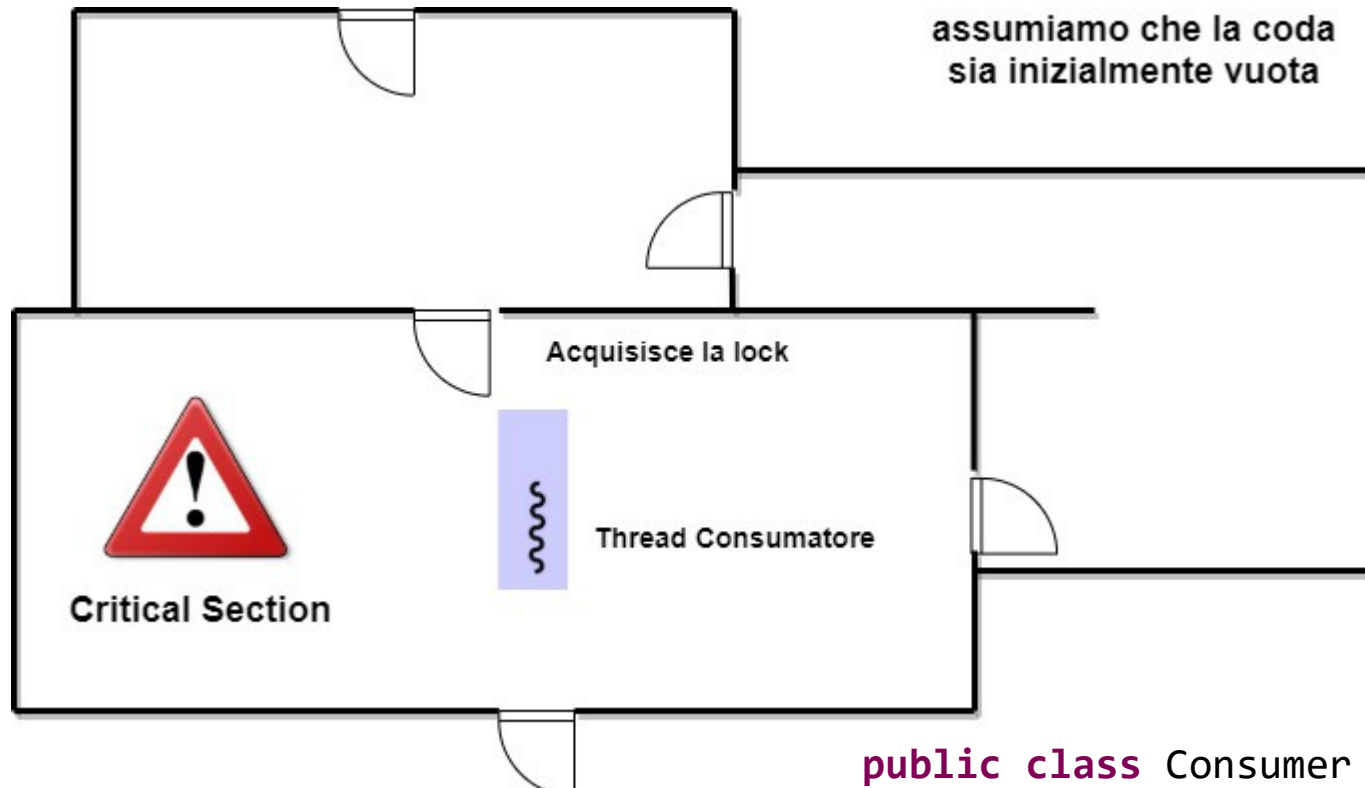
```
public class Producer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++)
        {queue.produce("MSG#" + count + Thread.currentThread().getName());
        ....
    }
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



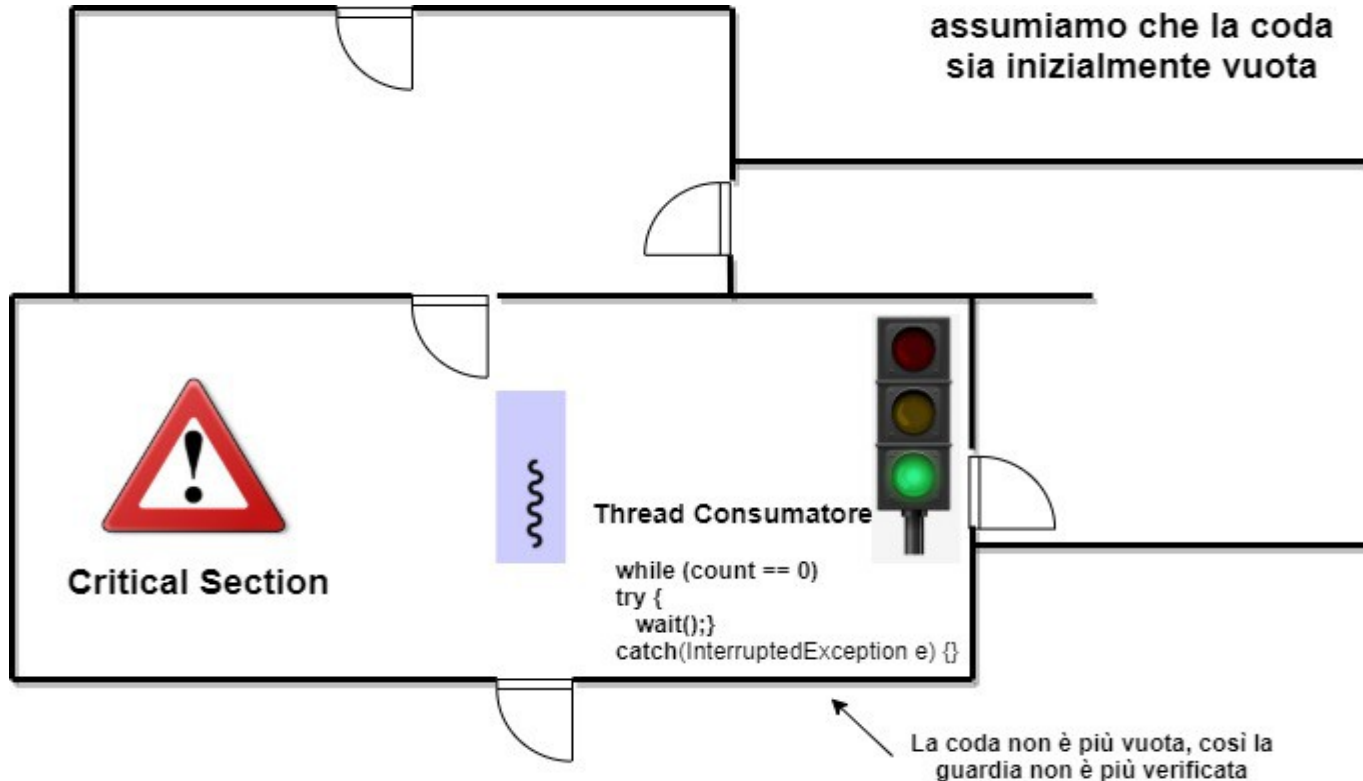
```
public class Producer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++)
        {queue.produce("MSG#" + count + Thread.currentThread())}
        ....
    }
}
```


PRODUTTORE CONSUMATORE “ILLUSTRATO”



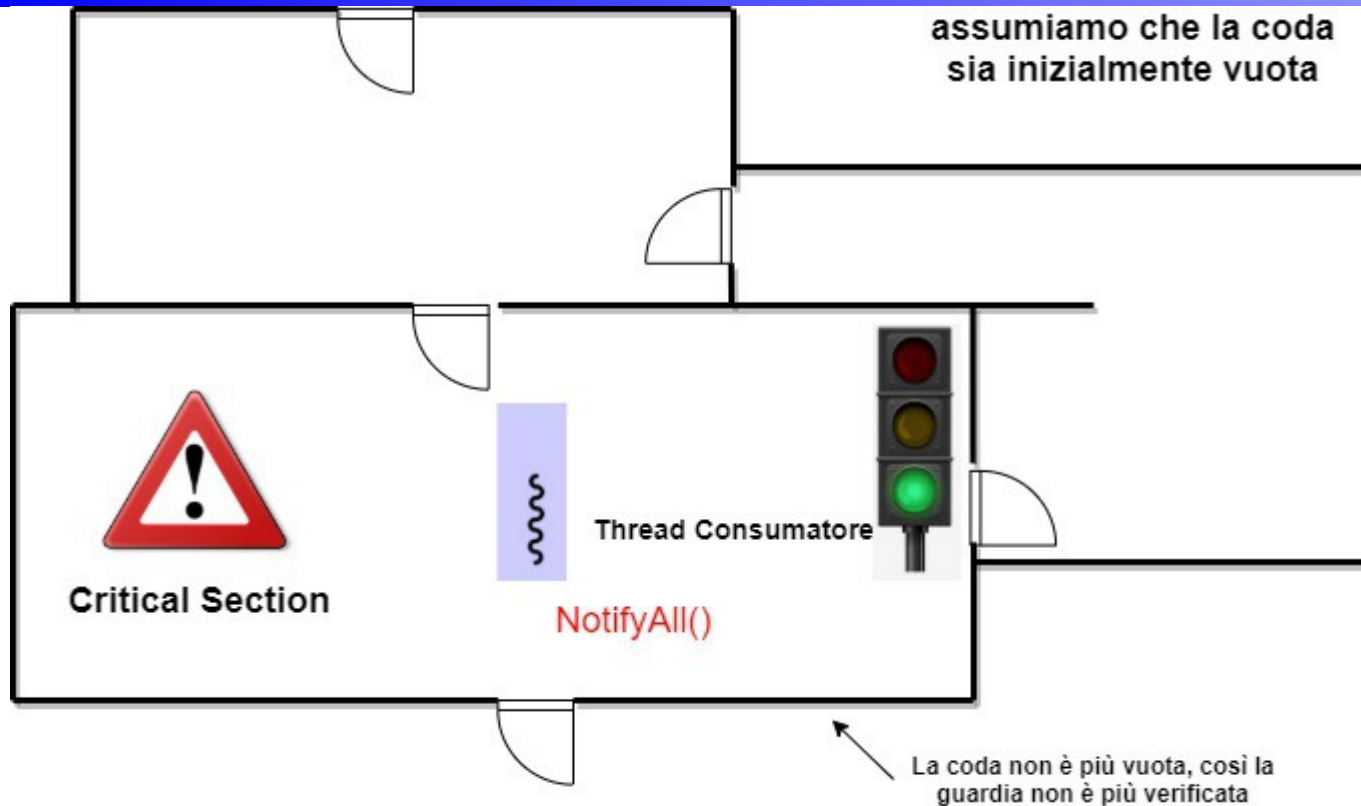
```
public class Consumer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++){
            Object o=queue.consume();
            ...}
    }
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



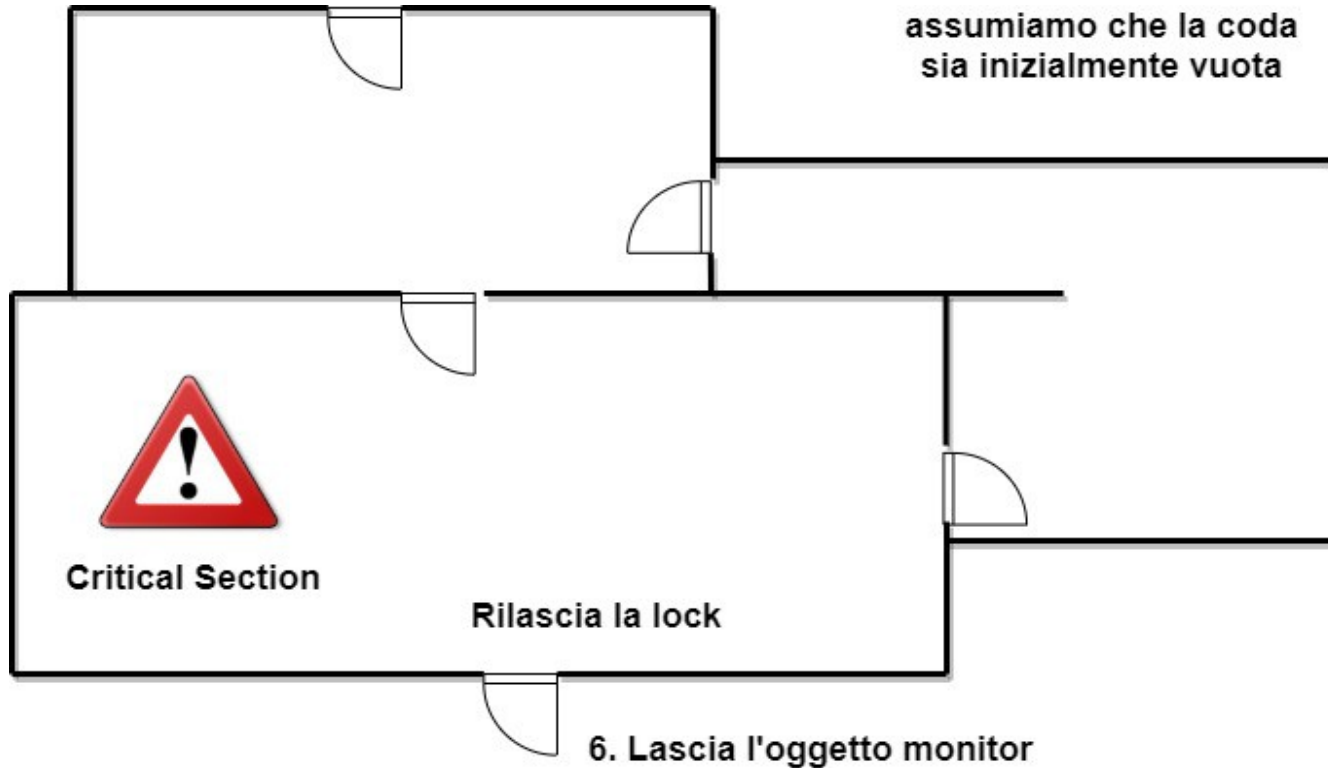
```
public class Consumer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++){
            Object o=queue.consume();
            ...}
    }
}
```

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();
            ...}
}
```

ASSIGNMENT 3: GESTIONE LABORATORIO

Il laboratorio di Informatica del Polo Marzotto è utilizzato da tre tipi di utenti, studenti, tesisti e professori ed ogni utente deve fare una richiesta al tutor per accedere al laboratorio. I computers del laboratorio sono numerati da 1 a 20. Le richieste di accesso sono diverse a seconda del tipo dell'utente:

- a) i professori accedono in modo esclusivo a tutto il laboratorio, poichè hanno necessità di utilizzare tutti i computers per effettuare prove in rete.
- b) i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice *i*, poichè su quel computer è installato un particolare software necessario per lo sviluppo della tesi.
- c) gli studenti richiedono l'uso esclusivo di un qualsiasi computer.

I professori hanno priorità su tutti nell'accesso al laboratorio, i tesisti hanno priorità sugli studenti.

Nessuno però può essere interrotto mentre sta usando un computer (prosegue nella pagina successiva)

ASSIGNMENT 3: GESTIONE LABORATORIO

Scrivere un programma JAVA che simuli il comportamento degli utenti e del tutor. Il programma riceve in ingresso il numero di studenti, tesisti e professori che utilizzano il laboratorio ed attiva un thread per ogni utente. Ogni utente accede k volte al laboratorio, con k generato casualmente. Simulare l'intervallo di tempo che intercorre tra un accesso ed il successivo e l'intervallo di permanenza in laboratorio mediante il metodo `sleep` della classe `Thread`. Il tutor deve coordinare gli accessi al laboratorio. Il programma deve terminare quando tutti gli utenti hanno completato i loro accessi al laboratorio.

Simulare gli utenti con dei thread e incapsulare la logica di gestione del laboratorio all'interno di un monitor.