

Reti e Laboratorio III

Modulo Laboratorio III

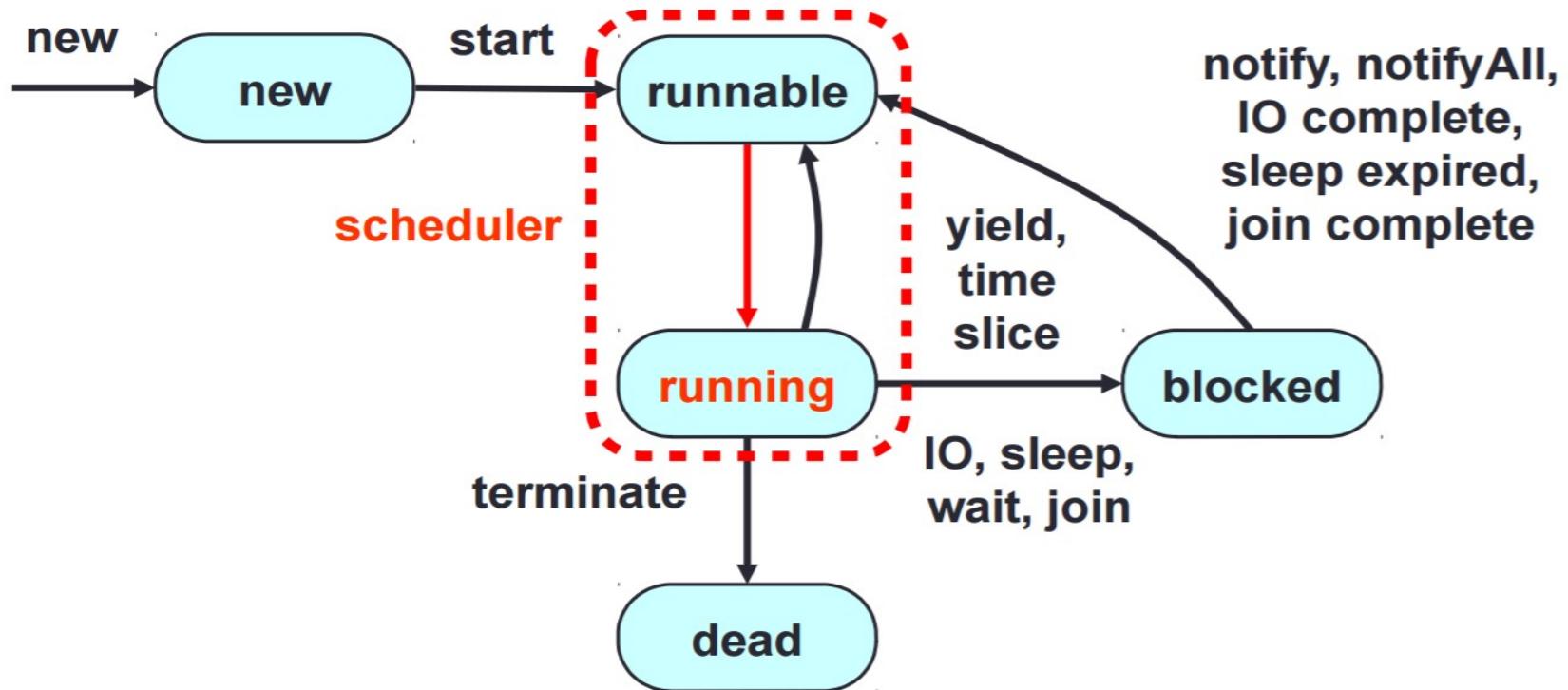
AA. 2025-2026

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 3
Callable, Monitor
13/10/2025

THREAD STATES AND JAVA OPERATIONS: RECAP



- yield segnala allo schedulatore di portare il thread corrente in stato di Runnable e di schedulare un altro thread
- sleep pone il thread nello stato di bloccato per un certo intervallo di tempo
- wait-notify la vedremo più avanti nella lezione
- join permette a un thread di attendere la terminazione di un altro thread

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, in modo asincrono, 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

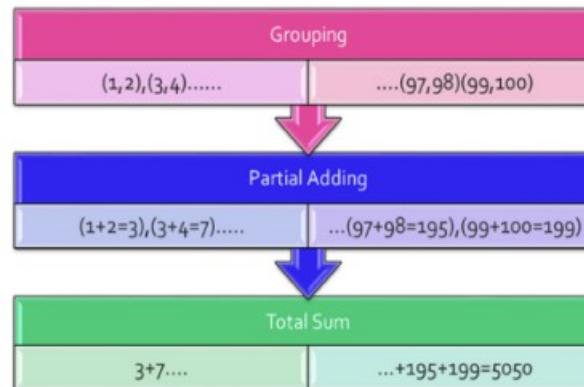
```
public interface Callable <V>  
    { V call() throws Exception; }
```

- 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`
- occorre utilizzare il metodo `submit()`, quando si sottomette la `Callable` ad un `threadpool`, invece della `execute()`

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, calcolare le somme parziali in modo concorrente
 - 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;
    }
}
```



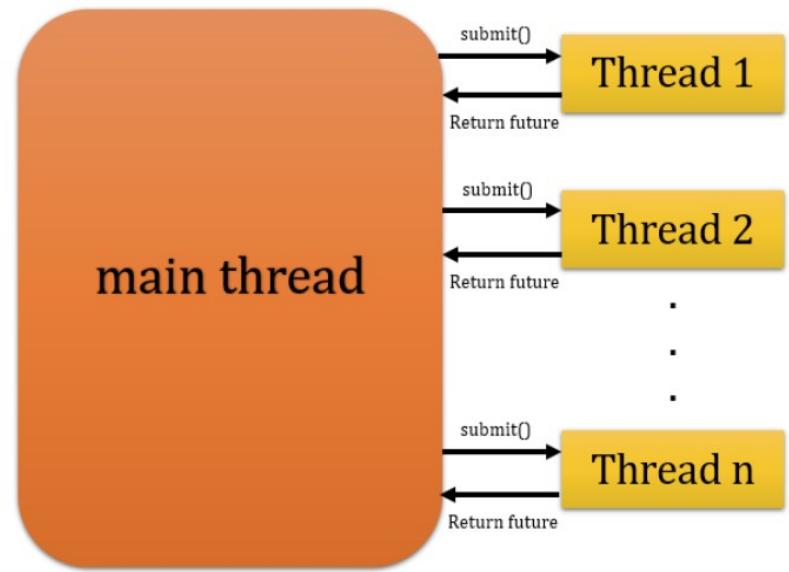
THREAD CHE RESTITUISCONO RISULTATI

```
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 = 1; i < 11; i=i+2) { // Create new Calculator object  
  
            Calculator c = new Calculator(i, i + 1);  
  
            list.add(executor.submit(c));}  
  
        int s=0;  
  
        // Now collect the results
```



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

THREAD CHE RESTITUISCONO RISULTATI

....

```
// Now collect the results

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(); }}
```



FIGHTING THREDS: CONDIVISIONE DI RISORSE



THREAD CHE CONDIVIDONO LA CONSOLE

```
class Forth implements Runnable {  
    public void run(){  
        while(true){  
            try {  
                Thread.sleep((int)(Math.random()*1000));  
            } catch (InterruptedException e) { return; }  
            System.out.print("*****");  
            System.out.flush();  
        }  
    }  
}
```



THREAD CHE CONDIVIDONO LA CONSOLE

```
class Back implements Runnable {  
    public void run(){  
        while(true){  
            try {  
                Thread.sleep((int)(Math.random()*1000));  
            } catch (InterruptedException e) { return; }  
            System.out.print("\b\b\b\b\b\b\b\b\b\b\b");  
            System.out.print("-----");  
            System.out.flush();  
        }  
    }  
}
```



THREAD CHE CONDIVIDONO LA CONSOLE

```
public class BackAndForth {  
  
    public static void main(String args[]){  
        Thread ts= new Thread(new Forth());  
        ts.start();  
        Thread bk= new Thread(new Back());  
        bk.start();  
    }  
}
```

- cosa accade se faccio eseguire il programma?



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



GENERARE UNA RACE CONDITION

```
class Cell {  
    private long value;  
  
    public Cell (long v)  
    {this.value=v;}  
  
    public void update(long delta) {  
        this.value += delta;  
    }  
  
    public long get() {return value;}  
}
```



GENERARE UNA RACE CONDITION

```
class Counter implements Runnable {  
    public int ticks ;  
    private Cell cell;  
    private int delta;  
    private int maxTicks;  
  
    Counter(Cell cell, int delta, int maxTicks) {  
        this.cell = cell;  
        this.delta = delta;  
        this.maxTicks = maxTicks;  
    }  
  
    public void run() {  
        ticks = 0;  
        while (ticks < maxTicks) {  
            cell.update(delta);  
            ++ticks;  
        }  
    }  
}
```



GENERARE UNA RACE CONDITION

```
public class Main {  
  
    public static void main(String[] args) {  
  
        int MAX_TICKS=1000000;  
  
        Cell cell=new Cell(0);  
  
        Counter up = new Counter(cell, 1, MAX_TICKS);  
  
        Counter down = new Counter(cell, -1, MAX_TICKS);  
  
        Thread upWorker = new Thread(up);  
        Thread downWorker = new Thread(down);  
  
        upWorker.start(); downWorker.start();  
  
        try {  
            upWorker.join(); downWorker.join();}  
  
        catch(Exception e) {}  
  
        System.out.println("");  
  
        System.out.printf("Cell value: %d\n", cell.get());  
  
        System.out.flush();}}
```

Risultati restituiti in esecuzioni diverse

Cell value 793507

Cell value -875261

Cell value -977321

Cell value 754164

AGGIORNARE UNA RISORSA CONDIVISA

- una singola istruzione in Cell.update

```
this.value += delta;
```

- ma eseguita non in maniera atomica....

```
// relevant bytecode
ALOAD 0
DUP
GETFIELD      Cell.value
LLOAD 1
LADD
PUTFIELD      Cell.value
```

- diversi possibili interleavings
- comportamento non deterministico: è quello che vogliamo?
- **thread safety**
 - “*nothing bad ever happens*”, in qualsiasi interleaving generato
 - spesso non banale da ottenere
 - problema di messa a punto di programmi concorrenti



JAVA: LOCK IMPLICITE

- due o più thread possono leggere un oggetto condiviso (shared objects, global data).
- è responsabilità di sincronizzare gli accessi in modo da evitare interleaving scorretti ed ottenere la thread safeness
- come effettuare la sincronizzazione, in JAVA?
 - con lock esplicite (non lo faremo)
 - in JAVA, tutti gli oggetti hanno una lock interna (intrinseca, implicita, monitor lock)
 - un synchronized method implementa una critical section con garanzie di mutua esclusione
 - i metodi acquisiscono implicitamente la lock sull'oggetto su cui vengono chiamati



```
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

```
class Cell {  
    private long value;  
    public Cell (long v)  
    {this.value=v;}  
  
    public synchronized void update(long delta) {  
        this.value += delta;  
    }  
    public long get() {return value;};  
}
```

Cell value 0

Cell value 0

Cell value 0

Cell value 0



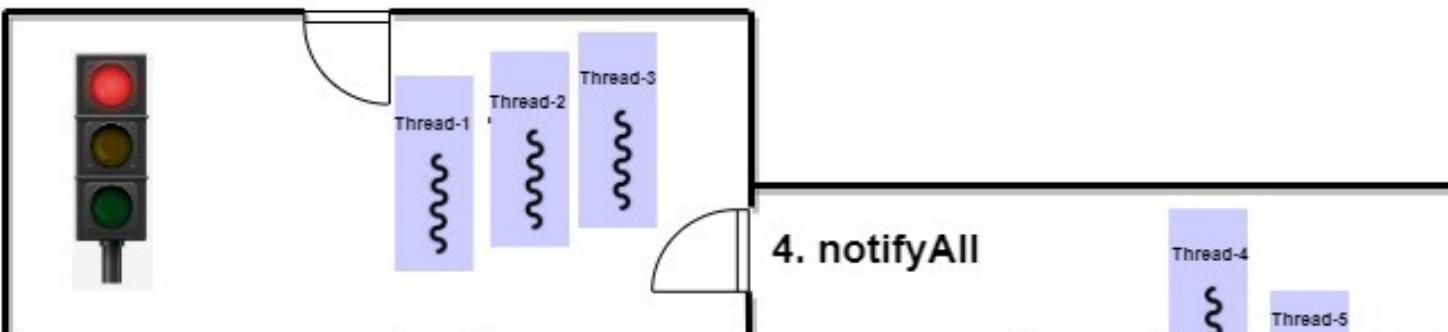
IL MONITOR

- 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 utilizzata per memorizzare i thread che hanno acquisito la lock, ma sono poi in attesa di una condizione sullo stato della risorsa
 - `wait`
 - `notify/notifyAll`



UN'OCCHIATA ALL'INTERNO DI UN MONITOR

1. Entra nell'oggetto monitor



4. notifyAll

2. Acquisisci la lock



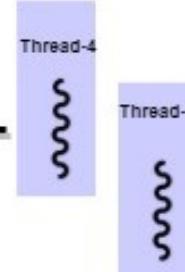
Critical Section

Active Thread



5. Rilascia la lock

3.wait



6. Lascia l'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

- i costruttori non devono essere dichiarati synchronized
 - il compilatore solleva una eccezione
 - di 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隐式 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();}
```



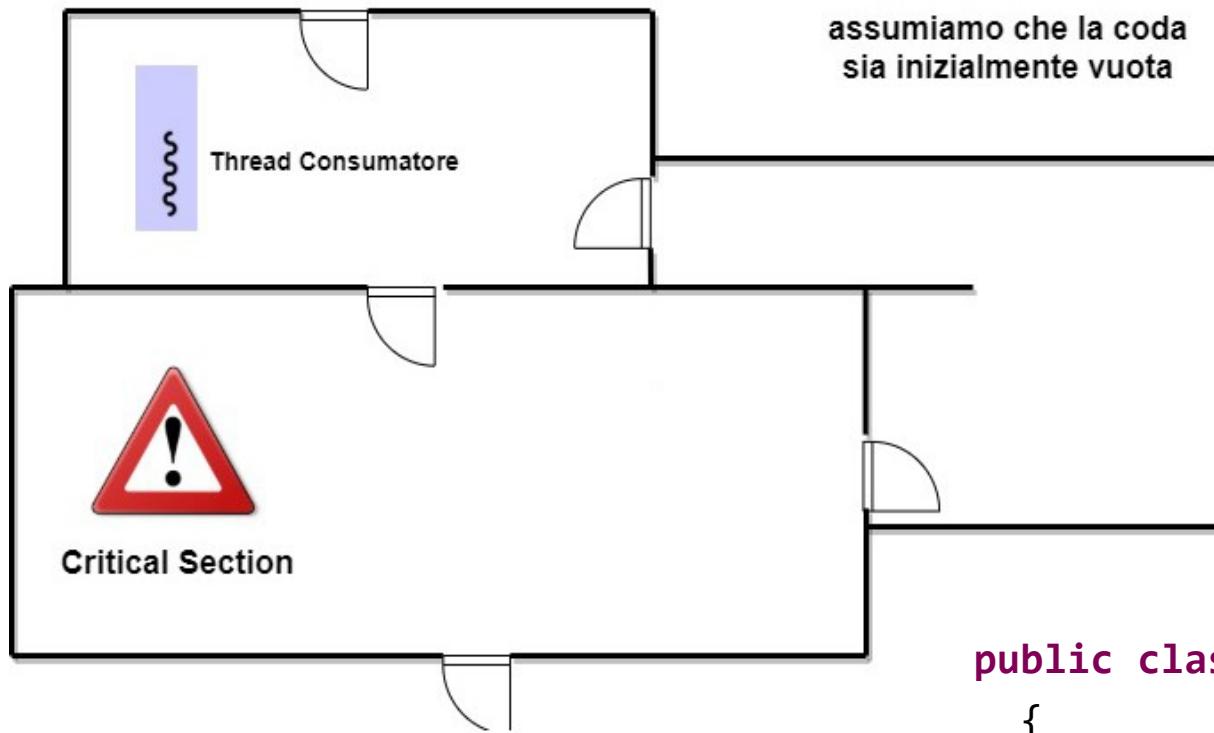
PRODUTTORE CONSUMATORE CON MONITOR

```
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”

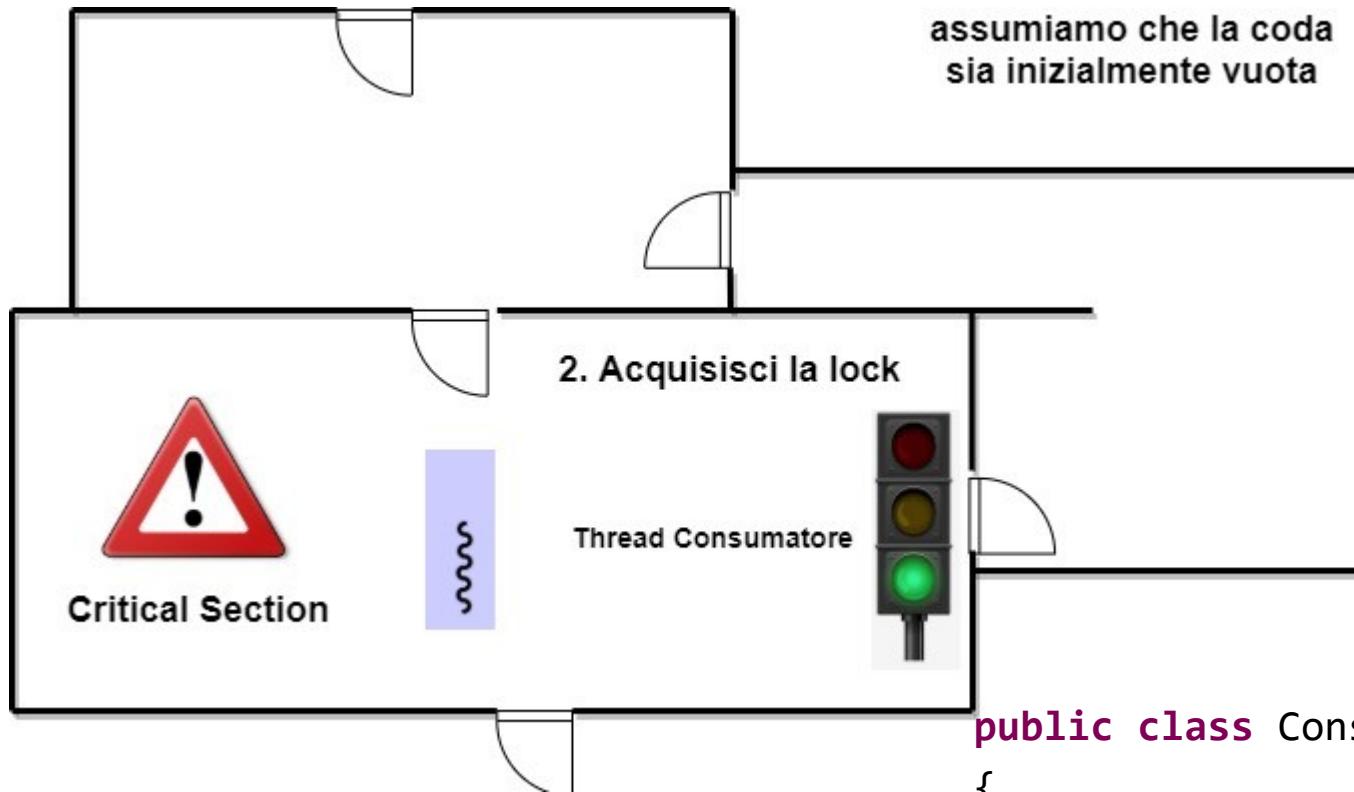
1. Entra nell'oggetto monitor



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

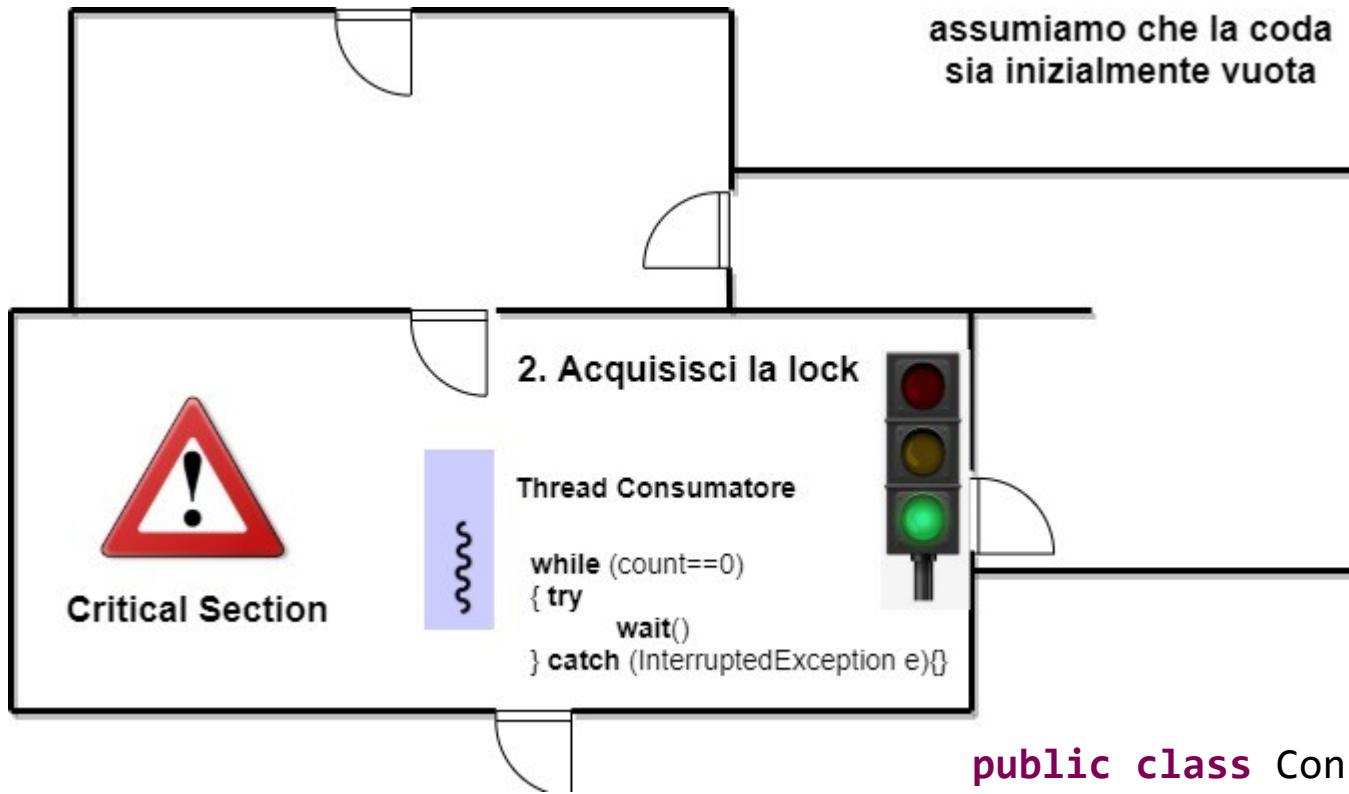


PRODUTTORE CONSUMATORE “ILLUSTRATO”



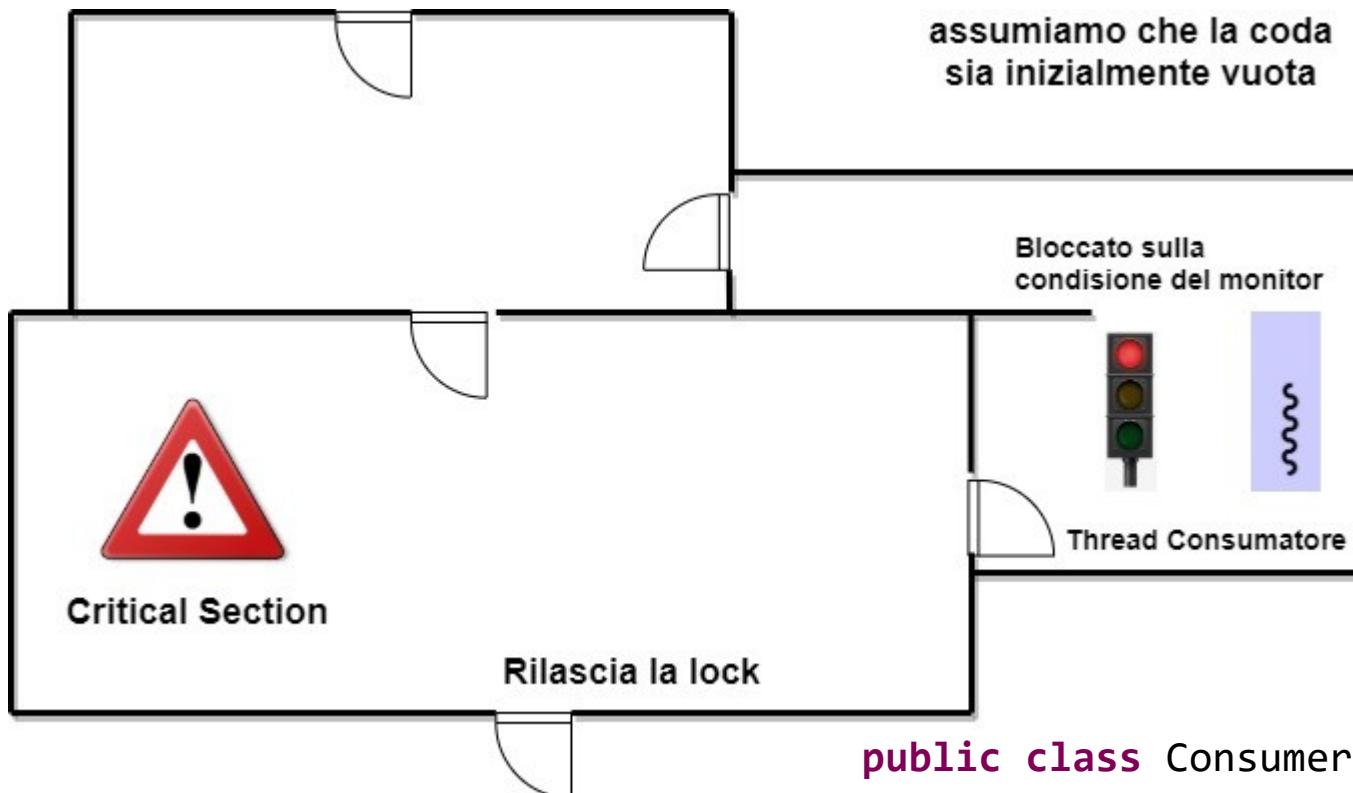
```
public class Consumer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.consume();  
            ... }  
    }  
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



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

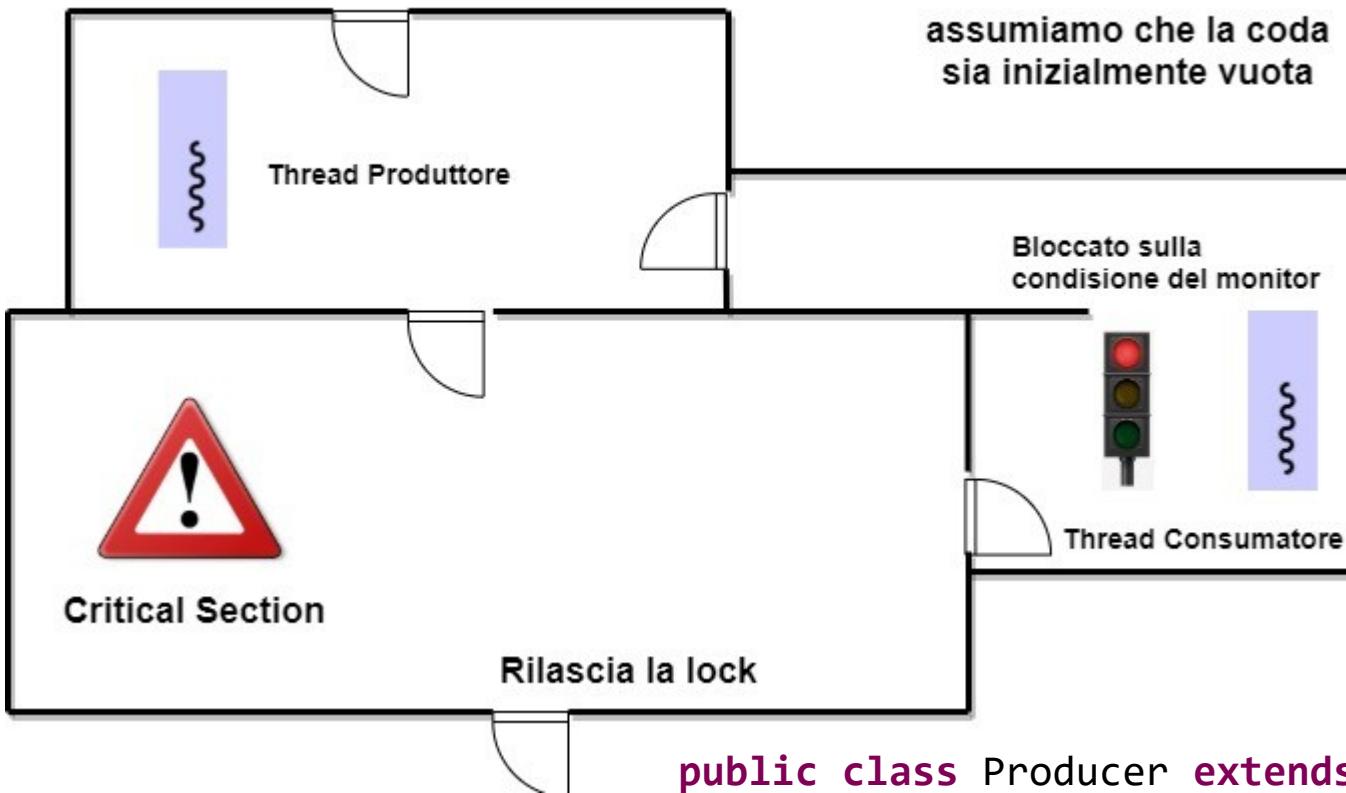
PRODUTTORE CONSUMATORE “ILLUSTRATO”



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

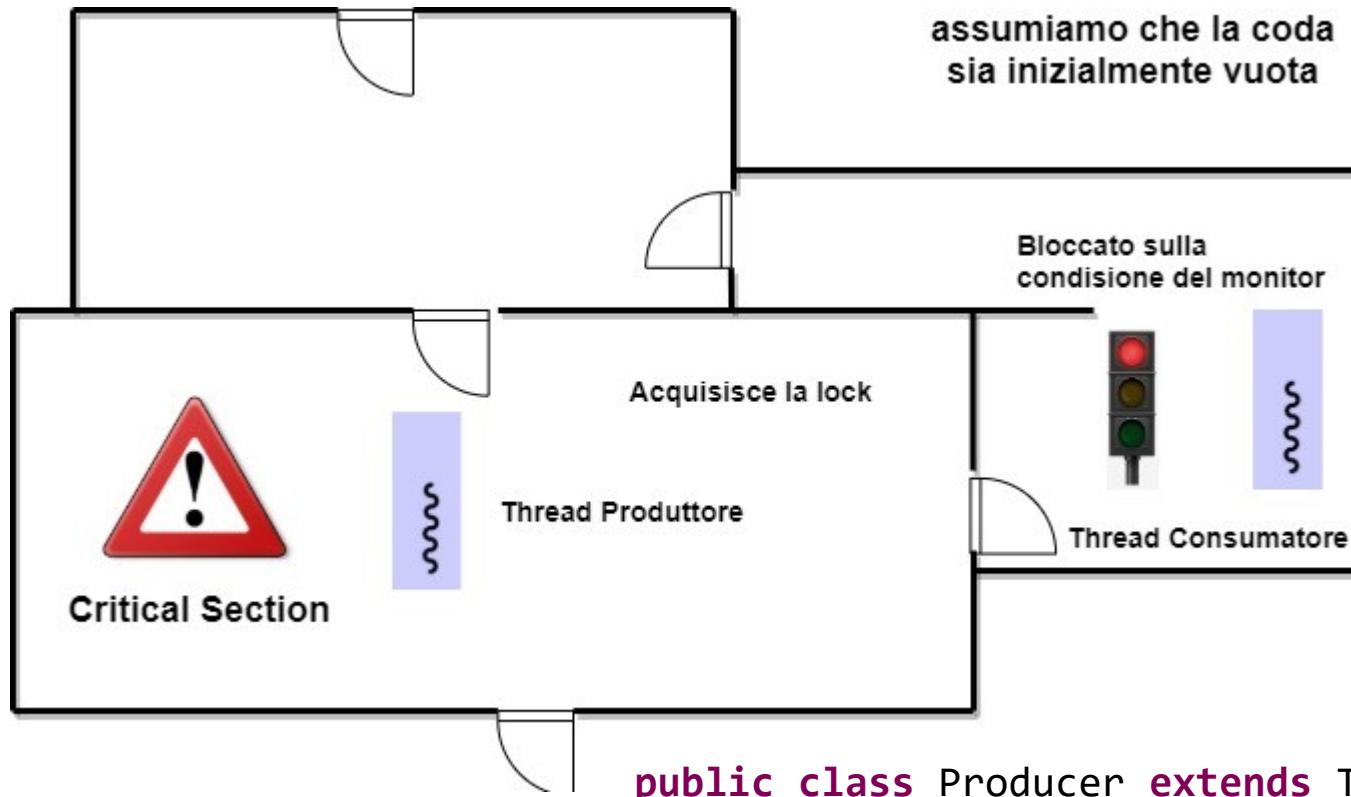
PRODUTTORE CONSUMATORE “ILLUSTRATO”

1. Entra nell'oggetto monitor



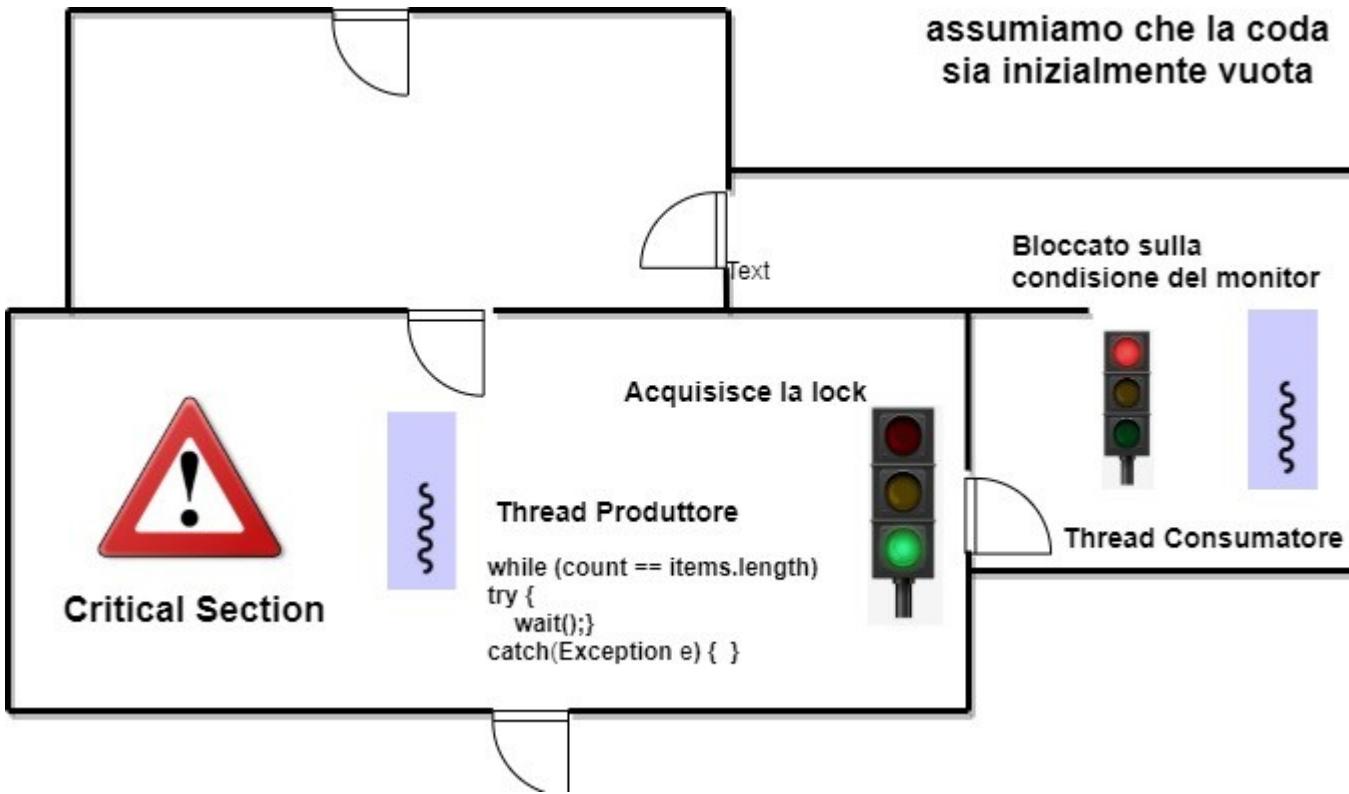
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread());  
        ....  
    }  
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



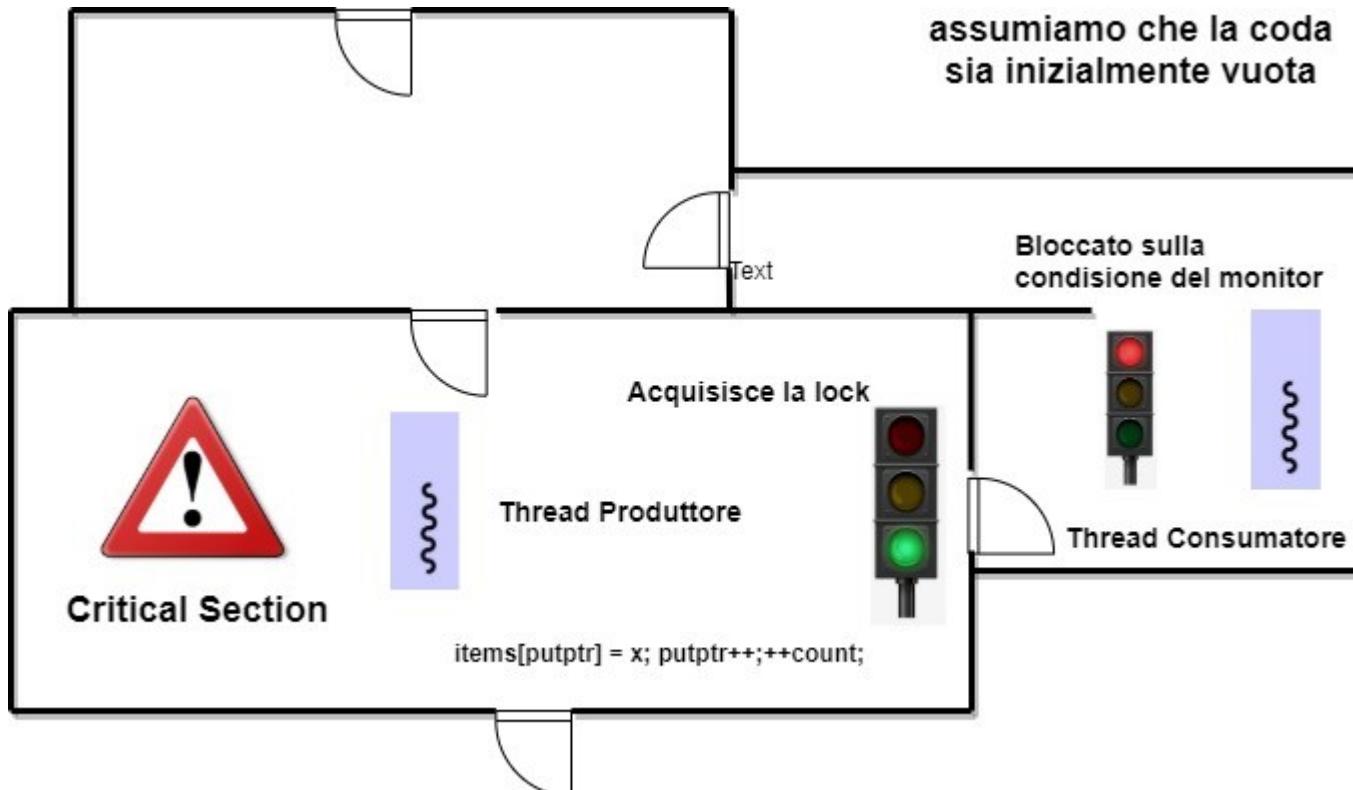
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#" + count + Thread.currentThread());  
        .....  
    }  
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



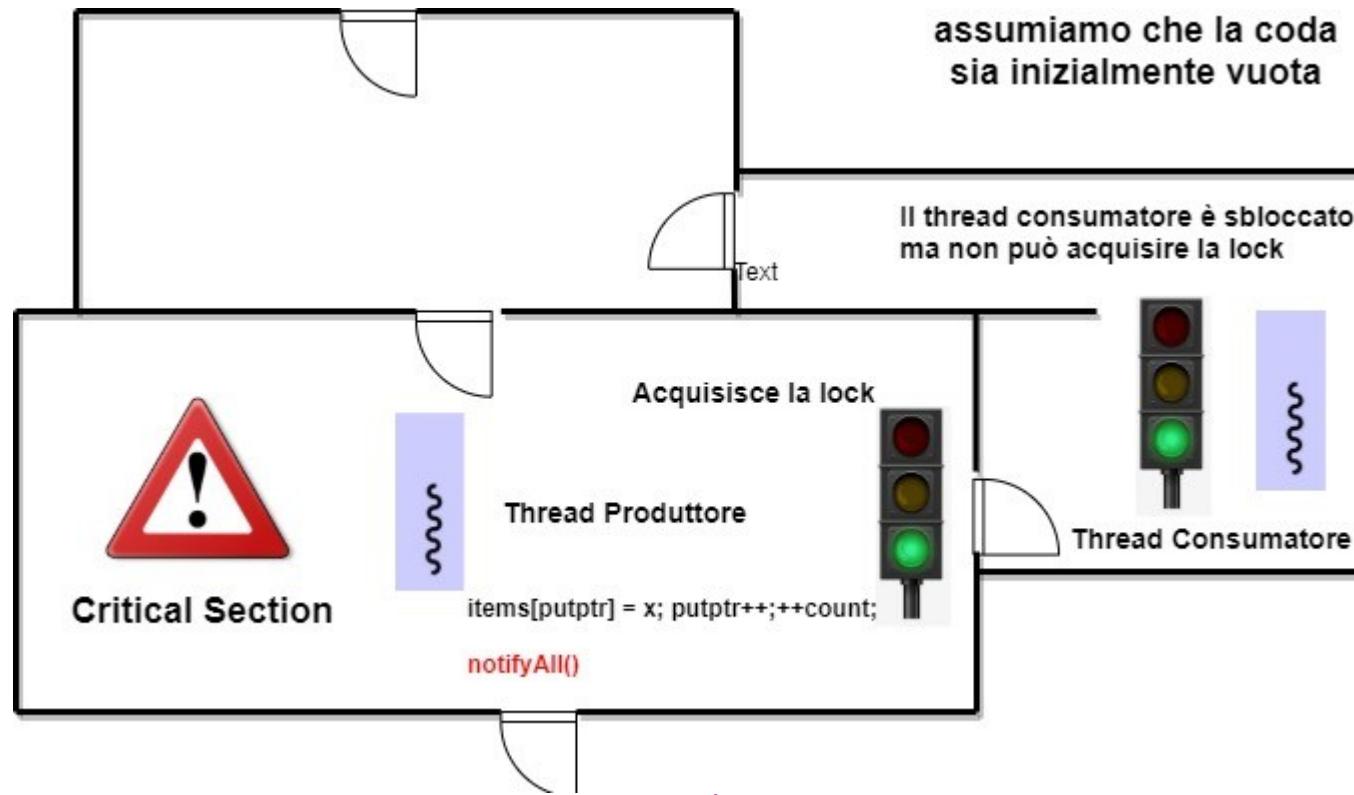
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread())  
        .....  
    }  
}
```

PRODUTTORE CONSUMATORE “ILLUSTRATO”



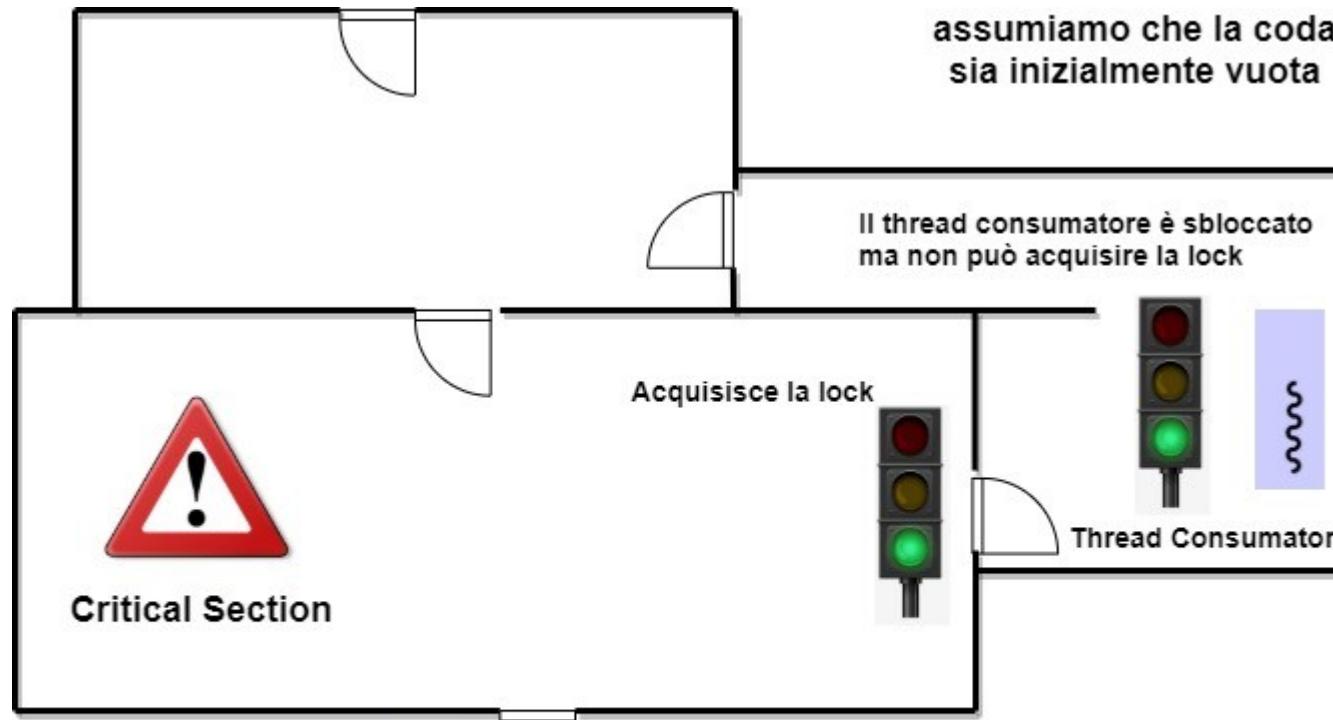
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread());  
        .....  
    }  
}
```

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”



Lascia l'oggetto monitor

```
public class Producer extends Thread
```

{

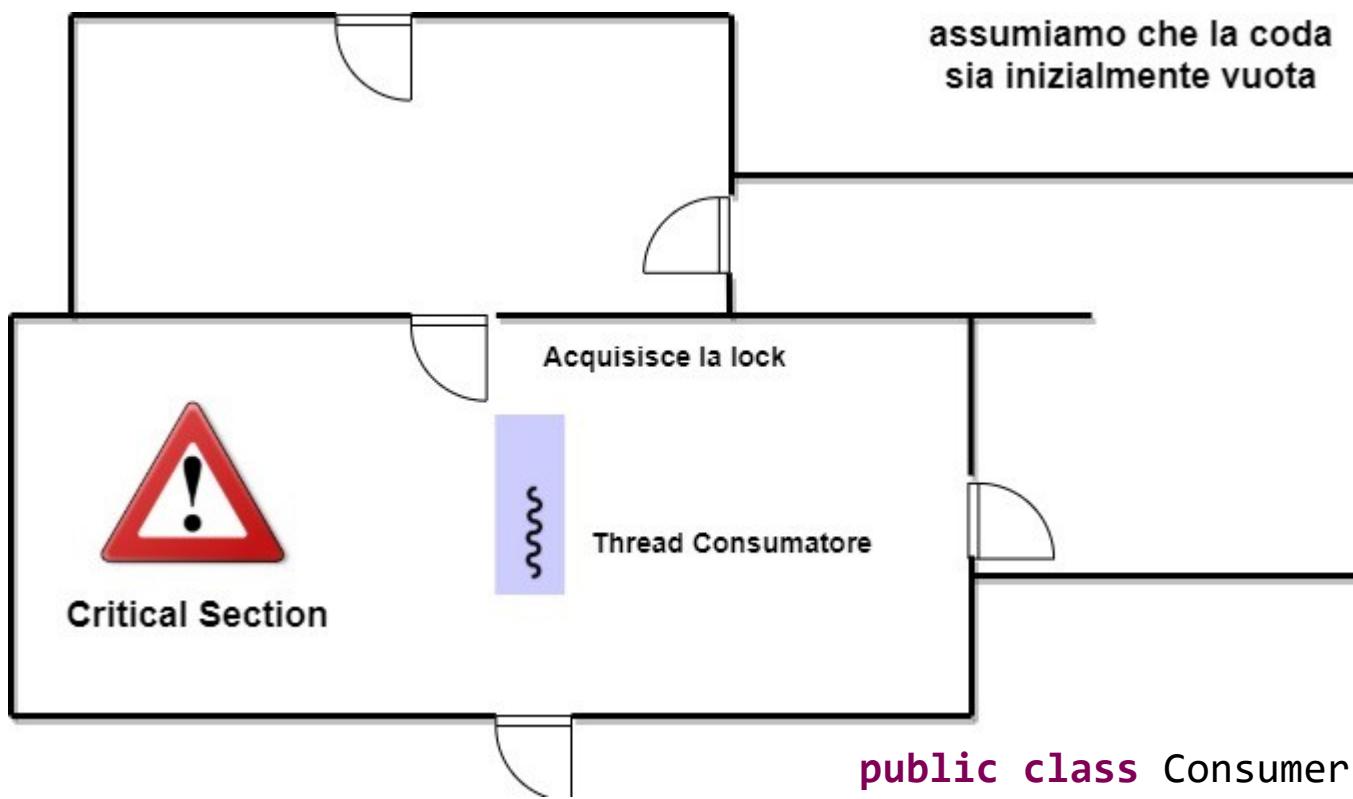
```
public void run(){
```

```
for(int i=0;i<10;i++)
```

```
{queue.produce("MSG#" + count + Thread.currentThread());
```

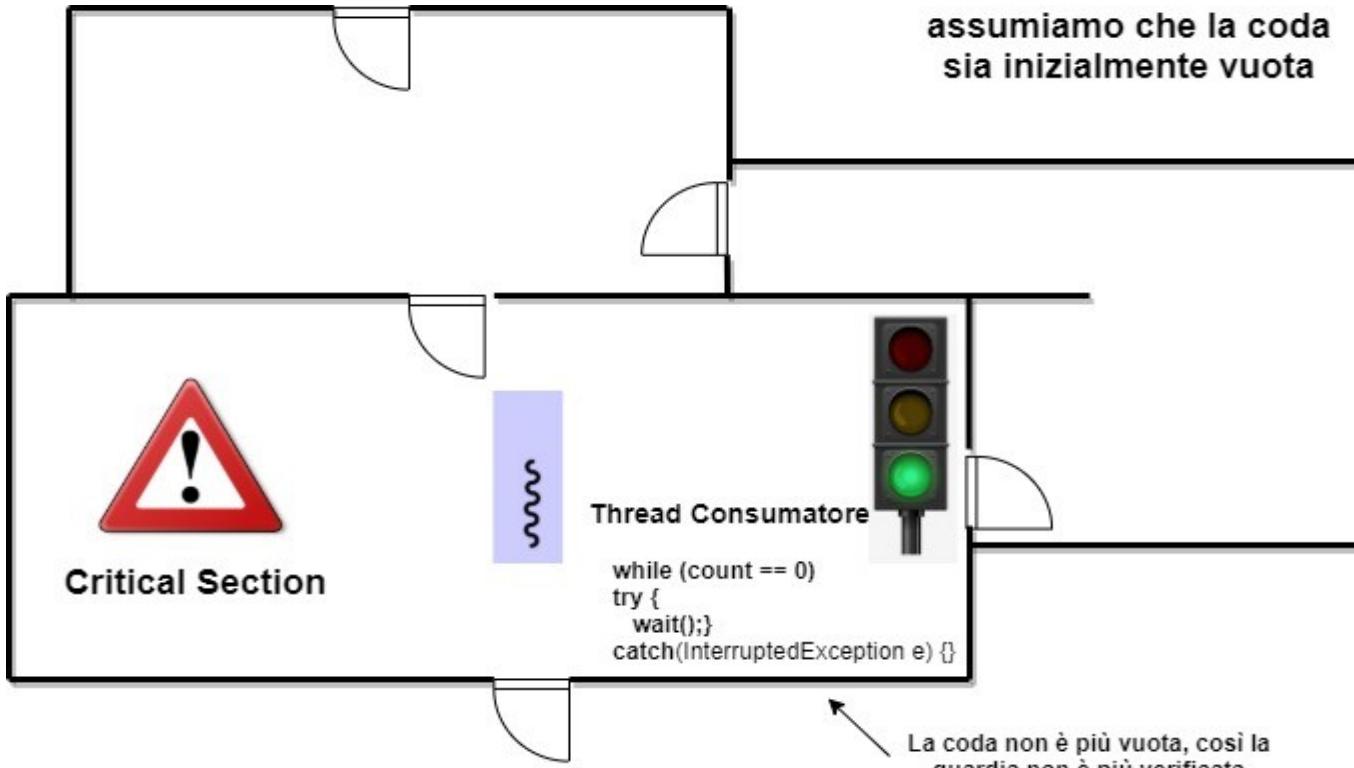
... ● ● ● ●

PRODUTTORE CONSUMATORE “ILLUSTRATO”



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

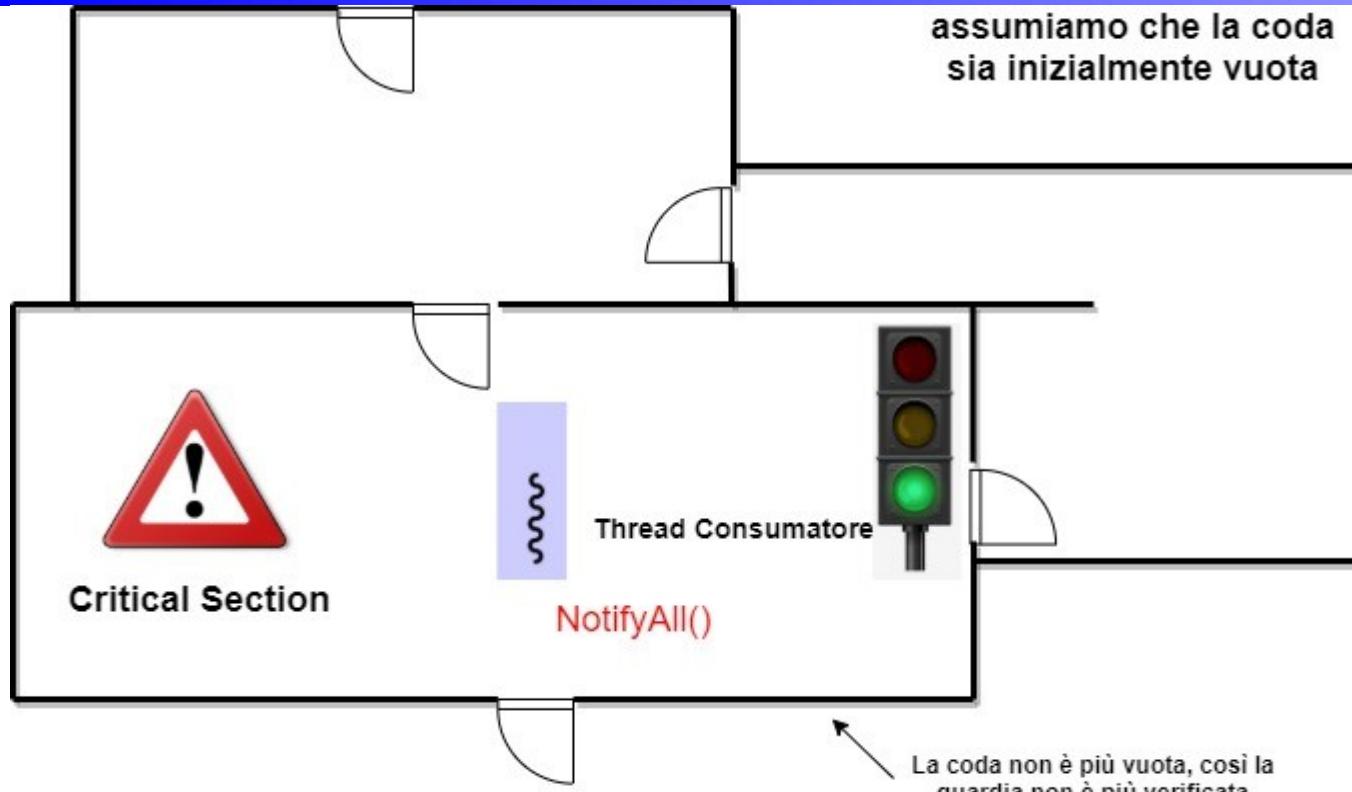
PRODUTTORE CONSUMATORE “ILLUSTRATO”



La coda non è più vuota, così la guardia non è più verificata

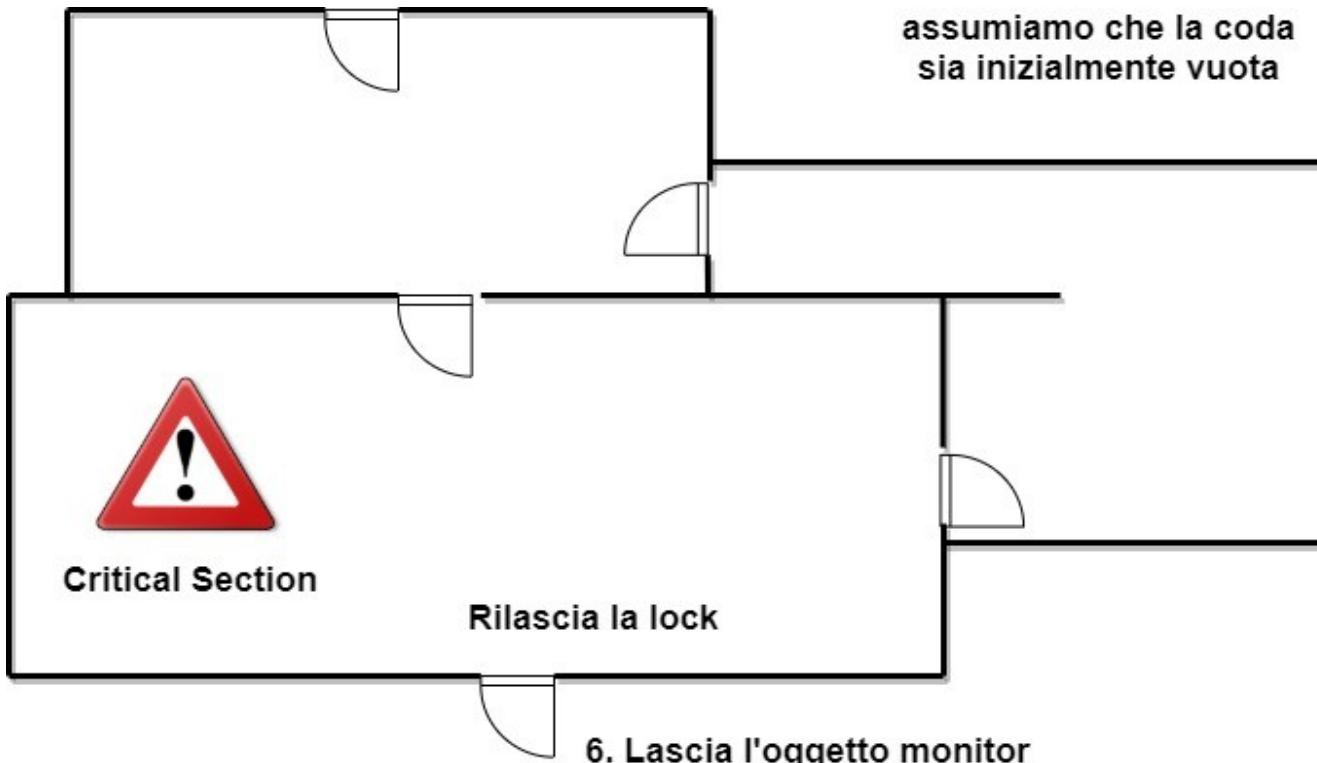
```
public class Consumer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++){
            Object o=queue.consume();
            ....}
    }
}
```

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”



6. Lascia l'oggetto monitor

```
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.

