

---

# **Laboratorio di Reti – B**

## **Lezione 3**





### **Thread synchronization: Lock e Condition Variables**

28/09/2021

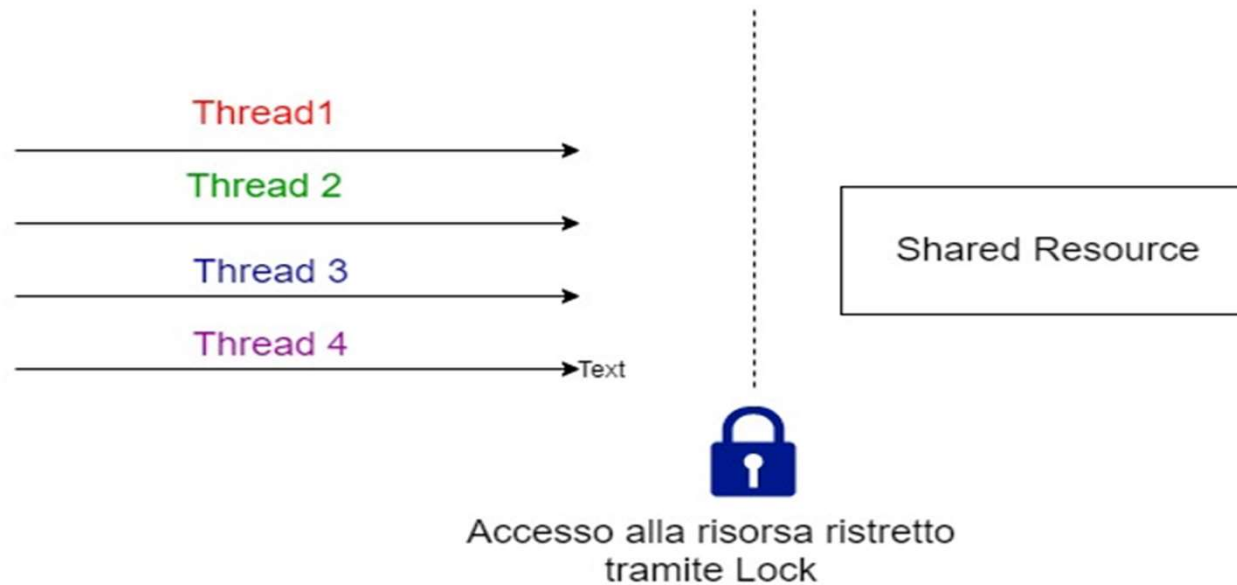
Federica Paganelli

# JAVA.UTIL.CONCURRENT IN JAVA 5

---

- esecuzione dei thread controllata e indipendente dalla logica dell'applicazione – Executor 
- possibilità di restituire un risultato per un task e lanciare eccezioni 
- classi Lock, variabili di condizione dedicate (questa lezione)
- Concurrent Collections (prossima lezione)
- Semafori 
- Variabili Atomic 

# MECCANISMI DI SINCRONIZZAZIONE: LOCK



Metafora: “come la chiave del bagno” (!)

- **chiave.lock()**: prova ad aprire la porta, se non è chiusa, entra e blocca la porta. Se è chiusa, aspetta che l'altro esca.
- **chiave.unlock()**: uscita dal bagno

# MECCANISMI DI SINCRONIZZAZIONE: LOCK

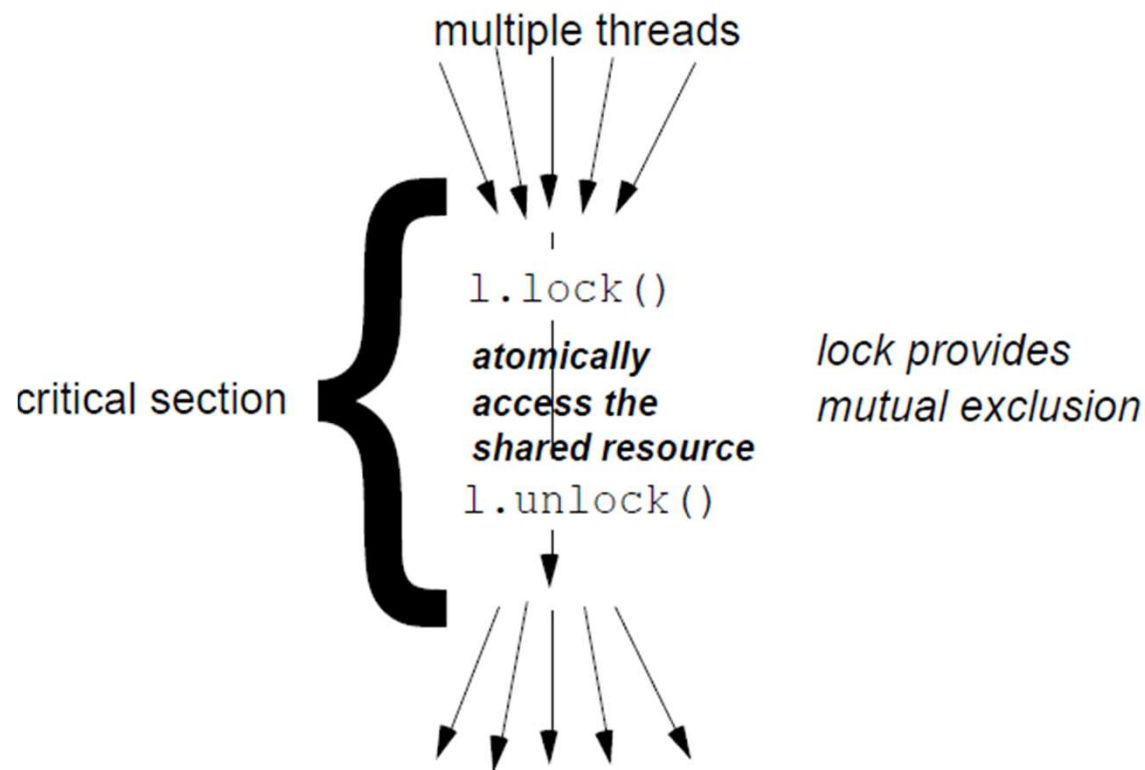
---

Cosa è una lock in JAVA?

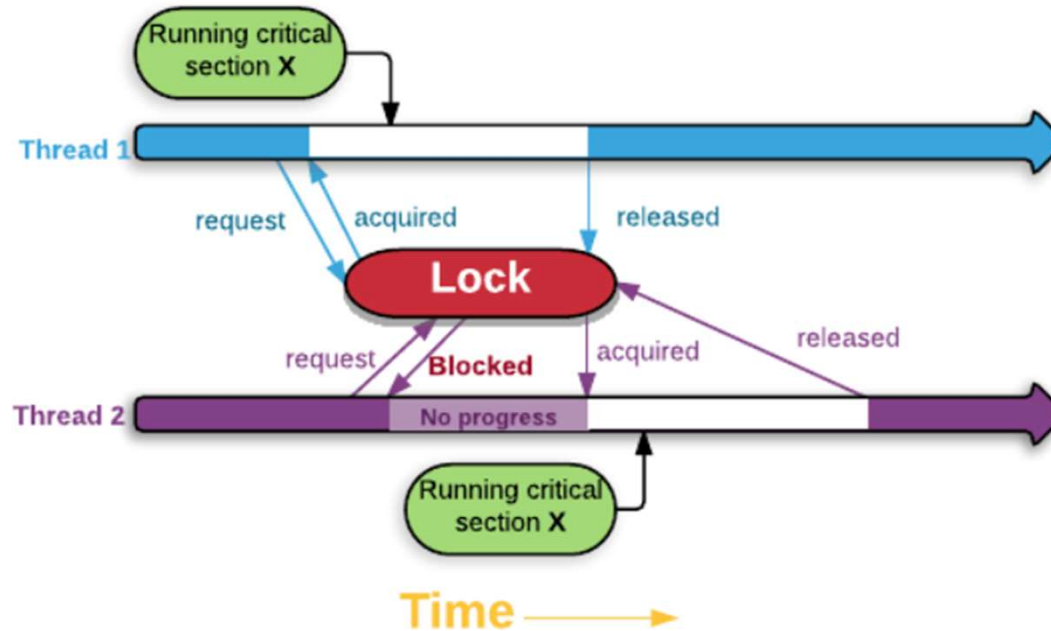
- un oggetto che può trovarsi in due stati diversi
  - “locked”/”unlocked”
  - stato impostato con i metodi: **lock( )** ed **unlock( )**
- un solo thread alla volta può impostare lo stato a “locked”, cioè ottenere la lock( )
  - gli altri thread che tentano di ottenere la lock si bloccano
- quando un thread tenta di acquisire una lock
  - rimane bloccato fintanto che la lock è detenuta da un altro thread,
  - rilascio della lock: uno dei thread in attesa la acquisisce

# MECCANISMI DI SINCRONIZZAZIONE: LOCK

- mutual exclusion lock (mutex): lock usate per definire “sezioni critiche”
- assicurano che solo un thread per volta possa entrare in una sezione critica
- Permettono di evitare “race conditions”



# MECCANISMI DI SINCRONIZZAZIONE: LOCK



la gestione dei thread bloccati dipende dalla politica di fairness

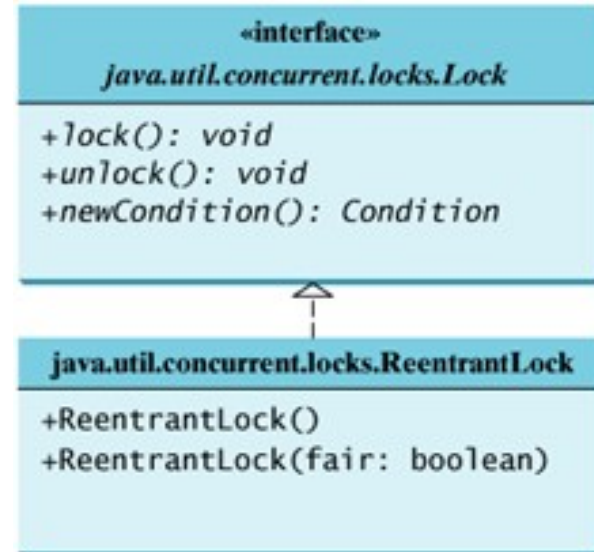
- **fairness:** thread in attesa serviti secondo una politica FIFO
- **non fairness;** non viene garantito un ordine di accesso particolare (generalmente politica di default)

# MECCANISMI DI SINCRONIZZAZIONE: LOCK

Interfaccia

`java.util.concurrent.locks.Lock`

```
interface Lock {  
    void lock();  
    void lockInterruptibly()  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit)  
    void unlock();  
    Condition newCondition() }
```



Implementazioni

- `ReentrantLock`
- `ReentrantReadWriteLock.ReadLock`
- `ReentrantReadWriteLock.WriteLock`

## Lock: utilizzo

---

```
Lock l = ...;  
l.lock();  
try {  
    // access the resource protected by this lock  
} finally {  
    l.unlock();  
}
```

notare l'uso del blocco finally

- cosa accade se viene sollevata un'eccezione?
- il metodo `unlock()` nel `finally()` garantisce che la lock venga rilasciata



# MECCANISMI DI SINCRONIZZAZIONE: LOCK

---

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

public class Account {
    private double balance;
    private final Lock accountLock=new ReentrantLock();
    public double getBalance() { return balance; }
    public void setBalance(double balance) { this.balance = balance;}
    public void addAmount(double amount) {
        try {
            accountLock.lock();
            double tmp=balance;
            tmp+=amount;
            balance=tmp;
        }
        finally {
            accountLock.unlock();
        }
    }
}
```

# MECCANISMI DI SINCRONIZZAZIONE: LOCK

```
public void subtractAmount(double amount){  
    accountLock.lock();  
    double tmp=balance;  
    tmp-=amount;  
    balance=tmp;  
    accountLock.unlock();  
}
```

- Output di alcune esecuzioni del programma:

Account : Initial Balance: 1000,000000

Account : Final Balance: 1000,000000

Account : Initial Balance: 1000,000000

Account : Final Balance: 1000,000000

## LOCK E PERFORMANCE

---

- L'uso delle lock introduce overhead, per cui vanno usate con oculatezza

- Inserire l'istruzione

```
long time1=System.currentTimeMillis();
```

prima dell'attivazione dei threads

e le istruzioni

```
long time2=System.currentTimeMillis();
```

```
System.out.println(time2-time1);
```

```
System.out.println(count);}}
```

alla fine del programma

Il tempo di esecuzione del programma senza uso di lock è circa la metà di quello con uso di lock !

# LOCKS E PERFORMANCE

---

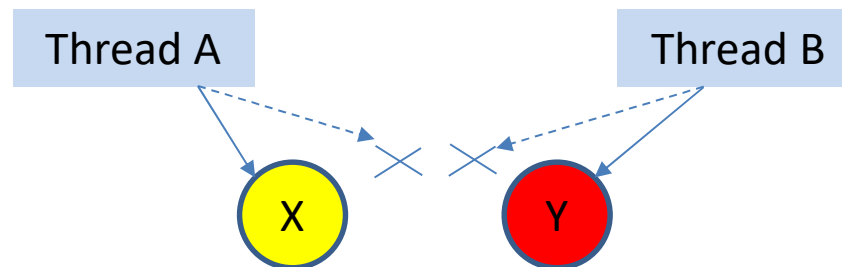
Le lock introducono una perdita di prestazioni dovuta a più fattori

- contention
- bookkeeping
- scheduling
- blocking
- unblocking

Performance penalty caratterizza tutti i costrutti a più alto livello introdotti da JAVA, basati su lock (synchronized, monitors,...)

# MECCANISMI DI SINCRONIZZAZIONE: LOCK

- Attenzione ai deadlocks:
  - **Thread(A)** acquisisce **Lock(X)** e **Thread(B)** acquisisce **Lock(Y)**
  - **Thread(A)** tenta di acquisire **Lock(Y)** e simultaneamente **Thread(B)** tenta di acquisire **Lock(X)**
  - Entrambe i threads bloccati all'infinito, in attesa della lock detenuta dall'altro thread!



- L'interfaccia **Lock** e la classe **ReentrantLock** che la implementa include un altro metodo utilizzato per ottenere il controllo della lock: **tryLock()**
  - tenta di acquisire la lock() e se essa è già posseduta da un altro thread, il metodo termina immediatamente e restituisce il controllo al chiamante.
  - restituisce un valore booleano, vero se è riuscito ad acquisire la lock(), falso altrimenti

# REENTRANT LOCKS

---

```
private ReentrantLock = lock new ReentrantLock();  
private static void accessResource() {  
    lock.lock();  
    // aggiorna risorsa  
    if (some condition()) {  
        accessResource();  
    }  
    lock.unlock();}}
```

invocazioni ricorsive per l'aggiornamento della risorsa

# REENTRANT LOCKS

---

- nel programma precedente il thread potrebbe entrare in deadlock con se stesso!
- per evitare queste situazioni: **reentrant locks** o **recursive lock**: utilizzano un contatore
  - **un thread può acquisire più volte la lock su uno stesso oggetto senza bloccarsi**
  - Il contatore viene incrementato ogni volta che un thread acquisisce la lock
  - E decrementato ogni volta che un thread rilascia la lock
  - lock viene definitivamente rilasciata quando il contatore diventa 0
- non tutte le implementazioni di lock sono rientranti
- Il meccanismo delle **lock rientranti** aiuta a prevenire situazioni di deadlock

# LOCK INTERRUPTIBLY

---

- se un thread è bloccato in attesa di una lock intrinseca, non è possibile “interagirci” in alcun modo, solo se acquisirà la lock sarà possibile inviargli una interruzione

LockInterruptibly()

- consente di “rispondere” ad una interruzione, mentre si è in attesa di lock()
- solleva una InterruptedException quando un altro metodo invoca il metodo interrupt

```
private Lock lock= new ReentrantLock();  
    public void increment() {  
        try {  
            lock.lockInterruptibly();  
            this.count++;  
        } catch(InterruptedException e) {  
            //do something }  
        finally {lock.unlock();}  
    }
```



# Read/Write Locks

---

- Ipotesi: applicazione che legge e scrive una risorsa
- La scrittura è meno frequente delle operazioni di lettura
- Due thread che leggono la stessa risorsa non causano problemi l'uno all'altro
- Invece, se un singolo thread desidera scrivere sulla risorsa, non devono essere in corso altre operazioni di lettura o scrittura.
- La ReentrantLock garantisce mutua esclusione ma è una soluzione inefficiente
- **Soluzione: ReadWriteLock**
  - Una lock di lettura: più thread possono acquisire la lock per leggere la risorse se nessun thread detiene la lock di scrittura.
  - Una lock di scrittura: un singolo thread alla volta può modificare la risorsa

# READ/WRITE LOCKS

---

- **interfaccia ReadWriteLock**: mantiene una coppia di lock associate, una per le operazioni di lettura e una per le scritture.
  - la read lock può essere acquisita da più thread lettori, purché non vi sia uno scrittore che ha acquisito la lock.
  - la write lock è esclusiva.
- implementazione: **ReentrantReadWriteLock()**

## Esempio senza READ WRITE Lock

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

public class SharedLocks {
    int a =1000, b=0;
    ReentrantLock l = new ReentrantLock();
    public int getsum () {
        int result;
        l.lock();
        result=a+b;
        l.unlock();
        return result;
    }

    public void transfer (int x) {
        l.lock();
        a = a-x;
        b = b+x;
        l.unlock();
    }
}
```

- L'operazione di transfer( ) non interferisce con la getSum( )
- Ma non è non consentita l'esecuzione concorrente di getSum() diverse.

## Esempio con READ WRITE LOCK

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

public class SharedLocks extends Thread {
    int a =1000, b=0;
    private ReentrantReadWriteLock readWriteLock = new
                                                ReentrantReadWriteLock();
    private Lock read  = readWriteLock.readLock();
    private Lock write = readWriteLock.writeLock();
    public int getsum (){
        int result;
        read.lock();
        result=a+b;
        read.unlock();
        return result;};
    public void transfer (int x) {
        write.lock();
        a = a-x;
        b = b+x;
        write.unlock();  }}
```

# THREAD COOPERATION

---

- l'interazione esplicita tra threads avviene in un linguaggio ad oggetti mediante l'utilizzo di **oggetti condivisi**
  - esempio: produttore/consumatore il **produttore P** produce un nuovo valore e lo comunica ad un thread **consumatore C**
- il valore prodotto viene incapsulato in un **oggetto condiviso** da P e da C, ad esempio una **coda** che memorizza i messaggi scambiati tra P e C
- la mutua esclusione sull'oggetto condiviso è garantita dall'uso di lock o metodi synchronized (prossima lezione), ma spesso non è sufficiente garantire **sincronizzazioni esplicite**
  - Ad es. un thread entra in una sezione critica e si blocca aspettando che una condizione venga soddisfatta, detenendo la lock...
    - Ad esempio un produttore vuole aggiungere un nuovo valore ma la coda è piena...
    - O viceversa

# THREAD COOPERATION

---

- E' necessario introdurre costrutti per **sospendere** un thread T quando **una condizione C** non è verificata e per **riattivare** T quando diventa vera
  - il produttore si sospende se il buffer è pieno rilasciando il lock
  - si riattiva quando c'è una posizione libera
  - compete per riacquisire il lock

# THREAD COOPERATION: PRODUTTORE/CONSUMATORE

---

- uno o più threads producono dati
  - **add**: aggiunge un elemento in fondo alla coda
- uno o più threads consumano dati
  - rimuove un elemento dalla testa della coda (FIFO)
- i threads interagiscono mediante una coda condivisa
  - se la coda è vuota, il/i consumatori si bloccano
  - se la coda è piena, il/i produttori si bloccano

# THREAD COOPERATION: PRODUTTORE/CONSUMATORE

---

- Ipotesi per l'esempio successivo
  - non si utilizzano strutture dati sincronizzate di JAVA (**blockingqueue**)
  - la coda è realizzata mediante una ArrayList la cui dimensione massima è prefissata



# THREAD COOPERATION

---

```
import java.util.*;
import java.util.concurrent.locks.*;
public class MessageQueue {
    private int bufferSize;
    private List<String> buffer = new ArrayList<String>();
    private ReentrantLock l = new ReentrantLock();
    public MessageQueue(int bufferSize){
        if(bufferSize<=0)
            throw new IllegalArgumentException("Size is illegal.");
        this.bufferSize = bufferSize; }
    public boolean isFull() {
        return buffer.size() == bufferSize; }
    public boolean isEmpty() {
        return buffer.isEmpty(); }
```

...

## THREAD COOPERATION: SOLUZIONE a)

---

```
public void put(String message){
```

```
    l.lock();
```

```
    while (isFull()) { }
```

```
    buffer.add(message);
```

```
    l.unlock(); }
```

```
public String get(){
```

```
    l.lock();
```

```
    while (isEmpty()) { }
```

```
    String message = buffer.remove(0);
```

```
    l.unlock();
```

```
    return message;
```

```
}
```

```
}}
```

ATTENZIONE: QUESTA SOLUZIONE

NON E' CORRETTA!!

- il thread che acquisisce la lock( ) e non può effettuare l'operazione a causa dello stato della risorsa, deve rilasciare la lock() per dare la possibilità ad altri thread di modificare lo stato della coda in modo che la condizione sia verificata

## THREAD COOPERATION SOLUZIONE b)

```
public void put (String message) {  
    l.lock();  
    while (isFull()) {  
        l.unlock();  
        l.lock(); }  
    buffer.add(message);  
    l.unlock();  
}  
public String get() {  
    l.lock();  
    while (isEmpty()) {  
        l.unlock();  
        l.lock(); }  
    String message = buffer.remove(0);  
    l.unlock(); return message;  
}  
}
```

ATTENZIONE: ANCHE QUESTA SOLUZIONE  
PRESENTA DEI PROBLEMI

1) BUSY WAITING  
2) LA CORRETTEZZA DIPENDE DALLA  
STRATEGIA DI SCHEDULAZIONE DEI  
THREAD

# THREAD COOPERATION SOLUZIONE c)

```
public void put (String message) {
```

```
    l.lock();
```

```
    while (isFull()) {
```

```
        l.unlock();
```

```
        Thread.yield();
```

```
        l.lock();
```

```
    }
```

```
    buffer.add(message);
```

```
    l.unlock(); }
```

```
public String get() {
```

```
    l.lock();
```

```
    while (isEmpty()) {
```

```
        l.unlock();
```

```
        Thread.yield();
```

```
        l.lock(); }
```

```
    String message = buffer.remove(0);
```

```
    l.unlock(); return message; }}
```

ATTENZIONE: LA CORRETTEZZA DIPENDE  
DALLA IMPLEMENTAZIONE DELLA YIELD

## THREAD COOPERATION SOLUZIONE d)

---

- I metodi sono eseguiti in mutua esclusione sull'oggetto condiviso.
- E' necessario inoltre
  - definire un insieme di **condizioni sullo stato** dell'oggetto condiviso
  - implementare meccanismi di **sospensione/riattivazione** dei threads sulla base del valore di queste condizioni
  - implementazioni possibili:
    - variabili di condizione:
      - definizione di **variabili di condizione**
      - metodi per **la sospensione** su queste variabili che usano **code** associate alle variabili in cui memorizzare i threads sospesi
    - meccanismi di monitoring ad alto livello

# CONDITION VARIABLES

---

- ad una lock possono essere associate un insieme di **variabili condizioni**
- lo scopo di queste condizioni è quello di permettere ai thread di controllare se una **condizione sullo stato della risorsa è verificata o meno** e
  - se la condizione è falsa, di **sospendersi rilasciando la lock()** ed inserire il thread in una coda in attesa di quella condizione
  - risvegliare un thread in attesa quando la condizione risulta verificata
- per ogni oggetto diverse code:
  - una per i threads in attesa di acquisire la lock( )
  - una associata ad ogni variabile di condizione
- sospensione su variabili di condizione associate ad un oggetto solo dopo aver acquisito la lock() su quell'oggetto, altrimenti

**IllegalMonitorException**

# CONDITION VARIABLES

---

- Oggetti di tipo **Condition** associati ad un oggetto lock( ).
- l'interfaccia **Condition** fornisce i meccanismi per sospendere un thread e per risvegliarlo

```
interface Condition {  
    void await()  
    boolean await( long time, TimeUnit unit )  
    long awaitNanos( long nanosTimeout)  
    void awaitUninterruptibly()  
    boolean awaitUntil( Date deadline)  
    void signal();  
    void signalAll();}
```

# PRODUTTORE/CONSUMATORE CON CONDIZIONI

---

```
public class Messagesystem {  
    public static void main(String[] args) {  
        MessageQueue queue = new MessageQueue(10);  
        new Producer(queue).start();  
        new Producer(queue).start();  
        new Producer(queue).start();  
        new Consumer(queue).start();  
        new Consumer(queue).start();  
        new Consumer(queue).start();  
    }  
}
```



## Esempio: PRODUTTORE

---

```
import java.util.concurrent.locks.*;
public class Producer extends Thread {
    private int count = 0;
    private MessageQueue queue = null;
    public Producer(MessageQueue queue){
        this.queue = queue;
    }
    public void run(){
        for(int i=0;i<10;i++){
            queue.put("MSG#" + count + Thread.currentThread());
            count++;
        }
    }
}
```

## Esempio: Consumer

---

```
public class Consumer extends Thread {  
    private MessageQueue queue = null;  
    public Consumer(MessageQueue queue){  
        this.queue = queue;  
    }  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.get();  
            int x = (int)(Math.random() * 10000);  
            try{  
                Thread.sleep(x);  
            }catch (Exception e){};  
        }  
    }  
}
```

## Esempio: Coda di messaggi

---

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

public class MessageQueue {
    final Lock lockcoda;
    final Condition notFull;
    final Condition notEmpty;
    int putptr, takeptr, count;
    final Object[] items;

    public MessageQueue(int size){
        lockcoda = new ReentrantLock();
        notFull = lockcoda.newCondition();
        notEmpty = lockcoda.newCondition();
        items = new Object[size];
        count=0;putptr=0;takeptr=0;}
}
```

# PRODUTTORE/CONSUMATORE CON CONDIZIONI

```
public void put(Object x) {  
    lockcoda.lock();  
    try {  
        while (count == items.length)  
            notFull.await();  
        // gestione puntatori coda  
        items[putptr] = x; putptr++; ++count;  
        if (putptr == items.length) putptr = 0;  
        System.out.println("Message Produced"+x);  
        notEmpty.signal();  
    }  
    finally {  
        lockcoda.unlock();  
    }  
}
```

# PRODUTTORE/CONSUMATORE CON CONDIZIONI

```
public Object get() {  
    lockcoda.lock();  
    try {  
        while (count == 0)  
            notEmpty.await();  
        Object x = items[takeptr];  
        takeptr=takeptr+1;  
        if (takeptr == items.length)  
            takeptr = 0;  
        --count;  
        notFull.signal();  
        System.out.println("Message Consumed"+x);  
        return x;}  
    finally {  
        lockcoda.unlock();  
    }  
}
```

# PRODUTTORE/CONSUMATORE CON CONDIZIONI

---

```
Message ProducedMSG#0Thread[Thread-2,5,main]
Message ProducedMSG#0Thread[Thread-0,5,main]
Message ProducedMSG#0Thread[Thread-1,5,main]
Message ProducedMSG#1Thread[Thread-2,5,main]
Message ProducedMSG#1Thread[Thread-0,5,main]
Message ProducedMSG#1Thread[Thread-1,5,main]
Message ProducedMSG#2Thread[Thread-2,5,main]
Message ConsumedMSG#0Thread[Thread-2,5,main]
Message ProducedMSG#2Thread[Thread-0,5,main]
Message ProducedMSG#2Thread[Thread-1,5,main]
Message ConsumedMSG#0Thread[Thread-0,5,main]
Message ConsumedMSG#0Thread[Thread-1,5,main]
Message ProducedMSG#3Thread[Thread-2,5,main]
Message ProducedMSG#3Thread[Thread-0,5,main]
Message ProducedMSG#3Thread[Thread-1,5,main]
Message ProducedMSG#4Thread[Thread-2,5,main]
Message ConsumedMSG#1Thread[Thread-2,5,main]
.....
```

## Riassumendo...

---

- `yield()`: una indicazione allo scheduler che segnala l'intenzione di rilasciare l'uso della CPU temporaneamente e consentire ad altri thread in stato Runnable (qualora ve ne siano) di avere una possibilità per essere eseguiti. Lo scheduler può ignorare questa indicazione. Non rilascia lock.
- `sleep()`: thread in pausa per un certo periodo di tempo. Nessun lock in possesso del thread viene rilasciato
- `await()` su un'istanza di Condition sospende l'esecuzione del thread e il lock associato è rilasciato. Rimane sospeso finché un thread notifica un cambiamento di stato della condizione (*signal()* or *signalAll()*) oppure un altro thread interrompe il thread (e l'interruzione è supportata)
- N.B. sorgenti precedenti a titolo di esempio...