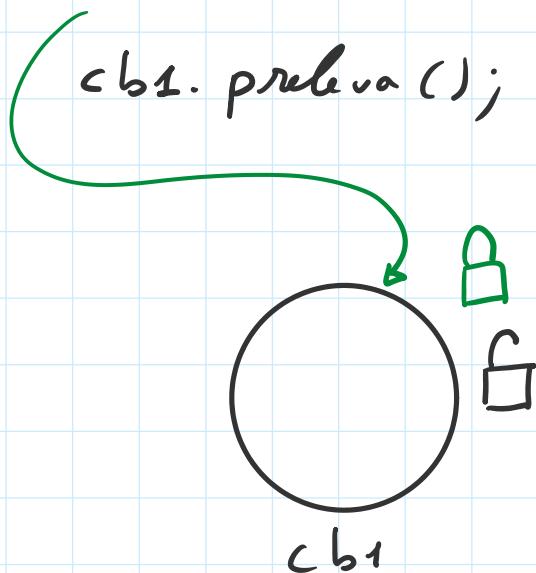


Un metodo synchronized cerca di prendere il lock dell'oggetto su cui viene invocato

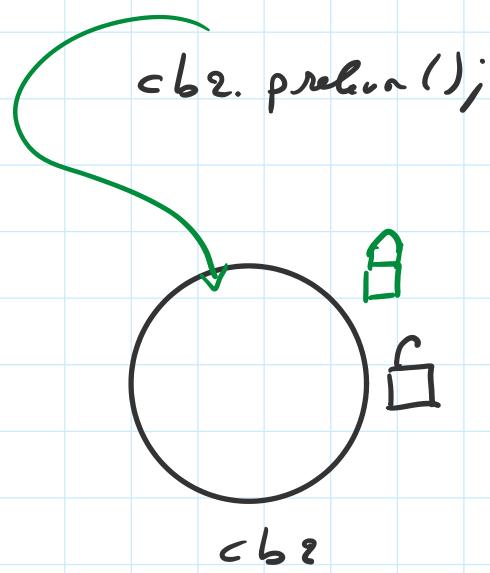
Supponiamo di avere

ContoBancario2 cb1 = new ContoBancario2( - );  
ContoBancario2 cb2 = new ContoBancario2( -- );

t1:



t2:



Oggetti diversi → locks diversi

I lock Java sono rientranti :

se un thread possiede il lock di un oggetto e prova a riprenderlo non si blocca

oggetto e prova a riprenderlo non si blocca

Esempio:

```
public class A {
```

```
    public synchronized void m1() {
```

=

```
    m2();
```

=

```
}
```



il lock viene rilasciato  
al termine

di m1()

```
    private synchronized void m2() {
```

=

```
}
```

```
}
```



l'acquisizione del lock  
avviene senza problemi  
(non c'è blocco)

t1:

```
A a = new A();
```

```
a.m1();
```

Nel caso di metodi synchronized il lock viene rilasciato sia

- in corso di esecuzione normale, quando termina l'esecuzione del corpo del metodo

- in corso di eventi anomali (eccezioni), quando si esce dal metodo.

## Blochi synchronized

la forma è

synchronized (oggetto) {

==

}

oggetto di cui vogliamo  
prendere il lock

==

blocco di istruzioni  
protetto dal lock  
in questione

Quando scriviamo

class B {

synchronized void m1() {

==

}

synchronized void m2() {

==

}

}

è lo stesso che scrivere

class B {

void m1() {

oggetto implicito

void m2() {

void m3() {

void m4() {

void m5() {

void m6() {

void m7() {

void m8() {

void m9() {

void m10() {

void m11() {

void m12() {

void m13() {

void m14() {

void m15() {

void m16() {

void m17() {

void m18() {

void m19() {

void m20() {

void m21() {

void m22() {

void m23() {

void m24() {

void m25() {

void m26() {

void m27() {

void m28() {

void m29() {

void m30() {

void m31() {

void m32() {

void m33() {

void m34() {

void m35() {

void m36() {

void m37() {

void m38() {

void m39() {

void m40() {

void m41() {

void m42() {

void m43() {

void m44() {

void m45() {

void m46() {

void m47() {

void m48() {

void m49() {

void m50() {

void m51() {

void m52() {

void m53() {

void m54() {

void m55() {

void m56() {

void m57() {

void m58() {

void m59() {

void m60() {

void m61() {

void m62() {

void m63() {

void m64() {

void m65() {

void m66() {

void m67() {

void m68() {

void m69() {

void m70() {

void m71() {

void m72() {

void m73() {

void m74() {

void m75() {

void m76() {

void m77() {

void m78() {

void m79() {

void m80() {

void m81() {

void m82() {

void m83() {

void m84() {

void m85() {

void m86() {

void m87() {

void m88() {

void m89() {

void m90() {

void m91() {

void m92() {

void m93() {

void m94() {

void m95() {

void m96() {

void m97() {

void m98() {

void m99() {

void m100() {

void m101() {

void m102() {

void m103() {

void m104() {

void m105() {

void m106() {

void m107() {

void m108() {

void m109() {

void m110() {

void m111() {

void m112() {

void m113() {

void m114() {

void m115() {

void m116() {

void m117() {

void m118() {

void m119() {

void m120() {

void m121() {

void m122() {

void m123() {

void m124() {

void m125() {

void m126() {

void m127() {

void m128() {

void m129() {

void m130() {

void m131() {

void m132() {

void m133() {

void m134() {

void m135() {

void m136() {

void m137() {

void m138() {

void m139() {

void m140() {

void m141() {

void m142() {

void m143() {

void m144() {

void m145() {

void m146() {

void m147() {

void m148() {

void m149() {

void m150() {

void m151() {

void m152() {

void m153() {

void m154() {

void m155() {

void m156() {

void m157() {

void m158() {

void m159() {

void m160() {

void m161() {

void m162() {

void m163() {

void m164() {

void m165() {

void m166() {

void m167() {

void m168() {

void m169() {

void m170() {

void m171() {

void m172() {

void m173() {

void m174() {

void m175() {

void m176() {

void m177() {

void m178() {

void m179() {

void m180() {

void m181() {

void m182() {

void m183() {

void m184() {

void m185() {

void m186() {

void m187() {

void m188() {

void m189() {

void m190() {

void m191() {

void m192() {

void m193() {

void m194() {

void m195() {

void m196() {

void m197() {

void m198() {

void m199() {

void m200() {

void m201() {

void m202() {

void m203() {

void m204() {

void m205() {

void m206() {

void m207() {

void m208() {

void m209() {

void m210() {

void m211() {

void m212() {

void m213() {

void m214() {

void m215() {

void m216() {

void m217() {

void m218() {

void m219() {

void m220() {

void m221() {

void m222() {

void m223() {

void m224() {

void m225() {

void m226() {

void m227() {

void m228() {

void m229() {

void m230() {

void m231() {

void m232() {

void m233() {

void m234() {

void m235() {

void m236() {

void m237() {

void m238() {

void m239() {

void m240() {

void m241() {

void m242() {

void m243() {

void m244() {

void m245() {

void m246() {

void m247() {

void m248() {

void m249() {

void m250() {

void m251() {

void m252() {

void m253() {

```

void m1() { oggetto imperativo
    synchronized (this) {
        = 
    }
}

void m2() {
    synchronized (this) {
        = 
    }
}

```

Ci sono alcune situazioni in cui non possiamo fare a meno dei blocchi synchronized

- 1) non tutto il codice di un metodo deve essere eseguito in mutua esclusione
- 2) sincronizzazione su oggetti diversi dall'oggetto impliato
- 3) sincronizzazione su un array

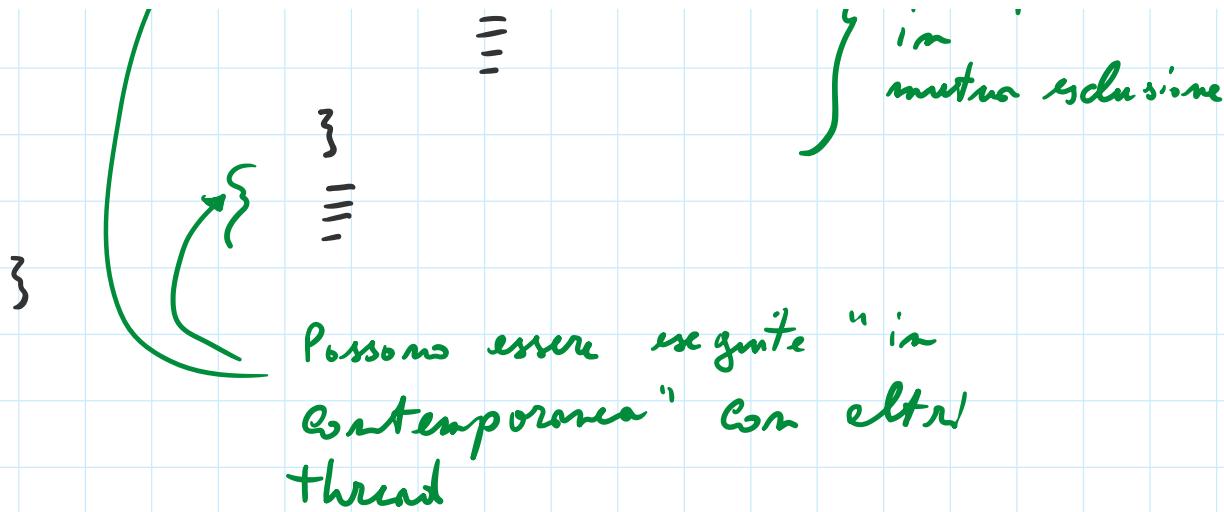
1)

```

public void m() {
    {
        = 
    }
    synchronized (this) {
        = 
    }
}

```

solo questi  
in  
mutua esclusione



2)

```
void m (Parametro p) {
    synchronized (p) {
        Codice che modifica p
    }
}
```

l'oggetto  
di cui si  
prende il  
lock è p

3)

```
int[] v = new int[~];
```

$t_1$  :

```
synchronized (v) {
```

$t_2$  :

```
synchronized (v) {
```

Esempio, più lock, ol'versi dall'oggetto impratico:

```
class Separat {  
    private int v1;  
    private int v2;  
    protected final Object l1 = new Object();  
    protected final Object l2 = new Object();
```

```
public int getV1 () {  
    synchronized (l1) {  
        return v1; } } // prendo solo il lock l1
```

```
public void setVs (int v) {  
    synchronized (l1) {
```

```
public int getV2() {  
    synchronized (l2) {  
        return v2;  
    }  
}
```

```
public void setV2(int v){  
    synchronized (l2){  
        v2 = v;  
    }  
}
```

```
public void makeEqual() {  
    synchronized (list) {
```

```

    synchronized (l1) {
        synchronized (l2) {
            v1 = v2;
        }
    }
}

```

Qui prende entrambi i lock

Potrò eseguire "in contemporaneo"

- `getV1()` con `setV2()` e `getV2()`
  - `setV1()` " "
  - `getV2()` " `setV1()` e `getV1()`
- ;

Questo aumenta il livello di parallelismo del codice

$\rho'$  è sotto esclusione tra

- `getV1` e `setV1` e `makeEqual`
- `setV2` e `getV2` e `makeEqual`

;

## Metodi statici sincronizzati

Quando un metodo statico è invocato come `synchronized` il lock è quello della classe (e non delle istanze della classe)

Questo significa che due thread non possono eseguire "in contemporanea" metodi statici sincronizzati sulla stessa classe.

Esempio:

```
public class ContoBancario {  
    private int numero;  
    private static int prossimoNumero;  
  
    public ContoBancario () {  
        numero = prossimoNumero + 1;  
    }  
    ;  
}
```

Se più thread possono creare istanze di Conto bancario possono venire fuori inconcististenze

Soluzione: aggiudiamo al campo statico prossimoNumero con un metodo statico synchronized.

```
public class ContoBancario {  
    private int numero;  
    private static int prossimoNumero;
```

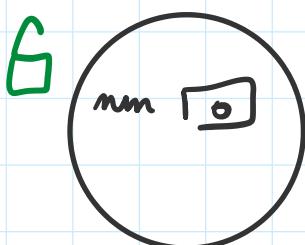
```

public ContoBancario() {
    numero = getNextAvailableNumber();
}
private static synchronized int
getNextAvailableNumber() {
    return nextAvailableNumber++;
}
;
}

```

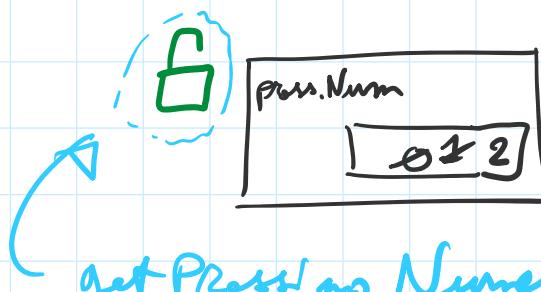
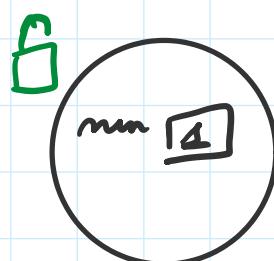
t<sub>1</sub>:

ContoBancario c<sub>1</sub> = new ContoBancario();



t<sub>2</sub>:

ContoBancario c<sub>2</sub> = new  
ContoBancario();



getNextAvailableNumber uses  
questo lock

Ci sono tre lock!

# Deadlock

Una situazione di blocco da cui non si riesce più ad uscire

Per potersi verificare ci devono essere almeno due thread e due lock

Per esempio

t<sub>1</sub>

:

acquisisce l<sub>1</sub>

:

tenta di

acquisire l<sub>2</sub>

↓  
si blocca

t<sub>2</sub>

:

acquisisce l<sub>2</sub>

:

tenta di acquisire l<sub>1</sub>

:

si blocca

:::::::

Lancia.java

:::::::

public class Lancia {

```
    public static void main(String[] args){  
        StatoGioco s = new StatoGioco();  
        ViteRimaste v = new ViteRimaste();  
        Mario m = new Mario(s, v);  
        Luigi l = new Luigi(s, v);  
        m.start();  
        l.start();  
    }  
}
```

:::::::

Luigi.java

:::::::

public class Luigi extends Thread {

```
    private StatoGioco sg;  
    private ViteRimaste vr;  
  
    public Luigi(StatoGioco sg, ViteRimaste vr) {  
        this.sg = sg;  
        this.vr = vr;  
    }  
    public void run(){  
        while(true) {  
            // altro codice  
            vr.controllaQuante(sg);  
        }  
    }  
}
```

```

}

:::::::
Mario.java
:::::::
public class Mario extends Thread {

    private StatoGioco sg;
    private ViteRimaste vr;

    public Mario(StatoGioco sg, ViteRimaste vr) {
        this.sg = sg;
        this.vr = vr;
    }
    public void run(){
        while(true) {
            // altro codice
            sg.controllaPunteggio(vr);
        }
    }
}
:::::::
StatoGioco.java
:::::::
public class StatoGioco {

    private boolean fine;

    public synchronized void controllaPunteggio(ViteRimaste vr){
        System.out.println(Thread.currentThread().getName() + " chiamato getPunteggio()");
        // il seguente metodo potrebbe essere chiamato quanto il punteggio
        // supera una certa soglia. Qui lo chiamiamo sempre per sollecitare
        // il problema
        vr.incrementa();
    }

    public synchronized void termina(){
        fine = true;
        System.out.println(Thread.currentThread().getName() + " chiamato termina()");
    }
}
:::::::
ViteRimaste.java
:::::::
public class ViteRimaste {

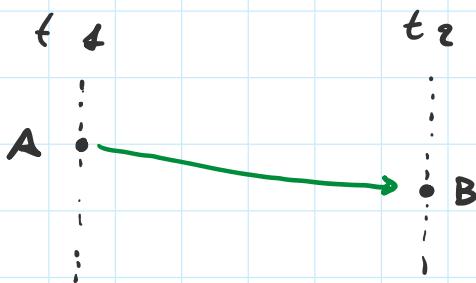
    private int q;

    public synchronized void incrementa(){
        // Anche qui supponiamo che ci sia del codice opportuno
        System.out.println(Thread.currentThread().getName() + " chiamato incrementa()");
    }

    public synchronized void controllaQuante(StatoGioco sg) {
        System.out.println(Thread.currentThread().getName() + " chiamato quante()");
        // termina() potrebbe essere chiamato quando il numero di vite diventa uguale a zero
        sg.termina();
    }
}

```

## Problema



Voglio eseguire l'operazione B in  $t_2$  solo dopo che  $t_1$  ha eseguito A

Comunicazione tra thread

- nel mondo della programmazione concorrente  
è detta sincronizzazione

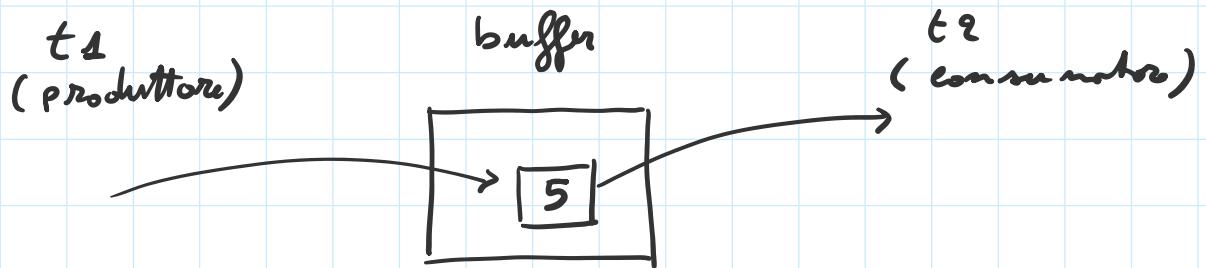
Il costrutto synchronized risolve i problemi di interferenza (mediante esecuzione in mutua esclusione).

I problemi di comunicazione (sincronizzazione) vengono risolti con i metodi

- `wait()`
- `notify()`
- `notifyAll()`

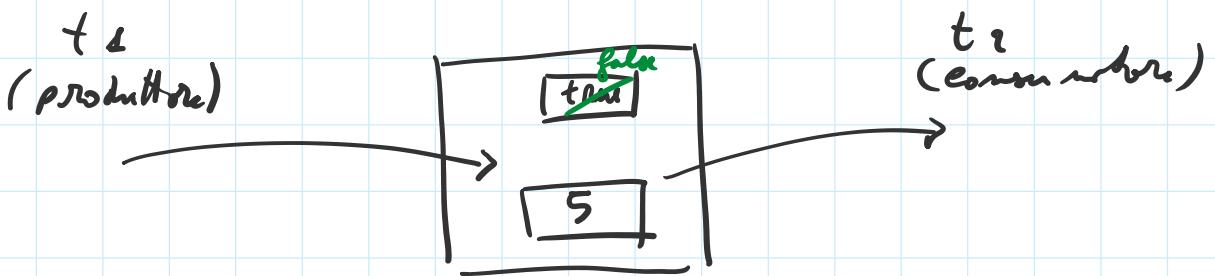
ereditati dalla classe `Object` e quindi disponibili in tutti gli oggetti

Esempio:



Come fa il produttore a capire se il valore precedente è già stato consumato?

Come fa il consumitore a capire se nel buffer c'è un nuovo valore?



I metodi `wait` e `notify` (`notifyAll`) consentono di sospendere la propria esecuzione e di inviare una notifica

`public final void wait() throws InterruptedException`  
il thread che invoca `wait`

- sospende la propria esecuzione
- rilascia il lock sull'oggetto
- riacquista automaticamente il lock quando esce dalla wait

public final void notify()

risveglia un thread bloccato su una wait

public final void notifyAll()

risveglia tutti i thread bloccati su una wait

Per poter chiamare i metodi wait, notify e notifyAll su un oggetto abbiamo possedere il lock di tale oggetto

Se non possediamo il lock generiamo

IllegalMonitorStateException

A ogn' oggetto è associato un "wait-set"

Quando un thread chiama wait su un oggetto il thread viene sospeso, rilascia il lock, viene inserito nel wait-set di tale oggetto

il lock, viene riassegnato nel wait-set dell'oggetto

Quando eseguo notify su un oggetto  
un thread a caso tra quelli nel wait-set  
riceve la notifica

- riacquista il lock automaticamente  
(eventualmente in competizione con  
altri thread che cercano di prendere  
lo stesso lock)
- viene riportato in coda pronta

Nel caso della notifyAll la notifica  
viene inviata a tutti i thread nel wait-set  
dell'oggetto

In effetti di wait ne esistono più versioni

pubblica void wait (long timeout) throws Interr.

il thread si blocca fino a che

- notify invocata su questo oggetto  
e il thread è selezionato (a caso)
- notifyAll invocata su questo  
oggetto
- scatta il timeout (millisecond)

- invoca il metodo interrupt sul thread

altra versione

void wait (long timeout, int nano)

void wait()

↳ corrisponde a  
wait(0)

+  
timeout infinito

Schemma tipico :

class X {

synchronized void faQuandoCondizione()  
throws InterruptedException {

while (! condizione)

wait();

// quello che devo fare

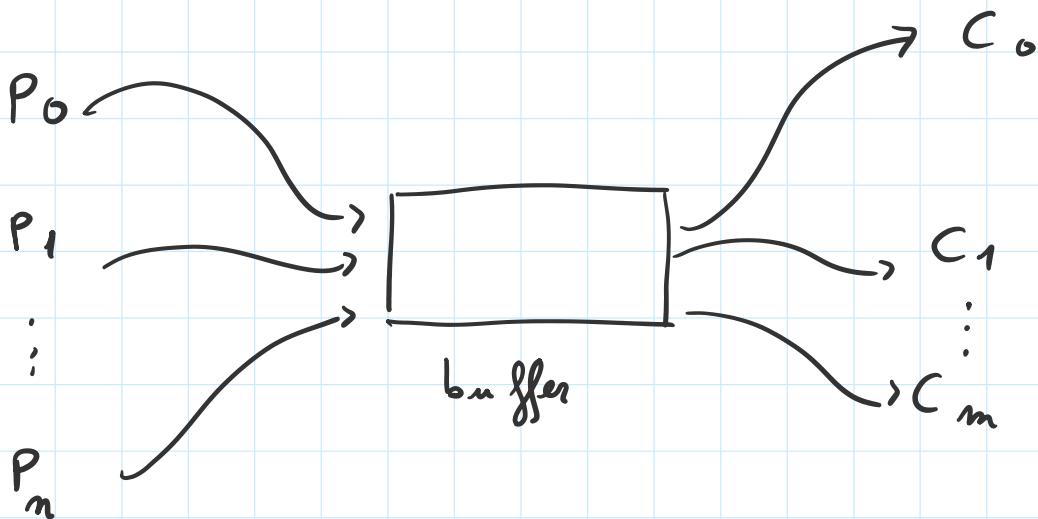
}

```

synchronized void cambiaCondizione() {
    // Cambia condizione
    notify(); // o notifyAll a second
    // del cont
}
}

```

### Produttori consumatori



Ogni produttore ciclicamente  
- produce un nuovo valore  
- lo immette nel buffer  
Se il buffer è pieno deve attendere

Ogni consumatore  
- tira fuori un valore dal buffer  
- lo consuma  
Se il buffer è vuoto deve attendere

buffer  
- è in grado di contenere un numero limitato di valori  
- è condiviso tra i produttori e i consumatori

```
public class Buffer {
```

```
private static final int DEFAULT_SIZE = 4;
```

*n° di valori che possono essere contenuti in buffer*  
*i valori trasferiti sono stralunati*

```
public class Buffer {
```

```
    private static final int DEFAULT_SIZE = 4;  
    private String[] vett;  
    private int testa, coda;  
    private int quanti;
```

```
    public Buffer(){  
        this(DEFAULT_SIZE);  
    }
```

```
    public Buffer(int n) {  
        vett = new String[n];  
    }
```

```
    public synchronized void inserisci(String s) throws InterruptedException {  
        while(pieno()) {  
            System.out.println(Thread.currentThread().getName() + ": mi blocco, buffer pieno");  
            wait();  
        }  
        vett[coda] = s;  
        coda = (coda + 1) % vett.length; // resto div. intera  
        quanti++;  
        notifyAll();  
    }
```

```
    public synchronized String estrai() throws InterruptedException {  
        while(vuoto()) {  
            System.out.println(Thread.currentThread().getName() + ": mi blocco, buffer vuoto");  
            wait();  
        }  
        String t = vett[testa];  
        testa = (testa + 1) % vett.length;  
        quanti--;  
        notifyAll();  
        return t;  
    }
```

```
    public synchronized boolean pieno() {  
        return quanti == vett.length;  
    }
```

```
    public synchronized boolean vuoto() {  
        return quanti == 0;  
    }
```

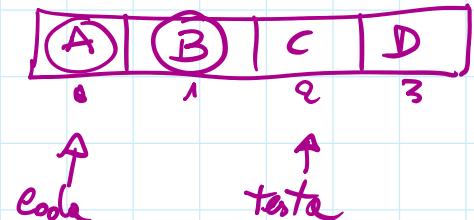
```
public class Produttore extends Thread {
```

```
    private Buffer b;
```

```
    public Produttore(String n, Buffer b) {  
        super(n);
```

i valori trasferiti sono stringhe

livello di riempimento  
utilizzato per le operazioni di estrazione e inserimento



```

public Produttore(String n, Buffer b) {
    super(n);
    this.b = b;
}

public void run(){
    int c = 0;
    try {
        while(true) {
            sleep((long)(Math.random()*1000));
            b.inserisci(getName() + ":" + c++);
            System.out.println(getName() + ": ho inserito");
        }
    } catch(InterruptedException ie) {
        System.out.println("Sono stato interrotto");
        return;
    }
}

```

*constructor di Thread*

*scrive a coda tra 0 e 1 (slow)*

*valore a coda (double)*

*dorme per un certo tempo (< 1 s)*

```

public class Consumatore extends Thread {

    private Buffer b;

    public Consumatore(String n, Buffer b) {
        super(n);
        this.b = b;
    }

    public void run(){
        try {
            while(true) {
                sleep((long)(Math.random()*1000));
                String s = b.estrai();
                System.out.println(getName() + ": ho estratto \"" + s + "\"");
            }
        } catch(InterruptedException ie) {
            System.out.println("Sono stato interrotto");
            return;
        }
    }
}

```

```

import java.util.Scanner;

public class Prova {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Quanti produttori? ");
        int p = sc.nextInt();
        System.out.println("Quanti consumatori? ");

```

```
int c = sc.nextInt();
System.out.println("Dimensione del buffer? ");
int b = sc.nextInt();

Buffer buf = new Buffer(b);
for(int i=0; i<p; i++)
    new Produttore("Produttore " + i, buf).start();
for(int i=0; i<c; i++)
    new Consumatore("Consumatore " + i, buf).start();
}
}
```