UNIVERSITÀ DEGLI STUDI DI BERGAMO
Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Esercitazione: Sincronizzazione indiretta/diretta in Java

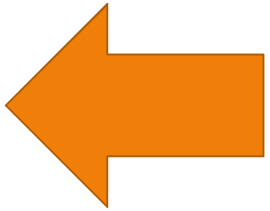
Corso di laurea in Ingegneria Informatica

- lock su oggetti e i metodi wait-notify-notifyAll
- Il problema del *Produttore-Consumatore*
- Esercizi (*Contatore*)

Prof.ssa Patrizia Scandurra

La sincronizzazione in Java

- Meccanismi di sincronizzazione
  1. **I semafori** -- API *java.util.concurrent*
  2. **Monitor mediante *lock su oggetti*** e i metodi ***wait / notify / notifyAll***
     - Detta anche *sincronizzazione indiretta / diretta*
  3. **Monitor mediante lock e condizioni**
     – API *java.util.concurrent*



Sui meccanismi di sincronizzazione...

Due aspetti da considerare:

1. Come il meccanismo realizza la **mutua esclusione** dei thread per accedere in modo esclusivo alle risorse?
   - Risorse fisiche o logiche (ad esempio, oggetti Java)?

2. Come il meccanismo realizza la **cooperazione** tra i thread?

La sincronizzazione indiretta/diretta dei thread in Java

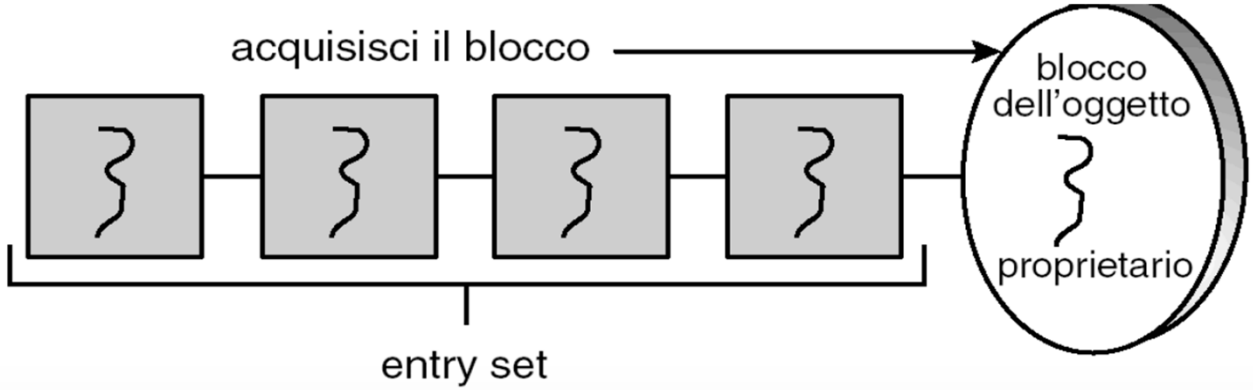
- Primo meccanismo di sincronizzazione di Java
  - Si ispira al concetto di *monitor*
  - Implementato **a livello di linguaggio (no API/libreria)!**
    - No package *java.util.concurrent*

- **Mutua esclusione (sincronizzazione indiretta)**
  - uso di **lock su oggetti** per l'accesso esclusivo ad oggetti condivisi (*sezione critica*)

- **Cooperazione (sincronizzazione diretta)**
  - uso dei metodi della classe Object: **wait() notify()** e **notifyAll()**

Mutua esclusione (o sincronizzazione indiretta)

- **In Java, ogni oggetto ha:**
  - **un blocco (*lock*)**
    - Ogni thread che deve effettuare delle operazioni in modo mutuamente esclusivo su un oggetto, deve prima acquisire il lock sull'oggetto e poi rilasciarlo quando ha terminato
    - Fintanto che un thread possiede il lock su un oggetto, nessun altro thread può acquisirlo
  - **una coda di attesa (*entry set*) per i thread**
    - I thread che tentano di acquisire il lock dell'oggetto ma già in possesso da un altro thread vengono bloccati in una coda detta *entry set*
    - Il primo thread presente nell'entry set può acquisire il lock, non appena questo viene rilasciato



La keyword **synchronized**

- Per acquisire il lock su un oggetto, il programmatore deve contrassegnare i **blocchi/metodi** che accedono all'oggetto condiviso come **synchronized**
  - eseguiti in modo *mutuamente esclusivo*
- Il lock viene rilasciato quando il thread esce dal blocco/metodo synchronized

```
public class SharedCounters {
    private Integer c1 = 0;
    private Integer c2 = 0;

    public void inc1() {
        sync
            //Sintassi:
            syncronized(object){
                ... //Sezione critica
            }
    }

    public v
        sync
        }
    }
}
```

Blocchi di codice synchronized sugli oggetti C1 e C2

```
public class SharedCounters {
    private Integer c1 = 0;
    private Integer c2 = 0;

    public synchronized void inc1() {
            c1++;
    }

    public synchronized void inc2() {
            c2++;
    }
}
```

Metodi synchronized

Metodi `synchronized`

- Un **metodo synchronized può chiamare un altro metodo synchronized** sullo stesso oggetto **senza provocare** il **blocco** del thread corrente
- Un **metodo synchronized ed un metodo non synchronized possono essere eseguiti concorrentemente** sullo stesso oggetto

Oggetti condivisi di classe con *attributi istanza* e *statici*

Esempio di riferimento per la sintassi di blocchi/metodi sincronizzati

- Consideriamo di condividere tra thread un oggetto della classe:

```
class OggettoCondiviso {
    static int iQuanti = 0;
    int conta;

    OggettoCondiviso(int conta) {
        this.conta = conta;
        iQuanti++;
    }

    void decrem(int dec) {
        conta-=dec;
    }
}
```

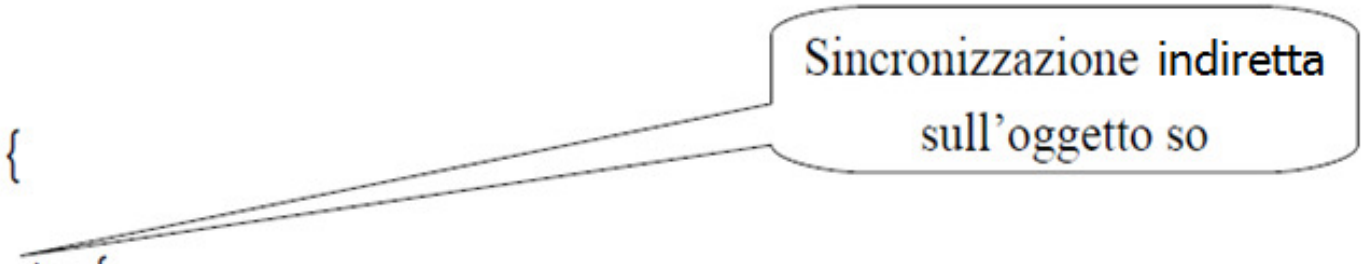
Sincronizzazione indiretta – blocco sincronizzato per attributi *istanza*

```
class MyRunnable implements Runnable {
    int iNum;
    OggettoCondiviso so;

    MyRunnable(int iNum, OggettoCondiviso so) {
        this.iNum = iNum;
        this.so = so;
    }

    public void run() {

        synchronized(so) {
            so.conta = so.conta + 7;
        }
    }
}
```



Sincronizzazione indiretta – blocco sincronizzato per attributi *static*

```
class MyRunnable implements Runnable {
    int iNum;
    OggettoCondiviso so;

    MyRunnable(int iNum, OggettoCondiviso so) {
        this.iNum = iNum;
        this.so = so;
    }

    public void run() {

        synchronized(OggettoCondiviso.class) {
            OggettoCondiviso.iQuanti++;
        }
    }
}
```

Sincronizzazione **indiretta** sulla classe OggettoCondiviso per campi static

Sincronizzazione indiretta – metodi sincronizzati per attributi *istanza*

```
class OggettoCondiviso {
    static int iQuanti = 0;
    int conta;

    OggettoCondiviso(int conta) {
        this.conta = conta;
        iQuanti++;
    }

    synchronized void decrem(int dec) {
        conta-=dec;
    }
}
```

Sincronizzazione **indiretta** sul metodo decrem: l'oggetto su cui avviene la sincronizzazione e' quello su cui viene invocato il metodo decrem

Sincronizzazione indiretta – metodi sincronizzati per attributi static

```
class OggettoCondiviso {
    static int iQuanti = 0;
    int conta;

    OggettoCondiviso(int conta) {
        this.conta = conta;
        iQuanti++;
    }

    static synchronized void decrQuanti() {
        iQuanti--;
    }
}
```

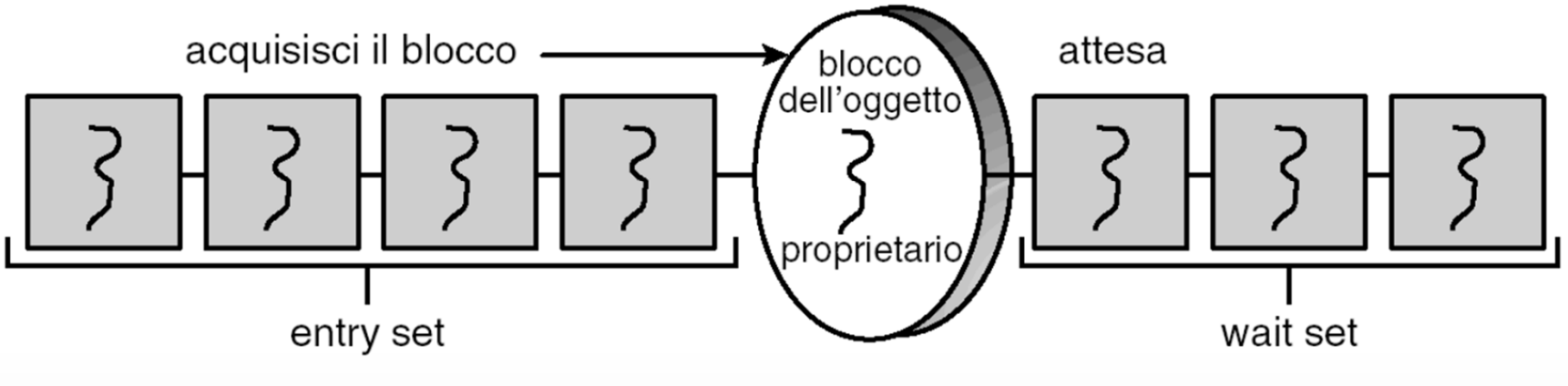
Sincronizzazione indiretta sul metodo static decrQuanti

Cooperazione (o sincronizzazione diretta)

- Cooperazione tra thread avviene tramite i metodi: **wait()** e **notify()** o **notifyAll()** da **invocare solo da un blocco/metodo synchronized**

Semantica:

- **wait()** mette il thread corrente in attesa in una coda *wait set* associata all'oggetto su cui aveva acquisito il lock (e che ora rilascia!), fino a che un altro thread che acquisisce il lock sul medesimo oggetto invoca il metodo notify() o notifyAll()



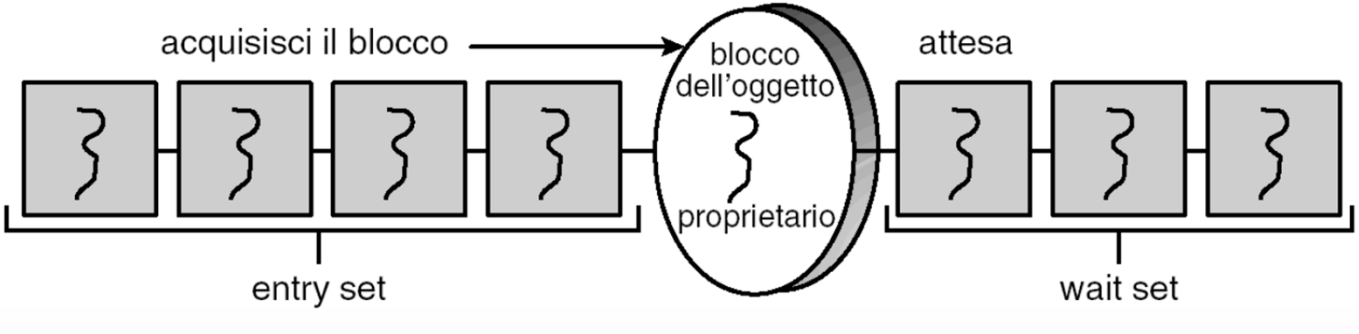
Cooperazione (o sincronizzazione diretta)

- **notify()** provoca le seguenti azioni:

- Viene selezionato arbitrariamente un thread T dal *wait set*

- T viene spostato dal *wait set* all'*entry set* e lo stato di T cambia da *waiting* a *blocking*

- T compete per il lock con gli altri thread dell'entry set



- **notifyAll()**: risveglia tutti i thread nel *wait set* e questi vengono aggiunti all'*entry set*

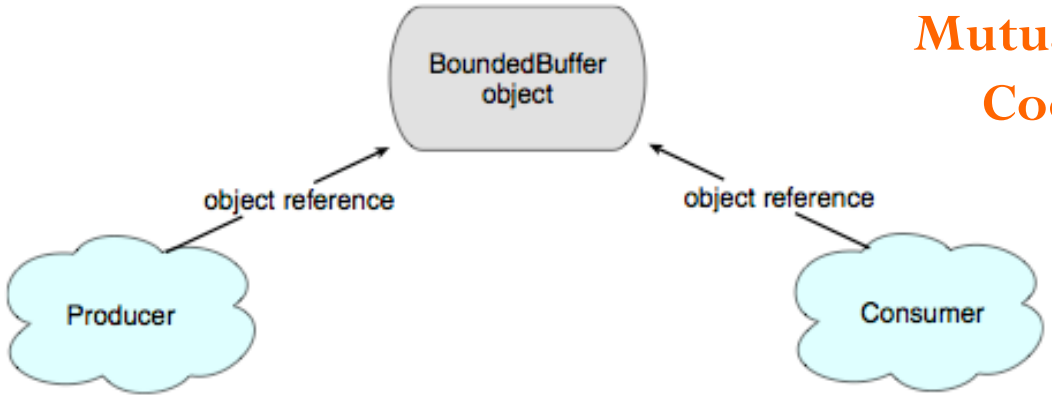
Esempio: il problema del produttore-consumatore

- Modello molto comune di cooperazione:
  - un processo (o thread) ***produttore*** *genera informazioni*
  - *che sono utilizzate da un* processo (o thread) ***consumatore***
- Un oggetto temporaneo in memoria (**buffer**) può essere riempito dal produttore e svuotato dal consumatore
  1. *buffer illimitato (unbounded-buffer): non c'è un limite teorico alla* dimensione del buffer
     - Il consumatore deve aspettare se il buffer è vuoto
     - Il produttore può sempre produrre
  2. *buffer limitato (bounded-buffer): la dimensione del buffer è fissata*
     - Il consumatore deve aspettare se il buffer è vuoto
     - Il produttore deve aspettare se il buffet è pieno

Il problema del produttore-consumatore (buffer limitato) in Java

```
public interface Buffer
{
    //  i produttori chiamano questo metodo
    public abstract void insert(Object item);

    // i consumatori chiamano questo metodo
    public abstract Object remove();
}
```



Mutua esclusione (sincr. indiretta)?
Cooperazione (sincr. diretta)?

<vedi cartella **boundedbuffer**>

Buffer limitato che usa la sincronizzazione in Java

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private int count, in, out;
    private Object[] buffer;
    public BoundedBuffer() { // il buffer è inizialmente vuoto
                count = 0;
                in = 0;
                out = 0;
                buffer = new Object[BUFFER SIZE];
    }
    public synchronized void insert(Object item) { // Vedi slide successiva
    }
    public synchronized Object remove() { // Vedi slide successiva
    }
}
```

Metodo insert() che utilizza wait/notify

```
public synchronized void insert(Object item) {
   while (count == BUFFER_SIZE) { //buffer pieno
            try { wait();}
            catch (Interru
   }
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
   notify();  //locazione piena
}
```

```
//Idioma per la sincr. diretta in metodo synchronized:
while (<condizione di attesa>){
      wait(); //attesa nel wait-set dell'oggetto
}
… //sezione critica e logica di cooperazione
notify();
```

Metodo remove() che utilizza wait/notify

```
public synchronized Object remove() {
    Object item;
    while (count == 0) { //buffer vuoto
            try { wait();}
            catch (InterruptedException e) { }
    }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    notify(); //locazione vuota
    return item;
}
```

Esercizio – sincronizzazione indiretta/diretta

- **Contatore sincronizzato**. Si progetti una applicazione Java che permetta a più thread di operare contemporaneamente su un oggetto condiviso di classe *Counter*, sfruttando opportunamente il modificatore *synchronized* per realizzare la mutua esclusione sull'oggetto condiviso.
  In particolare:
  - Un contatore (oggetto della classe Counter) è inizialmente a zero ed è in grado di contare fino a 10;
  - thread differenti devono poter incrementare/decrementare di una unità dallo stesso "contatore" (stesso oggetto di classe Counter) invocando il metodo increment()/decrement() della classe Counter*;* le operazioni di incremento/decremento devono essere possibili solo se mantengono rispettivamente il conteggio non superiore a 10/non negativo.
  - Per testare il corretto funzionamento dell'oggetto di classe Counter, si preveda un programma principale che istanzi un certo numero di thread figli: alcuni thread con ruolo "TaskI" che all'interno di un ciclo infinito si alternano tra dormire un pò (un certo num. random di ms) e incrementare il contatore; e altri thread con ruolo "TaskD" che all'interno di un ciclo infinito si alternano tra dormire un pò (un certo num. random di ms) e decrementare il contatore.