Semafori in Java

```
public class Semaphore {
 private int value, sospesi;
 public Semaphore (int v) {
  value=v; sospesi=0;}
 public synchronized void down () {
   if (value==0) {
      sospesi++;
      try { wait()};
      catch (InterruptedExeception e {}
      sospesi--;}
 else {
  value--; }
  }
 public synchronized void up () {
  if (sospesi>0)
      { notify(); }
         else{
         value++;}
 }
```


wait mette in attesa thread che lo esegue su quell'oggetto e poi rilascia il lock dell'oggetto sennò, altrimenti non potrei accedervi.

Nota: non mettere mai mani o occhi sulla variabile value. Altrimenti vista le considerazioni sulla sincronizzazione.

Monitor in java

- **Caratteristiche principali dei monitor Java**
  - **ME nell'esecuzione dei metodi synchronized**
  - **non ci sono variabili di condizione**
  - **wait(), notify() simili a sleep() , wakeup()**

I monitor e la sincronizzazione di thread in java

In Java, la sincronizzazione di thread per l'accesso concorrente ad oggetti condivisi si ottiene utilizzando i seguenti costrutti linguistici:

- **metodi *synchronized***
- **i metodi *wait() , notify()* e *notifyAll()***

Quando una classe *C* dichiara alcuni metodi come *synchronized* essa equivale ad un *monitor.* Se l'oggetto *o* è un'istanza di *C*, è garantita l'esecuzione in mutua esclusione (da parte di thread diversi che condividono l'oggetto) di tutti i metodi *synchronized* (sullo stesso oggetto *o*).
Le differenze principali rispetto ai monitor sono:

- **nella stessa classe possono convivere sia metodi *synchronized* sia metodi non *synchronized***
- **i metodi *wait()* e *notify()* corrispondono a *wait()* e *signal()* *ma* non sono associati ad una *condition* (esiste una sola coda di attesa)**

wait, notify

Sono metodi della classe *Object*:

```
public final void wait();                  /* rilascia il lock e si pone in wait */
public final void wait(long millis);       /* wait con timeout */
public final void notify();                /* rilascia il lock e sveglia un singolo thread*/
```

Posso avere una sola variabile di condizione: quella associata a oggetto monitor: esempio di produttore e cons. non va!

NOTA: Wait è sulla classe semaforo Semaforo è come se fosse variabile di condizione

Monitor e Semafori a confronto

I monitor costituiscono un meccanismo di sincronizzazione di più alto livello rispetto ai semafori, tuttavia qualsiasi problema di sincronizzazione risolto con i semafori può essere risolto con i monitor e viceversa: lo si dimostra facendo vedere come si può realizzare un semaforo attraverso i monitor e viceversa.

Realizzazione dei semafori tramite i monitor

```
type semaforo = monitor;

var semaforo_positivo:condition;
var val, sospesi:integer;

procedure entry init(valore_iniziale)
{   val = valore_iniziale; }

procedure entry down()
{ if (val==0)
    {sospesi = sospesi+1;
     semaforo_positivo.wait;
     sospesi = sospesi-1;}
 else { val = val –1;}
}

{ /*Inizializzazione */
   val = 0; sospesi = 0;
}
```

```
procedure entry up()
{
  if (sospesi>0) {
    semaforo_positivo.signal;}
  else
     { val = val +1;}
}
```

Uso dei monitor in java

```
public class ProducerConsumer {
    static final int N = 100;                  // constant giving the buffer size
    static producer p = new producer( );       // instantiate a new producer thread
    static consumer c = new consumer( );       // instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );  // instantiate a new monitor
    public static void main(String args[ ]) {
        p.start( );                            // start the producer thread
        c.start( );                            // start the consumer thread
    }
    static class producer extends Thread {
        public void run( ) {                   // run method contains the thread code
            int item;
            while (true) {                     // producer loop
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }    // actually produce
    }
    static class consumer extends Thread {
        public void run( ) {                   run method contains the thread code
            int item;
            while (true) {                     // consumer loop
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }    // actually consume
    }
```

Soluzione per Produttore/Consumatore in Java (parte 1)

Uso dei monitor in java

```
static class our_monitor {                 // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );    // if the buffer is full, go to sleep
        buffer [hi] = val;                 // insert an item into the buffer
        hi = (hi + 1) % N;                 // slot to place next item in
        count = count + 1;                 // one more item in the buffer now
        if (count == 1) notify( );         // if consumer was sleeping, wake it up
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );    // if the buffer is empty, go to sleep
        val = buffer [lo];                 // fetch an item from the buffer
        lo = (lo + 1) % N;                 // slot to fetch next item from
        count = count – 1;                 // one few items in the buffer
        if (count == N – 1) notify( );     // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
}
```

Soluzione per Produttore/Consumatore in Java (parte 2)

Se do wakeup un altro oggetto richiamando la wait, non rilascio il lock nel metodo synchronized

Su questa unica coda si accodano solo o consumatori o produttori. Quindi qui 1 coda basta.

Monitor e semaforo sono equivalenti