

# Coordinazione - Sincronizzazione

---

Nella scorsa lezione abbiamo visto come due thread, produttore e consumatore, debbano sincronizzarsi in modo tale da poter comunicare in modo corretto.

In java sono disponibili dei metodi della classe **Object**:

- **wait()**: sospende il thread invocante. Si sospende in un **wait set**, ovvero un insieme di thread associato a quell'oggetto.
- **notify()**: viene usata quando c'è stata una variazione di stato e all'invocazione di **notify()**, i thread sospesi per aver invocato **wait** vengono risvegliati.
  - **notifyAll()** consente di risvegliare **tutti i thread sospesi**.

Questi metodi sono presenti in ogni tipo di oggetto, perchè devono essere usati con l'obbiettivo di sospendere un thread se lo stato dell'oggetto non è tale per cui il thread possa andare avanti.

Siccome la valutazione viene fatta sullo stato dell'oggetto, il metodo sospensivo deve essere presente nella classe che usiamo per dare vita all'oggetto.

## SimpleBuffer - implementazione corretta

---

```

class SimpleBuffer{
    private int buf;
    private boolean dataAvailable = false;

    public synchronized int get() throws InterruptedException{
        while(!dataAvailable) wait();
        dataAvailable = false;

        notifyAll(); // oppure notify()
        return buf;
    }
}

```

La prima cosa da fare è aggiungere una variabile di stato booleana. Dobbiamo evitare che get restituisca in modo incondizionato il valore della variabile buffer, quindi prima di restituire deve verificare se può o meno restituirlo.

Se non è disponibile un nuovo dato, viene invocata **wait()**; nella nostra applicazione, è probabile che il consumatore venga sospeso, siccome esso usa la get().

```

class SimpleBuffer{
    private int buf;
    private boolean dataAvailable = false;

    // metodo get()

    public synchronized void put(int value) throws InterruptedException{
        while(dataAvailable) wait();
        buf = value;
        dataAvailable = true;
        notifyAll(); // oppure notify()
    }
}

```

put() viene scritto allo stesso modo; se dataAvailable è false, è possibile assegnare al buffer il valore, settare dataAvailable a true e quindi invocare notifyAll(),

Nel momento in cui viene invocato notify() il thread sospeso (nel nostro caso il consumatore) viene risvegliato.

## Che succede se risvegliamo il thread sbagliato?

E' possibile che con notifyAll() o notify() vengano risvegliati dei thread sbagliati.

Consideriamo una variante: invece di avere un unico consumatore ne abbiamo due; quando il produttore inserisce un dato nel buffer, risveglia tutti i consumatori in attesa che il nuovo dato venga inserito; questo vuol dire che **un solo consumatore** avrà la possibilità di accedere, mentre l'altro deve attendere.

Se all'interno del metodo get utilizzassimo un if, è chiaro che la condizione valutata dal secondo thread potrebbe essere tale per cui si esegue il codice che segue.

Quindi ogni volta che un thread viene risvegliato, poichè è possibile che le condizioni cambino, è necessario che la condizione di stato sia rivalutata, quindi il while serve per consentire al thread che si risveglia di verificare se effettivamente si è risvegliato per una condizione di stato a lui utile.

## Perchè è synchronized ?

Il metodo deve essere eseguito in mutua esclusione perchè altrimenti:

Se non fosse synchronized il thread produttore si sospende perchè non ci sono condizioni di stato utili per comunicare; viene poi risvegliato, ma siccome il codice non è eseguibile in maniera atomica, è possibile che la condizione di stato venga modificata da un altro thread (produttore); se il thread che si è risvegliato non ha la possibilità di eseguire in modo atomico il codice, è possibile che l'esecuzione di queste istruzioni sia realizzata con una condizione di stato che è stata però modificata.

## Pattern per wait

```
synchronized void method() InterruptedException{  
    while(!condition)  
        wait();  
    ...  
}
```

Dichiariamo il metodo sync e prima di eseguire delle istruzioni effettuiamo una verifica con un while loop su una variabile condition. Quello che stiamo dicendo è: **se non ci sono le condizioni per poter eseguire le istruzioni che seguono, aspetta.**

## Altro problema

Se il codice è eseguito in mutex ed il thread produttore si sospende, come facciamo in modo che un altro thread entri sullo stesso oggetto avendo previsto dei metodi synchronized?

**In altre parole:** siccome il metodo viene eseguito in mutex, quando un thread ne acquisisce il lock, nessun altro thread può accedervi. Cosa succede se però questo thread va in wait()?

**Questa situazione potrebbe determinare una situazione di stallo.**

## La soluzione

E' il comportamento della **wait()** ad essere importante; infatti quando il thread invoca la `wait()` e si sospende, poichè non può andare avanti ed l'oggetto potrebbe servire a qualcun altro, **la wait rilascia il lock**. Questo da la possibilità all'altro thread (consumatore nel nostro caso) di eseguire il codice di `get()`.

Ricordiamo che questo poteva essere un problema anche se l'altro thread voleva eseguire la `get()` (ed il primo thread era bloccato su `put()`), proprio perchè se un thread ha il lock su un metodo `synchronized`, ha il lock su tutti i metodi `sync` della classe.

---

fine teoria lezione 14

## Esercitazione 2020-10-23

---

### Esercizio 4.4

---

🚩 55:00 spiegazione esercizio - chat half duplex

### Esercizio 4.5

---

🚩 01:01 spiegazione esercizio

Nel client abbiamo il main dove viene creata un'istanza della classe, e poi viene avviato l'handler.

Nulla di nuovo, l'esercizio scritto era corretto.

## Esercizi 5.x - Esercizi sulla concorrenza

---

## Esercizio 5.1

Scrivere una classe di thread **Printer** il cui metodo `run()` scrive 100 volte un messaggio sullo `stdout`. Una volta scritta la classe dei thread vogliamo che siano istanziati 3 thread.

La traccia ci chiede di definire la classe dei thread in due modi diversi:

- Specializzazione della classe thread
- Implementazione dell'interfaccia `Runnable`

Come possiamo vedere i due thread vengono schedulati in maniera arbitraria.

## Esercizio 5.2

Scrivere un'applicazione di tipo produttore/consumatore vista durante la lezione 13 con e senza i meccanismi di sincronizzazione tra processi basati su `wait()` e `notify()`.

### Versione non sincronizzata

Si nota chiaramente come in questa versione *non si capisca nulla*, proprio perchè i due processi non sono sincronizzati, e leggono / scrivono a caso:

## Esercizio 5.3

---

Usare la concorrenza per scrivere un'applicazione client-server basato sul modello orientato ai flussi per calcolare il fattoriale di un numero. Il protocollo applicativo prevede lo scambio di due interi: uno è il numero di cui vogliamo calcolare il fattoriale mentre il secondo è il fattoriale calcolato.

Dobbiamo però cambiare l'implementazione realizzata nella prima parte dell'esercizio in modo da eseguire dal lato server sfruttando la concorrenza.

Lato server possiamo avere diversi effetti positivi; sia nel caso in cui abbiamo più processori, sia nel caso in cui non ne disponiamo.

Esercizio visto a 1:35

Possiamo scrivere l'applicazione con processi a singoli thread, ma il problema è che se un secondo client volesse contattare il server, dovrebbe attendere il completamento del primo thread prima di poter essere servito.

Dovremmo fare in modo che l'handler non sia bloccante, e quindi creare una **biforcazione del controllo** in modo tale da non bloccare il server.

## La soluzione

Invece di usare un handler che supporta un singolo thread, usiamo un handler modificato che ci permette di avviare l'esecuzione concorrente dei thread;

Ci basterà semplicemente estendere l'handler precedente, e creare un nuovo thread ogni volta che si invoca una nuova classe handler; ogni thread esegue l'handle() della superclasse.