Idiomatica: codice orribile, antipattern e pratiche discutibili

In questa rubrica analizzeremo i più comuni errori di codifica, di progettazione ed i tranelli in cui è facile cadere. Programmatore avvisato, mezzo salvato!

La massima di questo mese:

> "What we do not understand we do not possess"
> *Goethe*

#2: Sincronizzazioni che non si sincronizzano, o si sincronizzano troppo!

Agli albori di Java, una delle feature più importanti per chi decideva di passare a questo nuovo linguaggio era la sua capacità di poter gestire in maniera relativamente semplice – e ovviamente multipiattaforma – la programmazione concorrente.
Java aveva ed ha tuttora un supporto eccellente per la gestione di Thread multipli di esecuzione: dalla parola chiave *synchronized* alle concurrent utilities (inserite in java 5) il supporto alla programmazione concorrente in java è sempre stato – in teoria – alla portata di tutti.
Ma la realtà è che molti *pensano* di sapere cosa faccia realmente synchronized: e, di solito, si sbagliano di grosso... personalmente in questi anni ne ho sentite di tutti i colori, e spesso anche da persone apparentemente molto preparate.

Il cammino tipico di un programmatore ignaro delle infide trappole che si celano dietro alla programmazione concorrente è più o meno:

- non sincronizzare nulla, ignari di essere in un ambiente inerentemente multithread (servlet/jsp, per esempio)
- all'apparire di bug misteriosi, sincronizzare metodi più o meno a caso
- al persistere dei bug, sincronizzare tutti i metodi, tanto per sicurezza
- una volta che il bug è apparentemente risolto, sincronizzare solo gli accessi in scrittura, perchè le letture tanto sono in read only e così si ottimizza un po'
- al riapparire di bug che sembravano risolti, prendersela con la VM o con l'application server

Seriamente, ecco un semplice test per verificare se si è capito il meccanismo di sincronizzazione di Java:

- I metodi statici e quelli non statici, se sincronizzati, sono fra loro in mutua esclusione?
- Un costruttore può essere dichiarato synchronized?
- Se un thread A esegue lo statement a = 42 all'istante T0, è sicuro che il thread B in un istante successivo T1 > T0 "veda" il valore 42 in a?

Per ogni si risposto, un punto di penalizzazione!

La parola chiave synchronized fa in realtà tre cose diverse:

- Si occupa dell'atomicità di una operazione: semplicemente fa si che la VM cerchi di acquisire il lock dell'oggetto prima di "eseguire" il metodo (o il blocco di codice), acquisendolo se libero o mettendosi in attesa se no. Ma, bada bene, il lock è

dell'oggetto: non del metodo, ne' della classe. E' un attributo della classe Object, per cui ogni istanza ha il suo. Questo implica diverse cose:
    - due metodi sincronizzati di una stessa classe sono in mutua esclusione fra loro, se eseguiti sullo stesso oggetto.
    - Un metodo statico sincronizzato ed uno non statico sincronizzato non sono fra loro in mutua esclusione, perchè usano lock diversi. Il metodo statico (che, ricordiamoci, non ha il this), si sincronizza sul lock dell'oggetto di classe Class, quello non statico sull'istanza relativa.
    - Metodi sincronizzati non statici, eseguiti su istanze diverse, non sono fra loro in mutua esclusione
    - Un costruttore non ha senso che sia sincronizzato, perchè l'oggetto ancora non esiste (lo si sta costruendo), e dunque neanche il lock relativo. In effetti, non compila neanche, ma questo è marginale
- Si occupa della visibilità delle operazioni eseguite fra diversi thread
- Si occupa di garantire l'ordinamento delle operazioni eseguite da un thread dal punto di vista di un altro thread che “osserva”.

Se il primo punto è più o meno conosciuto, anche se spesso male interpretato, gli ultimi due sono ai più del tutto misteriosi. In caso di insufficiente sincronizzazione è perfettamente possibile che accadano le cose più strane, per esempio che il seguente codice abbia un loop infinito

```
class MyThread extends Thread {

    public void run() {
      while (!done) {
        // fai qualcosa ...
      }
    }

    public synchronized void setDone() {
      done = true;
    }

    private boolean done;

}
```

Regola: vanno sincronizzate anche le letture, e non solo le scritture, altrimenti non è affatto assicurato che un thread che cerca di leggere la variabile *done* abbia visibilità di una precedente scrittura della stessa da parte di un altro thread. Il perchè di questo comportamento, apparentemente strano, va cercato nei meandri del *Java Memory Model*. Ma avendo finito lo spazio a disposizione in questa rubrica, mi vedo costretto a rimandare ulteriori spiegazioni ad un prossimo approfondimento!

Esercizi:

- Andare a cercare eventuali servlet con il *doGet()* sincronizzato
- Andare a cercare eventuali servlet con il *SingleThreadModel*

Conclusioni:

Scrivere codice multithreaded è difficile ed è una cosa nota. Scrivere codice semplicemente

thread safe non è così difficile, ma non è neanche così banale come potrebbe sembrare ad una prima, superficiale analisi: il meccanismo di sincronizzazione va compreso a fondo per evitare gli errori più comuni ed ottenere il meglio da Java.