

Java sincronizzazione

Esercitazioni.2

A.Memo — febbraio 2004

bibliografia: Java, Patrick Naughton e Herbert Schildt

Sincronizzazione: perché?

- dopo aver creato più thread concorrenti, occorre regolamentare le loro interazioni
- interazione involontaria o **competizione**
 - garantire l'accesso mutuamente esclusivo alle risorse unarie
- interazione cercata o **cooperazione**
 - cooperazione a variabili condivise
 - cooperazione a scambio di messaggi

Tipi di interazioni

tipo di interazione		esempio	meccanismo di interazione
interazione involontaria	competizione	due processi tentano di accedere alla medesima periferica	sincronizzazione al momento dell'accesso alla risorsa condivisa
interazione indiretta	cooperazione a variabili condivise	un processo produce risultati indispensabili per l'avanzamento di un altro processo	sincronizzazione quando lo decide il programmatore
interazione diretta	cooperazione a scambio messaggi	un processo comunica ad un altro il raggiungimento di una particolare situazione	

Meccanismi di sincronizzazione

- | | | |
|--------------------|---|-----------------------------|
| soluzioni hardware | { | • disabilitazione interrupt |
| | | • istruzioni dedicate |
| soluzioni software | { | • variabile lucchetto |
| | | • alternanza stretta |
| soluzioni di S.O. | { | • semaforo binario (mutex) |
| | | • semaforo contatore |
| | | • monitor |
| | | • scambio messaggi |

sincronizzazione in Java

- tutte le classi, e quindi tutti gli oggetti istanziati, hanno un proprio monitor implicito, e per distinguere i metodi pubblici che devono soddisfare la mutua esclusione si utilizza il modificatore *synchronized*
- le successive invocazioni vengono messe in attesa
- il thread che ha invocato il metodo *synchronized* detiene esclusivamente il monitor fino al termine dell'esecuzione di quel metodo

Sincronizzazione in Java (2)

i metodi per gestire la sincronizzazione in Java sono essenzialmente:

- **wait()** pone in attesa il thread, e liberalizza l'accesso al monitor
- **notify()** risveglia il thread in attesa

Attenzione: possono essere invocati solo dal thread che possiede il controllo del monitor cui si riferiscono, quindi o dall'interno del metodo *synchronized* o da un altro metodo *static synchronized* della sua stessa classe

```
class Chiamata { // classe con un solo metodo di stampa in due fasi
    public void chiama(String testo) {
        System.out.print("[" + testo); // fase 1 di stampa
        try {                          // intervallo di pausa
            Thread.sleep(1000);
        } catch (InterruptedException e) { };
        System.out.println("]"); // fase 2 di stampa
    }
}
```

oggetto Chiamata che possiede il solo metodo Chiama che stampa un testo.
output dell'oggetto singolo: **[testo]**

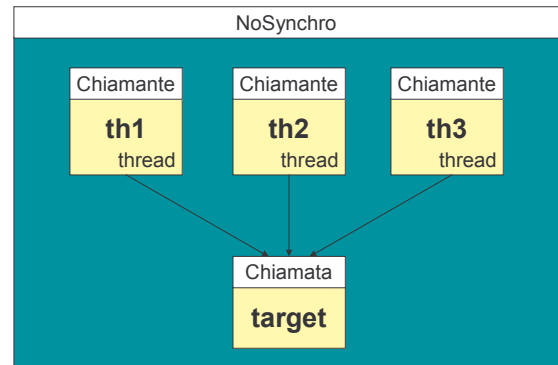
```
class Chiamante implements Runnable { // classe per thread multipli
    String msg; // testo da visualizzare
    Chiamata target; // oggetto per visualizzare msg
    Thread t;
    public Chiamante(Chiamata targ, String testo) {
        target = targ;
        msg = testo;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        target.chiama(msg);
    }
}
```

creo oggetti Chiamante di tipo thread che invocano il metodo chiama di un oggetto Chiamata che stampa un testo

```

class NoSynchro {
    public static void main(String args[] ) {
        Chiamata target = new Chiamata( );
        Chiamante th1 = new Chiamante(target,"Ciao");
        Chiamante th2 = new Chiamante(target,"Mondo");
        Chiamante th3 = new Chiamante(target,"Sincronizzato");
        try {
            th1.t.join( );
            th2.t.join( );
            th3.t.join( );
        } catch (InterruptedException e) { };
    }
}

```



output ottenuto

```

[Ciao [Mondo [Sincronizzato]
]
]

```

```

[Ciao]
[Mondo]
[Sincronizzato]

```

output desiderato

```

class ChiamanteSyn implements Runnable { // classe per thread multipli
    String msg; // testo da visualizzare
    Chiamata target; // oggetto per visualizzare msg
    Thread t;
    public ChiamanteSyn(Chiamata targ, String testo) {
        target = targ;
        msg = testo;
        t = new Thread(this);
        t.start( );
    }
    public void run() {
        synchronized(target) {
            target.chiama(msg);
        }
    }
}

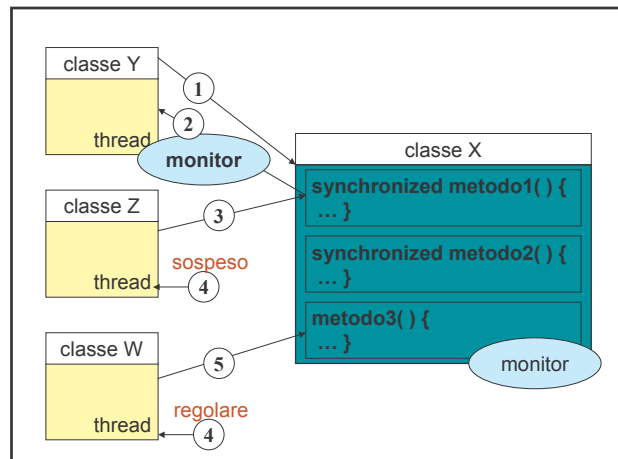
```

oltre ai metodi, posso sincronizzare anche gli oggetti

```

class Synchronro {
    public static void main(String args[] ) {
        Chiamata target = new Chiamata();
        ChiamanteSyn th1 = new ChiamanteSyn(target,"Ciao");
        ChiamanteSyn th2 = new ChiamanteSyn(target,"Mondo");
        ChiamanteSyn th3 = new ChiamanteSyn(target,"Sincronizzato");
        try {
            th1.t.join( );
            th2.t.join( );
            th3.t.join( );
        } catch (InterruptedException e) { };
    }
}

```



Esercizio 1

Versione semplice del produttore-consumatore

Un thread (Producer) produce dati (dallo 0 al 9, ciclicamente) e li memorizza in una variabile di scambio. Un thread (Consumer) legge i dati dalla variabile scambio e li visualizza.

Il consumatore deve leggere una sola volta il dato della variabile scambio. Produttore e consumatore impiegano un tempo casuale `sleep((int)(Math.random()*100))`; per generare e consumare il dato.

La variabile condivisa sia un oggetto di una classe, cui si accede solo tramite due metodi: `get()` e `put()`.

Stallo

```

public class Deadlock {
    public static void main(String[] args){
        final Object resource1 = "resource1";
        final Object resource2 = "resource2";
        Thread t1 = new Thread() {
            ..... // slide seguente
        };
        Thread t2 = new Thread(){
            ..... // slide seguente
        };
        t1.start();
        t2.start();
    }
}

```

adattato da "The Java Tutorial" Second Edition by Campione, M. Walrath, K Addison-Wesley 1998

```

Thread t1 = new Thread( ){
    public void run( ){
        synchronized(resource1){
            System.out.println("Thread 1: locked resource 1");
            try{
                Thread.sleep(50);
            } catch (InterruptedException e){ }
        } synchronized(resource2)
        System.out.println("Thread 1: locked resource 2");
    }
}

Thread t2 = new Thread( ){
    public void run( ){
        synchronized(resource2){
            System.out.println("Thread 2: locked resource 2");
            try{
                Thread.sleep(50);
            } catch (InterruptedException e){ }
        } synchronized(resource1)
        System.out.println("Thread 2: locked resource 1");
    }
}

```

Esercizio 2/A

Accesso ad una banca

Un oggetto della classe Banca è caratterizzato da un vettore di conti correnti, inizializzati a diversi valori, e possiede un metodo `transfer()` per spostare denaro da un conto ad un altro.

Il metodo `transfer()` deve essere in grado di soddisfare più richieste contemporaneamente, in maniera sicura

Esercizio 2/B

Accesso ad una banca – conto in rosso

Nell'esercizio precedente, se l'importo posseduto dal conto prima del trasferimento è inferiore, il trasferimento viene annullato. Modificare il metodo in modo che gestisca il conto in rosso: il trasferimento di prelievo non soddisfabile viene momentaneamente sospeso e riprende solo dopo che è arrivato un nuovo trasferimento di versamento su quel conto.