Programmazione Orientata agli Oggetti in Linguaggio Java

Tecniche di Programmazione: Thread Parte b

versione 1.0

Questo lavoro è concesso in uso secondo i termini di una licenza Creative Commons
(vedi ultima pagina)

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

- Sincronizzazione
- Blocchi
- Condizioni
- Sintassi Java Tradizionale
- Stati di un Thread

Sincronizzazione

- Sincronizzazione
  - ⇨meccanismo sulla base del quale viene regolato l'accesso dei thread alle risorse (oggetti) condivise
- Perchè è necessaria la sincronizzazione ?
  - ⇨per evitare che l'applicazione raggiunga stati inconsistenti
  - ⇨per evitare che il comportamento dipenda dall'ordine di esecuzione dei thread

Sincronizzazione

- Il problema delle corse
  - ⇨quando due thread diversi eseguono un metodo che aggiorna lo stato di un oggetto il risultato dipende in generale dall'ordine di esecuzione tra i due thread
  - ⇨si dice che si verifica una "corsa" tra i due thread che può produrre il fenomeno della "perdita di aggiornamento"

Sincronizzazione

- Nota
  - ⇨ si tratta di un problema molto più generale
  - ⇨ che riguarda qualsiasi contesto di esecuzione concorrente in cui i processi condividono risorse
- Vediamo un esempio classico
  - ⇨ due thread che aggiornano il valore di una proprietà pubblica X dell'oggetto o
  - ⇨ o.X = o.X + 1;

Sincronizzazione

- Dove nasce il problema
  - ⇨ l'istruzione o.X = o.X + 1; viene tradotta in bytecode in un blocco di istruzioni
  - ⇨ in pseudocodice macchina:

```
load R1 o.X    // carica in R1 il valore di o.X
add R1 1       // aggiungi a R1 1
store R1 o-X  // salva il valore di R1 in o.X
```

  - ⇨ non è detto che il blocco di istruzioni venga eseguito in maniera "atomica" dal thread, che potrebbe essere interrotto a metà

Sincronizzazione

- Vediamo due diversi casi

thread 1	o.X	thread 2
load X R1	3	
add R1 1	3	
store R1 X	4	
	4	load X R2
	4	add R2 1
	5	store R2 X

esecuzione seriale
risultato corretto

thread 1	o.X	thread 2
load X R1	3	
add R1 1	3	
	3	load X R2
	3	add R2 1
store R1 X	4	
	4	store R2 X

esecuzione concorrente, corsa
perdita di aggiornamento

Sincronizzazione

- Atomicità
  - ⇨l’esecuzione atomica delle operazioni (“tutto assieme o niente”) garantisce la correttezza
  - ⇨non è possibile che i thread interferiscano
  - ⇨l’esecuzione non atomica introduce inconsistenze nel caso di esecuzioni concorrenti
  - ⇨e richiede di introdurre tecniche di sincronizzazione per ottenere l’atomicità

Sincronizzazione

- Un caso più concreto in Java
  - ⇨ simuliamo il sistema informativo di una Banca
- Idea
  - ⇨ c'è una base di dati – che per semplicità rappresentiamo con un oggetto Banca
  - ⇨ che mantiene una lista di conti correnti
  - ⇨ ci sono diverse filiali che effettuano operazioni sui conti correnti dalla rete

Sincronizzazione

- Simuliamo la filiale
  - ⇨ con un thread che effettua numerose operazioni sui conti correnti della banca
  - ⇨ in particolare ci concentriamo sui bonifici e trasferiamo ogni volta una cifra a caso da un conto di partenza a caso ad un conto destinazione a caso
  - ⇨ naturalmente il saldo complessivo della banca dovrebbe restare immutato

Sincronizzazione

>> bancathread.nonsincronizzato

- bancathread.nonsincronizzato
  - ContoCorrente.java rappresenta un conto
  - Banca.java rappresenta la banca; viene condivisa da tutte le filiali
  - Filiale.java rappresenta una filiale; le operazioni di ciascuna filiale vengono eseguite lungo un thread separato
  - Principale.java avvia l’esecuzione

Sincronizzazione

- In sintesi
  - le corse tra i due thread provocano perdite di aggiornamento
  - è possibile che un thread di filiale venga interrotto mentre sta eseguendo l’operazione versa oppure l’operazione preleva
  - e che quindi il valore finale del saldo non sia corretto

Blocchi ("Lock")

- Come risolvere il problema
  - ⇨la soluzione standard è impedire che due thread possano lavorare concorrentemente su una stessa risorsa
  - ⇨per farlo si utilizzano blocchi ("lock")
- Idea
  - ⇨prima di effettuare operazioni su una risorsa condivisa è necessario "bloccarla"

Blocchi ("Lock")

thread 1	X	thread 2
lock X	3	
load X R1	3	
add R1 1	3	
store R1 X	4	
unlock X	4	
	4	lock X
	4	load X R2
	4	add R2 1
	5	store R2 X
	5	unlock X

esecuzione seriale
risultato corretto

thread 1	X	thread 2
lock X	3	
load X R1	3	
	3	lock X....
add R1 1	3	
store R1 X	4	
unlock X	4	
	4	load X R2
	4	add R2 1
	5	store R2 X
	5	unlock X

esecuzione concorrente
risultato corretto

Blocchi ("Lock")

>> java.util.concurrent.locks.Lock JavaDoc

- In java
  - a partire da J2SE 5.0 esiste un package esplicitamente dedicato alla sincronizzazione
  - java.util.concurrent
  - java.util.concurrent.locks fornisce il supporto alle operazioni di locking
- Lock di Java
  - oggetto di tipo java.util.concurrent.locks.Lock

Blocchi ("Lock")

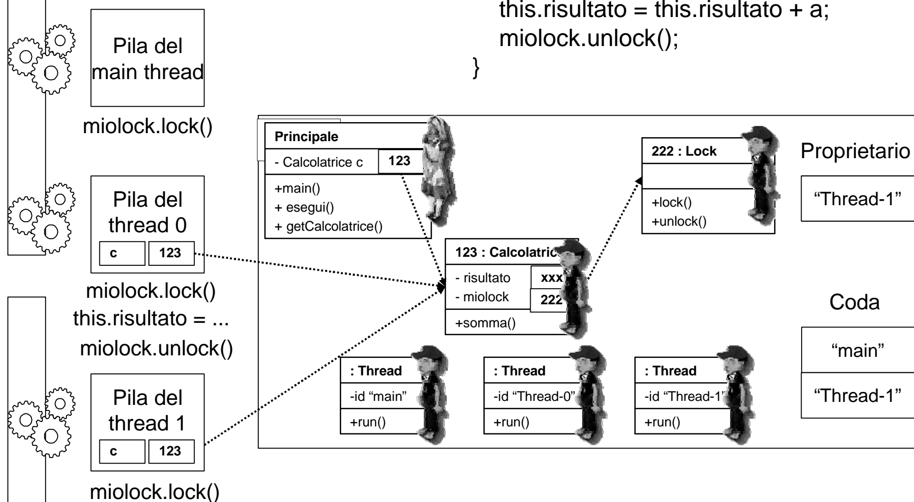
- Cos’è un lock di Java
  - oggetto con due metodi fondamentali
  - lock() e unlock()
  - ed una struttura di dati (coda) associata
  - il thread che chiama lock() blocca l’oggetto
  - tutti i thread successivi che chiamano lock() vengono introdotti in una coda
  - quando il lock viene rilasciato con unlock(), il lock risveglia uno degli oggetti della coda

Blocchi ("Lock")

- Come sincronizzare sezioni di codice
  - nel caso in cui un oggetto ponga problemi di sincronizzazione, può dichiarare un oggetto miolock di tipo Lock tra le sue proprietà
  - prima di un blocco di codice "pericoloso", viene aggiunta l'istruzione miolock.lock()
  - al termine del blocco di codice pericoloso viene aggiunta l'istruzione miolock.unlock()

Blocchi ("Lock")

```
public void somma(double a) {
    miolock.lock();
    this.risultato = this.risultato + a;
    miolock.unlock();
}
```



Blocchi ("Lock")

- Semantica del lock()
  - per eseguire la regione sincronizzata il thread deve acquisire il lock
  - se il thread si interrompe mantiene il lock
  - altri thread devono attendere che lo rilasci prima di cominciare l'esecuzione
  - altri thread possono però eseguire altri metodi dell'oggetto che non contengono regioni sincronizzate

Blocchi ("Lock")

- Utilizzo corretto di lock() e unlock()
  - è indispensabile che unlock() sia specificato in un blocco finally
  - per restituire il lock nel caso in cui si verifichi un'eccezione durante l'esecuzione del metodo

```
public void somma (double a) {
    miolock.lock()
    try {
        this.risultato += a;
    } finally {
        miolock.unlock();
    }
}
```

regione sincronizzata (contenuta nel blocco try)

Blocchi ("Lock")

- Come creare il lock
  - ⇨java.util.concurrent.locks.Lock è un’interfaccia
  - ⇨con varie implementazioni
- L’implementazione più usata
  - ⇨ReentrantLock
  - ⇨se un metodo che possiede il lock esegue di nuovo miolock.lock() NON deve riacquisire il lock (altrimenti bloccherebbe sè stesso)

Blocchi ("Lock")

- Nel caso della banca
  - ⇨varie operazioni da sincronizzare
  - ⇨il prelevamento non avviene in modo atomico > richiede sincronizzazione
  - ⇨il versamento non avviene in modo atomico > richiede sincronizzazione
  - ⇨il bonifico non avviene in modo atomico > richiede sincronizzazione

Blocchi ("Lock")

>> bancathread.sincronizzato

- Ma...
  - ⇨una volta sincronizzate queste operazioni è necessario sincronizzarne anche altre
- Regola generale
  - ⇨ogni proprietà sincronizzata in scrittura deve essere sincronizzata anche in lettura
  - ⇨per evitare che venga letta mentre è in uno stato inconsistente per via di una scrittura parziale

Blocchi ("Lock")

- Attenzione
  - ⇨la sincronizzazione ha vantaggi e svantaggi
- Vantaggio
  - ⇨garantisce la correttezza
- Svantaggi
  - ⇨riduce le prestazioni perchè serializza l'esecuzione
  - ⇨può provocare potenziali stalli

Blocchi ("Lock")

- Stallo
  - situazione in cui due thread diversi aspettano l’uno un lock posseduto dall’altro
- Esempio: due bonifici dal conto 10 al 12
  - filiale uno blocca il conto n. 10 e preleva
  - filiale due blocca il conto n. 12 e preleva
  - filiale uno richiede il blocco sul conto 10 ma deve aspettare (il conto è bloccato da filiale uno)
  - filiale due richiede il blocco sul conto 12 ma deve aspettare (il conto è bloccato da filiale due)

Tecniche di Programmazione: Thread >> Blocchi ("Lock")

Blocchi ("Lock")

- In Java
  - è il programmatore a dover gestire le potenziali situazioni di stallo
- Nel caso del nostro esempio
  - è indispensabile sincronizzare il metodo bonifico in modo che sia l’oggetto Banca il punto di sincronizzazione
  - i bonifici vengono effettuati uno alla volta e non ci sono problemi di stallo

Condizioni

- Schematizziamo il meccanismo di locking
  - ⇨un thread richiede il lock
  - ⇨ma il lock è bloccato
  - ⇨il thread entra nella coda, comincia ad attendere ("wait") e viene sospeso
  - ⇨il lock lo risveglia segnalandogli che il lock si è liberato ("signal") e può risvegliarsi

Condizioni

- Questo meccanismo
  - ⇨può essere reso più sofisticato
  - ⇨spesso un thread ha bisogno, dopo aver acquisito il lock, di verificare una condizione
  - ⇨se la condizione non è verificata, non può procedere e quindi il lock non gli serve
- Nel nostro esempio
  - ⇨il prelevamento dovrebbe essere effettuato solo se il saldo attuale lo consente

Condizioni

- In particolare
  - ⇨ è necessario che i thread comunichino
  - ⇨ un thread che riceve il lock verifica se il saldo è sufficiente
  - ⇨ se non lo è rinuncia al lock e si mette in attesa
  - ⇨ non appena un altro thread versa soldi sul conto, segnala a tutti i thread in attesa di verificare se la condizione è verificata

Condizioni

- Il meccanismo di sincronizzazione di Java
  - ⇨ consente di gestire questi problemi utilizzando oggetti di tipo Condition
- java.util.concurrent.locks.Condition
  - ⇨ oggetto a cui è associata una coda di thread in attesa (tipicamente di una condizione)
  - ⇨ altri thread possono risvegliare i thread nella coda, chiedendo all’oggetto di inviare un segnale

Condizioni

- Per ottenere un oggetto Condition
  - ⇨è possibile crearlo da un oggetto Lock
  - ⇨attraverso il metodo newCondition()
  - ⇨la condizione è collegata al lock
- Per mettersi in attesa
  - ⇨dopo avere ottenuto il lock, un thread chiama sull'oggetto condizione il metodo await()
  - ⇨rinuncia al lock e viene sospeso e inserito nella coda

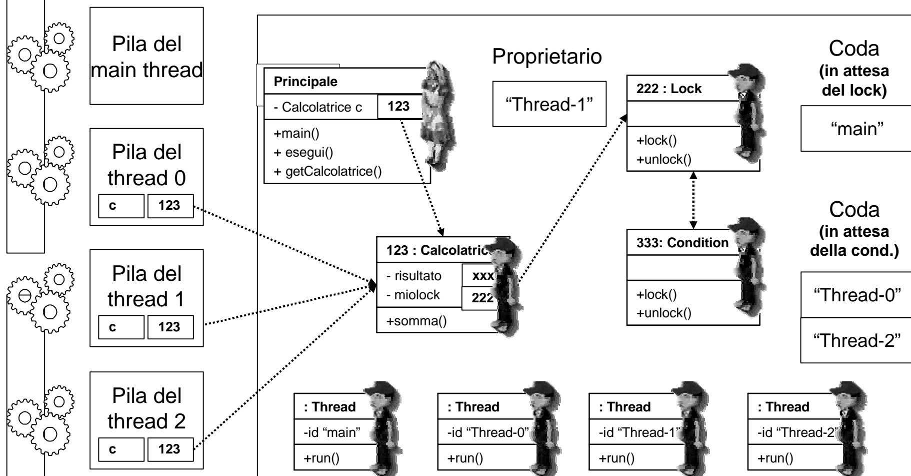
Condizioni

- Per segnalare un evento
  - ⇨un thread chiama sull'oggetto condizione il metodo signalAll()
  - ⇨questo risveglia tutti i thread nella coda, che diventano eseguibili
  - ⇨devono attendere il loro turno per cercare di riottenere il lock e riverificare la condizione

Condizioni

- Nota
  - ⇨tutto questo ha senso solo in una regione sincronizzata
  - ⇨cioè dopo avere eseguito il metodo lock()
  - ⇨il thread in attesa della condizione viene trattato in modo particolare rispetto all'acquisizione del lock per poter riprendere l'esecuzione dal punto in cui attendeva

Condizioni



Condizioni

>> bancathread.condizione

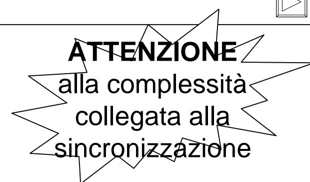
- Nel nostro esempio
  - prima di prelevare, ogni thread dovrebbe verificare che ci sia disponibilità sufficiente
  - se non è così, deve attendere che un altro thread versi dei soldi nel conto
  - a questo punto il thread originale può risvegliarsi ed effettuare il prelevamento
  - per farlo associo alla Banca un oggetto condition

Condizioni

- Attenzione alla differenza
  - tra await() e sleep()
- await()
  - viene eseguito su un oggetto condition
  - in una regione sincronizzata
  - sospende il thread e rilascia il lock
- sleep()
  - viene eseguito sul thread
  - non rilascia nessuno dei lock in possesso del thread

Condizioni

- Riassumendo
  - ⇨ vari nuovi strumenti di comunicazione
  - ⇨ i blocchi e le condizioni
- Monitor
  - ⇨ terminologicamente, un componente con un blocco valido per tutti i metodi ed almeno una condizione si chiama monitor



Condizioni

- Nota molto bene
  - ⇨ i due strumenti non sono per niente semplici da utilizzare
  - ⇨ usati assieme, aumentano moltissimo la probabilità che si verifichino stalli
- Esempio
  - ⇨ se la scelta del conto corrente da cui prelevare viene fatta a caso, la probabilità che con la condizione ci sia stallo è 1

Condizioni

>> bancathread.condizione

- Infatti
  - ⇨dopo un po' tutti i thread restano in attesa su conti con saldi negativi
  - ⇨la probabilità dell'evento che dovrebbe risvegliarli (versamento sul conto) si riduce man mano che i thread si riducono
  - ⇨questo porta ad una situazione si stallo
- Soluzione usata
  - ⇨ogni filiale preleva sempre dallo stesso conto

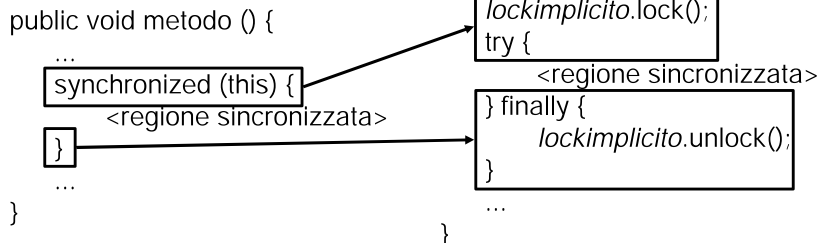
Sintassi Java Tradizionale

- Prima di J2SE 5.0
  - ⇨la gestione della sincronizzazione era diversa
- Idea
  - ⇨ogni oggetto Java dalla versione 1.0 ha un oggetto lock implicito associato
  - ⇨ed un oggetto condition implicito associato
  - ⇨per lavorare con gli oggetti si usano parole chiave opportune e metodi opportuni

Sintassi Java Tradizionale

- Per dichiarare una regione sincronizzata
  - ⇨la parola chiave synchronized prima del blocco di istruzioni
- Semantica

public void metodo () {



Sintassi Java Tradizionale

- Nota
  - ⇨è possibile sincronizzare anche interi metodi
  - ⇨con il modificatore synchronized

```
public synchronized void metodo ()
{
    <regione sincronizzata>
}
```

```
public void metodo () {
    lockimplicito.lock();
    try {
        <regione sincronizzata>
    } finally {
        lockimplicito.unlock();
    }
}
```

Sintassi Java Tradizionale

- Per mettersi in attesa di una condizione
  - i thread possono chiamare il metodo wait() direttamente utilizzando un riferimento all'oggetto in cui è definito il metodo
  - ovvero l'oggetto a cui appartiene il lock implicito
  - il metodo wait() rimanda al corrispondente metodo della condizione implicita

Sintassi Java Tradizionale

- Per risvegliare i metodi in attesa
  - i thread possono chiamare il metodo notifyAll(), sempre sull'oggetto a cui appartiene il lock
  - il metodo rimanda al metodo di segnalazione dell'oggetto condizione implicita
- Al solito
  - wait() e notifyAll() possono essere chiamati solo all'interno di una regione sincronizzata

Sintassi Java Tradizionale

>> bancathread.sincronizzato

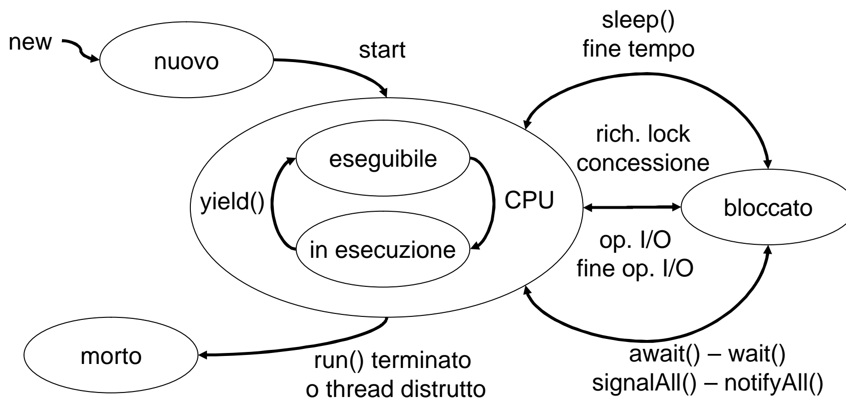
- I metodi wait() e notifyAll()
  - ⇨sono ereditati da Object, per cui sono richiamabili su qualsiasi riferimento
- Nel caso della banca
  - ⇨bancathread. tradizionale contiene il codice già visto che utilizza la sintassi Java tradizionale
- Differenza con il nuovo meccanismo
  - ⇨l'utilizzo di oggetti espliciti è più flessibile

Stati di un Thread

- Riassumendo
  - ⇨un thread può avere diversi stati
  - ⇨"nuovo": appena viene creato, prima che venga avviato
  - ⇨"eseguibile": quando è pronto ad essere eseguito dalla CPU
  - ⇨"in esecuzione": quando possiede la CPU
  - ⇨"bloccato": quando è sospeso
  - ⇨"morto": quando è terminato

Stati di un Thread

- Il diagramma degli stati di un thread



Tecniche di Programmazione: Thread >> Stati di un Thread

Stati di un Thread

- L’esecuzione con i thread
  - ⇨un processo molto articolato
  - ⇨in cui si alternano operazioni che creano nuovi thread e li avviano (“fork”)
  - ⇨ad operazioni di comunicazione e sincronizzazione tra i thread
  - ⇨ad operazioni in cui i thread si uniscono (“join”)

Riassumendo

- Sincronizzazione
- Blocchi
- Condizioni
- Sintassi Java Tradizionale
- Stati di un Thread

Termini della Licenza

- This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/1.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

- Questo lavoro viene concesso in uso secondo i termini della licenza “Attribution-ShareAlike” di Creative Commons. Per ottenere una copia della licenza, è possibile visitare http://creativecommons.org/licenses/by-sa/1.0/ oppure inviare una lettera all’indirizzo Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.