1 Programmazione concorrente in Java

Il linguaggio Java è un interessante linguaggio "general purpose" introdotto dalla Sun Microsystems e subito "esploso" come linguaggio per piccole applicazioni Internet. In seguito ha conquistato un grande credito anche nelle applicazioni generali, principalmente per le sue qualità di completo linguaggio ad oggetti e di indipendenza dalla piattaforma (ilsuo motto è: "compile once and run everywhere").
Il linguaggio è multipiattaforma perchè si basa su un approccio a "macchina virtuale" (v.m. "virtual machine"); il compilatore realizza un codice eseguibile (byte code) per una CPU "virtuale", che non esiste.
La macchina virtuale è un interprete, diverso per ogni piattaforma sulla quale il programma viene fatto eseguire, che ha il compito di tradurre le istruzioni da byte code a linguaggio macchina della CPU "ospite" e di far eseguire il programma.
Java ha istruzioni specifiche per la creazione di thread indipendenti e la loro sincronizzazione.
In questo capitolo si tratta solo delle caratteristiche di Java che riguardano il "multithreading", dando per scontato "il resto".

Thread non interagenti

Anche quando non è indispensabile che due programmi comunichino fra loro, può essere conveniente farli eseguire contemporaneamente, per maggiore efficienza o semplicità di realizzazione.
In Java un programma può lanciate nuovi thread che, pur condividendo lo stesso spazio di memoria del programma che lo crea, hanno "vita propria", indipendente, dal punto di vista dell'esecuzione.
Se i thread creati non interagiscono, non ci sono particolari problemi di programmazione, come già anticipato nel capitolo generale sulla programmazione concorrente.
Anche se si potrebbe non apprezzarlo "dall'esterno", tutti i programmi Java sono comunque multithreaded; molto spesso, durante la loro esecuzione, le librerie o la macchina virtuale creano nuovi thread, che eseguono in parallelo al programma principale.
La possibilità di creare nuovi thread è data anche al programmatore"normale", con le le istruzioni che andiamo ad introdurre. Da sottolineare che i metodi dei thread Java sono indipendenti dal S.O: la v.m. si incarica di interfacciarsi con il Sistema Operativo, chiamando i semafori che il S.O. mette a disposizione, oppure, se il S.O. non ha funzioni di sincronizzazione fra i processi, di realizzare "da sola" tutta la sincronizzazione.

Un thread Java è realizzato con una classe che eredita dala classe "Thread", oppure che implementa l'interfaccia "runnable" (quest'ultimo caso, più flessibile ma più complesso, non verrà trattato in questo testo).

La dichiarazione di una nuova classe che eredita da "Thread" ci permetterà in seguito di creare da quella classe uno o più oggetti che eseguono in parallelo.

Sintassi:
```
class <nome Classe> extends Thread
```

un'istanza di questa classe viene creata come al solito, con "new":
```
<nome Istanza> = new <nome Classe> ();
```

Per far partire l'istanza come thread indipendente, basta lanciare il suo metodo start:
```
<nome Istanza>.start;  // fa partire <nome Istanza> come thread
                       // (la v.m. lancia il suo metodo run())
```

Il metodo .start(), evocato nel programma che lancia il thread, fa partire il metodo .run() della classe. Il metodo run() ha questo prototipo:
```
public void run() {
    // metodo del thread che costituisce il programma da eseguire
}
```

il metodo run() viene fatto partire, in ogni istanza della classe "threaded", quando viene usato il metodo .start() sull'oggetto (istanza).

1.0.1 Metodi principali per i thread Java non interagenti

<oggetto>.run() metodo di partenza del thread
<oggetto>.yield() fa sospendere il thread corrente per lasciare l'esecuzione agli altri

<oggetto>.sleep(long) interrompe l'esecuzione del thread corrente per il numero indicato di millisecondi. (il thread non perde il controllo dei monitor durante le sleep())

<oggetto>.suspend() sospende l'esecuzione di questo thread, che riprende solo con una resume()
<oggetto>.resume() riprende un thread sospeso
<oggetto>.stop() fa abortire il thread indicato (deprecato per la difficoltà di gestione)

<oggetto>.getName() acquisisce il nome di questo thread
<oggetto>.setName(String) cambia il nome di questo thread
<oggetto>.currentThread() riferimento al thread che esegue correntemente
<oggetto>.toString() nome del thread, priorità e gruppo di thread

<oggetto>.getPriority() priorità di questo thread
<oggetto>.setPriority(int) cambia la priorità di questo thread

<oggetto>.interrupt() interrompe questo thread
<oggetto>.interrupted() controlla se questo thread è interrotto (chiamata sulla classe (metodo static))
<oggetto>.isInterrupted() controlla se questo thread è stato interrotto (chiamata sull'istanza corrente del thread)
<oggetto>.isAlive() controlla se questo thread è vivo

Sincronizzazione
Costrutti Java in "stile monitor"

Metodi "synchronized"

Dichiarazione Synchronized

```
synchronized <dichiarazione di un metodo> {
    <codice del metodo>
}
```

Il metodo definito come "synchronized" è una delle procedure di un monitor. Per questioni di efficienza, in Java non è obbligatorio che tutte le procedure di una classe che ha un monitor siano "synchronized", per cui non stiamo trattando con un monitor "puro".

Sincronizzazione esplicita

Esistono dei metodi per indicare esplicitamente che il thread deve sospendersi in una coda o deve far riprendere altri processi sospesi in una coda.

```
.wait()
sospensione nella coda del monitor

.notify(), .notifyAll()
ripresa dell'esecuzione da una coda
```

Blocchi "synchronized"

Sintassi della dichiarazione di un blocco Synchronized

```
synchronized (<espressione>) <Blocco>

Blocco = {<qualsiasi insieme di istruzioni in Java>}
```

Java però non verifica che tutte le procedure che utilizzano i dati "protetti" dalle procedure "synchronized" siano effettivamente modificati in procedure "synchronized" .

1.0.2 Deadlock