1 Lezione del 16-10-25

Ritorniamo sull'argomento dei thread.

Aggiungiamo, rispetto alla scorsa lezione, che ogni thread ha il suo stack (con il suo record di attivazione). Inoltre, tutti i thread condividono lo stesso heap. Il GC non rimuove gli elementi che sono raggiungibili da qualunque thread.

1.0.1 Interfaccia Runnable

Visto che in Java l'ereditarietà è unitaria, dover sempre estendere la classe *Thread* può essere limitante. In questo Java fornisce l'interfaccia *Runnable*, che quando implementata rende una classe *eseguibile* (come una lambda o un funtore).

Come per la classe Thread, l'interfaccia Runnable è provvista di un metodo astratto `run()` da ridefinire per specificarne il comportamento.

Quando si crea un'oggetto *Thread* si può quindi usare un'oggetto che implementa l'interfaccia Runnable come argomento al costruttore:

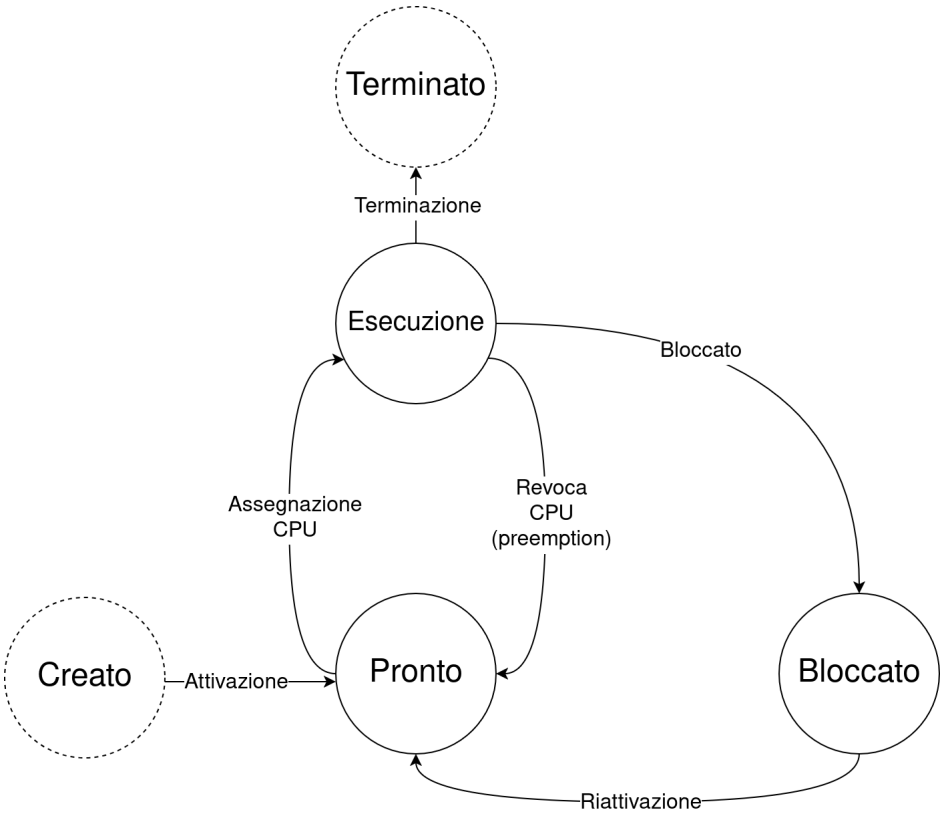
```
class MyThread implements Runnable {
  @Override
  void run() {
    System.out.println("Sono un thread");
  }
}

// da qualche altra parte
Thread myThread = new Thread(new MyThread());
myThread.start();

// stampera' "Sono un thread" da un nuovo thread
```

1.1 Stati di un thread

Un thread può trovarsi in diversi stati, schematizzabili attraverso lo schema (preso direttamente dagli appunti di sistemi operativi):



In Java le transizioni fra stati possono corrispondere a particolari chiamate di metodi da dentro l'oggetto thread. In particolare:

- *Pronto*: il thread arriva qui quando si chiama il metodo `start()`, e può tornarci alla fine del quanto di tempo assegnatogli, per *preemption*, o per concessione volontaria (attraverso `Thread.yield()`).

  Inoltre, torna qui se era bloccato e viene sbloccato (fine `sleep()`, `notify()`, blocchi su risorse rilasciati, ecc...);

- *Bloccato*: un thread arriva qui quando effettua una sospensione volontaria (`sleep()`, `wait()`, blocchi `synchronized`) o quando viene bloccato involontariamente su risorse (file, I/O, ecc...);

- *Esecuzione*: questo è lo stato in cui il thread effettivamente esegue.

1.1.1 Thread e S/O

Il concetto di thread in Java è chiaramente parallelo a quello di thread nei S/O.

Vediamo una breve outline cronologica di questa relazione:

- Oggi la prassi è quella di associare ad ogni thread Java un thread di S/O, e lasciare che siano i meccanismi di gestione dei processi del S/O a gestire i thread Java in esecuzione;

- Storicamente, questo non è stato sempre il caso: le prime JVM erano infatti processi monolitici, con un unico flusso di esecuzione, che gestivano internamente tutti i diversi thread.

In ogni caso, le specifiche del linguaggio si dimostrano (volutamente) blande sull'argomento thread, in quanto la maggior parte delle problematiche associate alla gestione di questi è associata alla struttura specifica del S/O.

1.1.2 Priorità thread

Di un thread Java si può impostare la priorità (un numero da 1 a 10, via `setPriority (int prio)`), e questa dovrebbe grossomodo corrispondere alla priorità del corrispondente thread S/O (grossomodo in quanto l'S/O potrebbe prevedere più di 10 livelli di priorità).

Per questo motivo, bisognerebbe usare la priorità dei thread Java solo come indicazione per il S/O e come ottimizzazione, e non fidarsi completamente di ottenere del comportamento sperabile.

Il thread di default parte con priorità normale (il valore medio, cioè 5). Quando si creano nuovi thread, questi ereditano la priorità del thread padre (quindi 5 dal thread di default).

La schedulazione dei thread Java è in genere di tipo *fixed priority*.

1.1.3 Terminare e riattivare thread

Quando un thread termina, non si può usare nuovamente `start()` per rimetterlo in esecuzione, ma bisogna crearne una nuova istanza ed eseguire quella.

Se da un thread si vuole terminare l'esecuzione del solo thread, si può chiaramente ritornare da `run()`. Se si vuole invece terminare l'esecuzione dell'intero programma, si può usare `System.exit()`.

1.1.4 Metodo sleep

Il metodo `sleep(long t)` permette di sospendere il thread per il numero di millisecondi specificato in `t`.

Il metodo può lanciare l'eccezione *InterruptedException*, e questo accade quando il thread viene interrotto. Per questo bisogna definire un blocco `catch` che catturi tale eccezione e fornisca un qualche tipo di handler:

```
1  void run() {
2    while(true) {
3      System.out.println("Tick");
4      try {
5        Thread.sleep(1000);
6      } catch(InterruptedException e) {
7        System.err.println("Il thread e' stato svegliato. " + e.getMessage()
       );
8      }
9    }
10 }
```

1.2 Corse critiche

Un problema in cui si può incorrere quando si sfruttano i thread è quello delle **corse critiche**.

Quando più thread interagiscono con lo stesso oggetto, infatti, potrebbero lasciarlo in stati inconsistenti a causa di scritture contemporanee che si cancellano fra di loro. In altre parole, le operazioni su campi di oggetti in Java non sono (di default) *atomiche*,

e i metodi che le implementano possono essere deschedulati (per preemption) quando l'oggetto su cui stanno lavorando è ancora in stato inconsistente.

1.2.1 Monitor, metodi `synchronized`

In Java questo problema viene risolto sfruttando il modello dei **monitor**. Secondo questo modello, ad ogni oggetto è associato un **lock**, che può essere *aperto* o *chiuso*.

Per agire su un oggetto, un thread deve acquisire un lock. Quando avrà finito la sua operazione, potrà rilasciare tale lock.

Dal punto di vista sintattico, questo si implementa usando il modificatore (applicabile ai metodi) `synchronized`. Quando un thread esegue un metodo `synchronized` e agisce su un'oggetto x, quello che accade è:

- Acquisisce il lock di x;
- Esegue il corpo del metodo ;
- Rilascia il lock di x.

Tutti i metodi dichiarati come `synchronized` condividono per ciascun oggetto un lock (il lock predefinito). I metodi non `synchronized` possono invece bypassare il meccanismo dei lock, anche quando un metodo che invece è `synchronized` ha il lock sull'oggetto cercato.

Per questo è importante dichiarare come `synchronized` tutti i metodi che rischiano di portare oggetti in stati inconsistenti. Di contro, è abbastanza naturale definire metodi che non richiedono di rispettare i lock (metodi non bloccanti, di lettura, ecc...).

Abbiamo quindi che i metodi `synchronied` *cercano* di ottenere i lock sugli oggetti che modificano. I lock si acquisiscono per conto del thread corrente: se quel thread è già in possesso del lock cercato, non si blocca e va direttamente all'esecuzione del metodo.

Nel caso di metodi `synchronized` il lock viene rilasciato:

- Al normale termine del esecuzione del metodo;
- In caso di situazioni particolari come le eccezioni.

1.2.2 Blocchi `synchronized`

La parola chiave `synchronized` può essere usata anche per definire *blocchi* di codice, e non solo metodi. In questo caso tali blocchi si comportano esattamente come i metodi che abbiamo descritto finora.

I blocchi sincronizzati accettano un argomento, che è quello su cui vogliamo bloccare. Nel caso di metodi sincronizzati, l'oggetto è implicitamente la classe del metodo:

```
class A {
  void synchronized foo() {
    // sincronizza su A
  }

  void bar() {
    synchronized(this) {
      // come sopra ma inutilmente esplicito
    }
  }
}
```

Coi blocchi sincronizzati si possono però avere altri oggetti su cui sincronizzare. Il loro uso è quindi necessario quando:

- Non tutto il codice di un metodo deve essere eseguito in mutua esclusione (sincronizzato). Questo è particolarmente utile aumentare il grado di parallelismo dei thread: se si sincronizza solo quando gli oggetti effettivamente ci servono, si lascia tempo agli altri thread per lavorare sullo stesso oggetto;
- Si vuole sincronizzare su oggetti diversi da quello implicito. Ad esempio, si potrebbe voler dire:

```
void metodo(Parametro p) {
  synchronized(p) {
    // qui abbiamo il lock su p
  }
}
```

  In questo caso l'oggetto di cui si prende il lock non è più la classe corrente, ma l'oggetto di tipo *Parametro* che passiamo nell'argomento *p*.

  Questo può essere utile quando si usano oggetti che non prevedono la sincronizzazione di default: in questo modo imponiamo *noi* che le operazioni che svolgiamo vengano sincronizzate;

- Si vuole sincronizzare su un array. Questo in quanto le array non possono essere soggetto di lock impliciti (non sono classi che definiscono metodi). La sintassi è in questo caso analoga all'esempio precedente.

Più lock su più oggetti possono essere ottenuti innestando blocchi `synchronized`.

1.2.3 Lock statici

Quando il modificatore `synchronized` viene usato su metodi statici il lock implicito non è sull'*istanza* di classe, ma sull'intera classe. In questo modo si possono definire metodi che esegono in maniera mutualmente esclusiva sull'intero oggetto classe.

1.3 Deadlock

Nel caso due thread debbano ottenere un lock appartenente all'altro per portare avanti la loro operazione, e non siano ancora in condizioni di rilasciare il loro lock, si incorre in una situazione detta **deadlock**.

Un risultato teorico è che *se si prendono i lock sempre nello stesso ordine*, i deadlock non possono verificarsi. Questo chiaramente non è sempre facile dal punto di vista implementativo.

1.4 Sincronizazzione

Veniamo a come eseguire operazioni in thread solo nella condizione che un altro thread abbia eseguito la sua operazione. Questo è un problema di *sincronizzazione* o in particolare di **comunicazione** fra thread.

Facciamo attenzione al fatto che la parola chiave `synchronized` è fuorviante: in verità questa risolve un problema di *interferenza*, o più propriamente di **mutua esclusione** dell'accesso a determinate risorse.

Il problema che vogliamo invece risolvere adesso è di *sincronizzazione* temporale nell'esecuzione di operazioni in stretto ordine cronologico.

In Java questo problema viene risolto sfruttando metodi definiti direttamente sulla classe *Object*:

- `notify()`: permette di *notificare*, appunto, un oggetto nel *wait-set* dell'oggetto che lo chiama (più dettagli in seguito);
- `notifyAll()`: come sopra, ma notifica tutti gli oggetti nel *wait-set*;
- `wait()`: permette di sospendere l'esecuzione di un thread fino al verificarsi di una determinata condizione (notificata da qualcun'altro). Questo è un metodo che non restituisce nulla ma può lanciare la *InterruptedException*. Quando si chiama `wait()` i lock dell'oggetto vengono rilasciati e acquisiti automaticamente al rientro in esecuzione.

  Ad ogni oggetto è associato un *wait-set*. Quando si chiama wait su un oggetto il thread viene sospeso e inserito nel *wait-set* di quell'oggetto. Quando il *wait-set* riceve una notifica con `notify()`, uno degli oggetti in attesa viene risvegliato e messo in coda pronti. Se invece la notifica è con `notifyAll()`, vengono risvegliati tutti.