Tutorial Breve:

Java e la gestione dei Thread

di Claudio De Sio Cesari

I thread, rappresentano il mezzo mediante il quale, Java fa eseguire un'applicazione da più Virtual Machine contemporaneamente, allo scopo di ottimizzare i tempi del runtime. Ovviamente, si tratta di un'illusione: per ogni programma solitamente esiste un'unica JVM ed un'unica CPU. Ma la CPU può eseguire codice da più progetti all'interno della gestione della JVM per dare l'impressione di avere più processori.

L'esperienza ci ha insegnato che l'apprendimento di un concetto complesso come la gestione dei thread in Java, richiede un approccio graduale sia per quanto riguarda le definizioni, sia per quanto riguarda le tecniche di utilizzo. Tale convincimento, oltre ad avere una natura empirica, nasce dall'esigenza di dover definire un thread quale oggetto della classe `Thread`*, cosa che può portare il discente facilmente a confondersi. Didatticamente inoltre, è spesso utile utilizzare schemi grafici per aiutare il discente nella comprensione. Ciò risulta meno fattibile quando si cerca di spiegare il comportamento di più thread al runtime, poichè un schema statico non è sufficiente. Per aiutare il lettore nella comprensione degli esempi presentati in questo tutorial, viene quindi fornita una semplice applet che permette di navigare tra le schematizzazioni grafiche dei momenti topici di ogni esempio.*

Definizione provvisoria di Thread

Quando lanciamo un'applicazione Java vengono eseguite le istruzioni contenute in essa in maniera sequenziale, a partire dal codice del metodo main. Spesso, soprattutto in fase di debug, allo scopo di simulare l'esecuzione dell'applicazione, il programmatore

immagina un cursore che scorre sequenzialmente le istruzioni, magari, simulando il suo movimento con un dito che punta sul monitor. Un tool di sviluppo che dispone di un debugger grafico invece, evidenzia concretamente questo cursore. Consapevoli che il concetto risulterà familiare ad un qualsiasi programmatore, possiamo per il momento identificare un thread proprio con questo cursore immaginario. In tal modo, affronteremo le difficoltà dell'apprendimento in maniera graduale.

Cosa significa "multi-threading"

L'idea di base è semplice: immaginiamo di lanciare un'applicazione Java. Il nostro cursore immaginario, scorrerà ed eseguirà sequenzialmente le istruzioni partendo dal codice del metodo `main`. *Quindi al runtime, esiste almeno un thread in esecuzione, ed il suo compito è quello di eseguire il codice, seguendo il flusso definito dall'applicazione stessa.*

Per "multi-threading" si intende il processo che porterà un' applicazione, a definire più di un thread, assegnando ad ognuno compiti da eseguire parallelamente. Il vantaggio che può portare un'applicazione multi-threaded, è relativo soprattutto alle prestazioni della stessa. Infatti i thread possono "dialogare" allo scopo di spartirsi nella maniera ottimale l'utilizzo delle risorse del sistema. D'altronde, l'esecuzione parallela di più thread all'interno della stessa applicazione è vincolata all'architettura della macchina su cui gira. In altre parole, se la macchina ha un unico processore, in un determinato momento `x`*, può essere in esecuzione un unico thread. Ciò significa che un'applicazione non multi-threaded che richiede in un determinato momento alcuni input (da un utente, una rete...) per poter proseguire nell'esecuzione, quando si trova in stato di attesa non può eseguire nulla. Un'applicazione multi-threaded invece, potrebbe eseguire altro codice mediante un altro thread, "avvertito" dal thread che è in stato di attesa.*

Solitamente, l'essere multi-threaded, è una caratteristica dei sistemi operativi (per esempio Unix), piuttosto che dei linguaggi di programmazione. La tecnologia Java, tramite la Virtual Machine, ci offre uno strato d'astrazione per poter gestire il multi-threading direttamente dal linguaggio. Altri linguaggi (come il C/C++), solitamente sfruttano le complicate librerie del sistema operativo per gestire il multi-threading, lì dove possibile. Infatti tali linguaggi, non essendo stati progettati per lo scopo, non supportano un meccanismo chiaro per gestire i thread [1].

In Java, i meccanismi della gestione dei thread, risiedono essenzialmente:

1. *Nella classe* `Thread` *e l' interfaccia* `Runnable` *(package* `java.lang`*)*

2. *Nella classe* `Object` *(ovviamente package* `java.lang`*)*

3. *Nella JVM e nella keyword* `synchronized`

La classe `Thread` *e la dimensione temporale*

Come abbiamo precedentemente affermato, quando si avvia un'applicazione Java, c' è almeno un thread in esecuzione, appositamente creato dalla JVM per eseguire il codice dell'applicazione [2]. *Nel seguente esempio introdurremo la classe* `Thread` *e vedremo che anche con un unico thread è possibile creare situazioni interessanti.*

```
1   public class ThreadExists {
2     public static void main(String args[]) {
3       Thread t = Thread.currentThread();
4       t.setName("Thread principale");
5       t.setPriority(10);
6       System.out.println("Thread in esecuzione: " + t);
7       try {
8         for (int n = 5; n > 0; n--) {
9           System.out.println("" + n);
10          t.sleep(1000);
11        }
12      }
13      catch (InterruptedException e) {
14        System.out.println("Thread interrotto");
15      }
16    }
17  }
```

Output:

```
C:\TutorialJavaThread\Code>java ThreadExists

Thread in esecuzione: Thread[Thread principale,10,main]

5

4

3

2

1
```

Analisi di `ThreadExists`

Questa semplice classe produce un risultato tutt'altro che trascurabile: la gestione del tempo. La durata dell'esecuzione del programma è infatti quantificabile in circa 5 secondi. Analizziamo il codice nei dettagli:

Alla riga 3, viene chiamato il metodo statico della classe `Thread currentThread`*. Questo restituisce l'indirizzo dell'oggetto* `Thread` *che sta leggendo l'istruzione, che viene assegnato al reference* `t`*. Questo passaggio è particolarmente delicato: abbiamo identificato un thread con il "cursore immaginario" che processa il codice, ed in questo passaggio abbiamo ottenuto un reference a questo cursore. Una volta ottenuto un reference, è possibile gestire il thread, gestendo così l'esecuzione (temporale) dell'applicazione! Notiamo la profonda differenza tra la classe* `Thread`*, a tutte le altre classi della libreria standard. La classe* `Thread` *astrae un concetto che non solo è dinamico, ma addirittura rappresenta l'esecuzione stessa dell'applicazione! Il currentThread infatti non è "l'oggetto corrente" , che è solitamente individuato dalla parola chiave* `this`*, ma l'oggetto (thread corrente) che esegue l'oggetto corrente. Potremmo affermare che un oggetto* `Thread` *si trova in un'altra dimensione rispetto agli altri oggetti: la dimensione temporale.*

Continuiamo con l'analisi della classe `ThreadExists`*. Alle righe 4 e 5, scopriamo che è possibile non solo assegnare un nome al*

thread, ma anche una priorità. La scala delle priorità dei thread in Java, va dalla priorità minima 1 alla massima 10, e la priorità di default è 5 [3]. *Alla riga 6, viene stampato l'oggetto* `t` *(ovvero* `t.toString()`*). Dall'output notiamo che vengono stampate informazioni sul nome e la priorità del thread, oltre che sulla sua appartenenza al gruppo dei thread denominato main. I thread infatti appartengono a dei* `ThreadGroup`*, ma non ci occuperemo di questo argomento in dettaglio, (consultare la documentazione). Tra la riga 7 e la riga 12 viene dichiarato un blocco* `try` *contenente un ciclo* `for` *che esegue un conto alla rovescia da 5 ad 1. Tra una stampa di un numero ed un altra c'è una chiamata al metodo* `sleep()` *sull'oggetto* `t`*, a cui viene passato l'intero* `1000`*. In questo modo il thread che esegue il codice, farà un pausa di un secondo (1000 millisecondi) tra la stampa di un numero ed un altro. Tra la riga 13 e la riga 15 viene definito il blocco* `catch` *che gestisce una* `InterruptedException`*, che il metodo* `sleep` *dichiara nella sua clausola* `throws`*. Questa eccezione scatterebbe nel caso in cui il thread non riesca ad eseguire "il suo codice" perchè stoppato da un altro thread. Nel nostro esercizio però, vi era che un unico thread, e quindi la gestione dell'eccezione non aveva senso. Nel prossimo paragrafo vedremo come creare altri thread, sfruttando il thread principale.*

L'interfaccia `Runnable` *e la creazione dei thread*

Per avere più thread basta istanziarne altri dalla classe `Thread`*. Nel prossimo esempio noteremo che quando si istanzia un oggetto* `Thread`*, bisogna passare al costruttore un'istanza di una classe che implementa l'interfaccia* `Runnable`*. I questo modo infatti, il nuovo thread, quando sarà fatto partire (mediante la chiamata al metodo* `start()`*), andrà ad eseguire il codice del metodo* `run` *dell'istanza associata. L' interfaccia* `Runnable` *quindi, richiede l' implementazione del solo metodo* `run` *che definisce il comportamento di un thread, e l' avvio di un thread si ottiene con la chiamata del metodo* `start()`*. Dopo aver analizzato il prossimo esempio, le idee dovrebbero risultare più chiare.*

```
1   public class ThreadCreation implements Runnable {
2     public ThreadCreation () {
3       Thread ct = Thread.currentThread();
4       ct.setName("Thread principale");
5       Thread t = new Thread(this, "Thread figlio");
6       System.out.println("Thread attuale: " + ct);
7       System.out.println("Thread creato: " + t);
8       t.start();
```

```
9       try {
10        Thread.sleep(3000);
11      }
12      catch (InterruptedException e) {
13        System.out.println("principale interrotto");
14      }
15      System.out.println("uscita Thread principale");
16    }
17    public void run() {
18      try {
19        for (int i = 5; i > 0; i--) {
20          System.out.println("" + i);
21          Thread.sleep(1000);
22        }
23      }
24      catch (InterruptedException e) {
25        System.out.println("Thread figlio interrotto");
26      }
27      System.out.println("uscita Thread figlio");
28    }
29    public static void main(String args[]) {
30      new ThreadCreation();
31    }
32  }
```

Output:

```
C:\TutorialJavaThread\Code>java ThreadCreation

Thread attuale: Thread[Thread principale,5,main]

Thread creato: Thread[Thread figlio,5,main]
```

```
5

4

3

uscita Thread principale

2

1

uscita Thread figlio
```

Analisi di `ThreadCreation`

Nel precedente esempio oltre al thread principale, ne è stato istanziato un secondo. La durata dell'esecuzione del programma è anche in questo caso quantificabile in circa 5 secondi. Analizziamo il codice nei dettagli:

A supporto della descrizione dettagliata del runtime dell'esempio, viene presentata un applet che schematizza graficamente la situazione dei due thread, nei momenti topici del runtime. Clicka qui

L'applicazione al runtime viene eseguita a partire dal metodo `main` *alla riga 29. Alla riga 30 viene istanziato un oggetto della classe* `ThreadCreation` *poi il nostro cursore immaginario si sposta ad eseguire il costruttore dell'oggetto appena creato, alla riga 3. Qui il nostro cursore immaginario (il thread corrente), ottiene un reference* `ct`*. Notiamo che "*`ct` *esegue* `this`*" . A* `ct` *viene poi assegnato il nome "thread principale" . Alla riga 5 viene finalmente istanziato un altro thread, dal thread corrente* `ct` *che sta eseguendo la riga 5 [fig. 1 dell'applet]. Viene utilizzato un costruttore che prende in input due parametri. Il primo (*`this`*) è un oggetto* `Runnable` *(ovvero un'istanza di una classe che implementa l'interfaccia* `Runnable`*), il secondo è ovviamente il nome del thread. L'oggetto* `Runnable` *contiene il metodo* `run`*, che diventa l'obbiettivo dell'esecuzione del thread* `t`*. Quindi, mentre per il thread principale l'obbiettivo dell'esecuzione è scontato, per i thread che vengono istanziati bisogna specificarlo passando al*

costruttore un oggetto `Runnable`*. Alle righe 6 e 7 vengono stampati messaggi descrittivi dei due thread. Alla riga 8 viene finalmente fatto partire il thread* `t` *mediante il metodo* `start` [4]*. La partenza di un thread non implica che il thread inizi immediatamente ad eseguire il suo codice, ma solo che è stato reso eleggibile per l'esecuzione [fig. 2]. Quindi, il thread* `ct`*, dopo aver istanziato e reso eleggibile per l'esecuzione il thread* `t`*, continua nella sua esecuzione fino a quando giunto ad eseguire la riga 10, incontra il metodo* `sleep` *che lo ferma per 3 secondi* [5]*. A questo punto il processore è libero dal thread* `ct`*, e viene utilizzato dal thread* `t`*, che finalmente può eseguire il metodo* `run`*. Ecco che allora il thread* `t` *va ad eseguire un ciclo, che come nell'esempio precedente, realizza un conto alla rovescia, facendo pausa da un secondo. Esaminando l'output verifichiamo che il codice fa si che, il thread* `t` *stampi il 5 [fig. 3], faccia una pausa di un secondo [fig. 4], stampi 4, pausa di un secondo, stampi 3, pausa di un secondo. Poi si risveglia il thread* `ct` *che stampa la frase "uscita thread principale" [fig. 5], e poi "muore" . Quasi contemporaneamente viene stampato 2 [fig. 6], pausa di un secondo, stampa di 1, pausa di un secondo, stampa di "uscita thread figlio" .*

Nell'esempio quindi, l'applicazione ha un ciclo di vita superiore a quello del thread principale, grazie al thread creato.

La classe `Thread` *e la creazione dei thread*

Abbiamo appena visto come un thread creato deve eseguire codice di un oggetto istanziato da una classe che implementa l'interfaccia `Runnable` *(l'oggetto* `Runnable`*). Ma la classe* `Thread` *stessa implementa l'interfaccia* `Runnable`*, fornendo un implementazione vuota del metodo* `run`*. È quindi possibile fare eseguire ad un thread il metodo* `run` *definito all'interno dello stesso oggetto thread. Per esempio:*

```
1   public class CounterThread extends Thread {
2     public void run() {
3       for (int i = 0; i<10; ++i)
4         System.out.println(i);
5     }
6   }
```

è possibile istanziare un thread senza specificare l'oggetto `Runnable` *al costruttore e farlo partire con il solito metodo* `start`*:*

```
1   CounterThread thread = new CounterThread ();

2   thread.start();
```

[6]

Sicuramente la strategia di fare eseguire il metodo `run` *all'interno dell'oggetto thread stesso, è più semplice rispetto a quella vista nel paragrafo precedente. Tuttavia ci sono almeno tre buone ragioni per preferire il passaggio di un oggetto* `Runnable`*:*

1. *In Java una classe non può estendere più di una classe alla volta. Quindi, implementando l'interfaccia* `Runnable`*, piuttosto che estendere* `Thread`*, permetterà di utilizzare l'estensione per un altra classe.*

2. *Solitamente un oggetto della classe* `Thread` *non dovrebbe possedere variabili d'istanza private che rappresentano i dati da gestire. Quindi il metodo* `run` *nella sottoclasse di* `Thread`*, non potrà accedere, o non potrà accedere in una maniera "pulita" , a tali dati.*

3. *Dal punto di vista della programmazione object oriented, una sottoclasse di* `Thread` *che definisce il metodo* `run`*, combina due funzionalità poco relazionate tra loro: il supporto del multi–threading ereditato dalla classe* `Thread`*, e l'ambiente esecutivo fornito dal metodo* `run`*. Quindi in questo caso l'oggetto creato è un thread, ed è associato con se stesso, e questa non è una soluzione molto object oriented*

Priorità, scheduler e sistemi operativi

Abbiamo visto come il metodo `start` *chiamato su di un thread non implichi che questo inizi immediatamente ad eseguire il suo codice (contenuto nel metodo* `run` *dell'oggetto associato). In realtà la JVM, definisce un thread scheduler, che si occuperà di decidere in ogni momento quale thread deve trovarsi in esecuzione. Il problema che la JVM stessa è un software che gira su di un determinato sistema operativo, e la sua implementazione, dipende dal sistema. Quando si gestisce il multi-threading ciò può apparire evidente come nel prossimo esempio. Infatti, lo scheduler della JVM, deve comunque rispettare la filosofia dello scheduler del sistema operativo, e questa può cambiare clamorosamente tra sistema e sistema. Prendiamo in considerazione i due più importanti sistemi attualmente in circolazione: Unix e Windows (qualsiasi versione). Il seguente esempio produce output completamente diversi, su i due sistemi:*

```
1   public class Clicker implements Runnable {
2     private int click = 0;
3     private Thread t;
4     private boolean running = true;
5     public Clicker(int p) {
6       t = new Thread(this);
7       t.setPriority(p);
8     }
9     public int getClick(){
10      return click;
11    }
12    public void run() {
13      while (running) {
14        click++;
15      }
16    }
17    public void stopThread() {
18      running = false;
19    }
20    public void startThread() {
21      t.start();
22    }
23  }
24  public class ThreadRace {
25    public static void main(String args[]) {
26      Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
27      Clicker hi = new Clicker(Thread.NORM_PRIORITY + 2);
28      Clicker lo = new Clicker(Thread.NORM_PRIORITY - 2);
29      lo.startThread();
```

```
30      hi.startThread();
31      try {
32        Thread.sleep(10000);
33      }
34      catch (Exception e){}
35      lo.stopThread();
36      hi.stopThread();
37      System.out.println(lo.getClick()+" vs. "+hi.getClick());
38    }
39  }
```

Output Solaris 8:

```
solaris% java ThreadRace

0 vs. 1963283920
```

Vari Output Windows 2000:

```
C:\TutorialJavaThread\Code>java ThreadRace

15827423 vs. 894204424

C:\TutorialJavaThread\Code>java ThreadRace

32799521 vs. 887708192

C:\TutorialJavaThread\Code>java ThreadRace

15775911 vs. 890338874
```

```
C:\TutorialJavaThread\Code>java ThreadRace

15775275 vs. 891672686
```

Analisi di `ThreadRace`

Il precedente esempio è composto da due classi: `ThreadRace` *e* `Clicker`. `ThreadRace` *contiene il* `main` *e rappresenta quindi la classe principale. Immaginiamo di lanciare l'applicazione, clicka qui per il supporto dell'applet:*

Alla riga 27 , viene assegnata al thread corrente, la priorità massima tramite la costante statica intera della classe `Thread` `MAX_PRIORITY`*, che ovviamente vale* `10`*. Alle righe 28 e 29, vengono istanziati due oggetti dalla classe* `Clicker`*,* `hi` *e* `lo`*, ai cui costruttori vengono passati i valori interi* `7` *e* `3`*. Gli oggetti* `hi` *e* `lo`*, tramite costruttori creano due thread associati ai propri metodi* `run` *rispettivamente con priorità* `7` *e priorità* `3`*. Il thread principale, continua nella sua esecuzione chiamando su entrambi gli oggetti* `hi` *e* `lo`*, il metodo* `startThread`*, che ovviamente rende eleggibili per l'esecuzione i due thread a priorità 7 e 3 nei rispettivi oggetti [fig. 1 dell'applet]. Alla riga 33, il thread principale a priorità 10 va a "dormire" per una decina di secondi, lasciando disponibile la CPU agli altri due thread [fig. 2]. In questo momento dell'esecuzione dell'applicazione che lo scheduler avrà un comportamento dipendente dalla piattaforma:*

Comportamento Windows (Time-Slicing o Round-Robin scheduling):

Un thread può trovarsi in esecuzione solo per un certo periodo di tempo, poi deve lasciare ad altri thread la possibilità di essere eseguiti. Ecco che allora l'output di Windows evidenzia che entrambi i thread hanno avuto la possibilità di eseguire codice. Il thread a priorità 7, ha avuto a disposizione per molto più tempo il processore, rispetto al thread a priorità 3. [7]

Comportamento Unix (Preemptive scheduling): un thread in esecuzione, può uscire da questo stato solo nelle seguenti situazioni:

1. *Viene chiamata un metodo di scheduling come* `wait()` *o* `suspend()`
2. *Viene chiamato un metodo di blocking, come quelli dell' I/O*

3. *Può essere "buttato fuori" dalla CPU da un altro thread a priorità più alta che diviene eleggibile per l'esecuzione*

4. *Termina la sua esecuzione (il suo metodo* `run`*).*

Quindi il thread a priorità 7, ha occupato la CPU, per tutti i 10 secondi che il thread principale (a priorità 10), è in pausa. Quando quest'ultimo si risveglia rioccupa di forza la CPU.

Su entrambi i sistemi poi l'applicazione continua con il thread principale che, chiamando il metodo `stopThread` *(righe 35 e 36) su entrambi gli oggetti* `hi` *e* `lo`*, setta le variabili* `running` *a* `false`*. Il thread principale termina la sua esecuzione stampando il "risultato finale" [fig. 3]. A questo punto parte il thread a priorità 7, la cui esecuzione si era bloccata alla riga 13 o 14 o 15. Se riparte dalla riga 14, la variabile* `click` *viene incrementata un'altra ultima volta, senza però influenzare l'output dell'applicazione. Quando si troverà alla riga 13, la condizione del ciclo* `while` *non verrà verificata, e quindi il thread a priorità 7 terminerà la sua esecuzione[fig. 5]. Stessa sorte toccherà al thread a priorità 3 [fig. 5].*

Thread e Sincronizzazione

Abbiamo sino ad ora identificato un thread come il cursore immaginario. Per quanto è utile poter pensare ad un thread in questi termini, riteniamo il momento maturo per poter dare una definizione più "scientifica" di thread.

Un thread è un ***processore virtuale****, che esegue* ***codice*** *su determinati* ***dati****.*

Nell'esempio precedente `ThreadRace`*, i due thread creati (quelli a priorità 7 e 3), eseguono lo stesso codice (il metodo* `run` *della classe* `Clicker`*), utilizzando dati diversi (le variabili* `lo.click` *ed* `hi.click`*). Quando però due o più thread necessitano contemporaneamente dell' accesso ad una fonte di dati condivisa, bisogna che accedano ai dati uno alla volta, cioè i loro metodi vanno sincronizzati (*`synchronized`*). Consideriamo il seguente esempio:*

```
1   class CallMe {
2     /*synchronized*/public void call(String msg) {
```

```
3       System.out.print("[" + msg);
4       try {
5         Thread.sleep(1000);
6       }
7       catch (Exception e){};
8       System.out.println("]");
9     }
10  }


11  class Caller implements Runnable {
12    private String msg;
13    private CallMe target;
14    public Caller(Callme t, String s) {
15      target = t;
16      msg = s;
17      new Thread(this).start();
18    }
19    public void run() {
20      //synchronized(target){
21      target.call(msg);
22      //}
23    }
24  }


25  public class Synch {
26    public static void main(String args[]) {
27      CallMe target = new CallMe();
28      new Caller(target, "Hello");
29      new Caller(target, "Synchronized");
```

```
30       new Caller(target, "World");
31     }
32   }
```

Output senza sincronizzazione

```
[Hello[Synchronized[World]
]
]
```

Output con sincronizzazione

```
[Hello]
[Synchronized]
[World]
```

Analisi di `Synch`

Nell'esempio ci sono tre classi: `Synch`, `Caller`, `CallMe`. *Immaginiamo il runtime dell'applicazione, e partiamo dalla classe* `Synch` *che contiene il metodo* `main` *(clicka qui per il supporto dell'applet). Il thread principale, alla riga 27 istanzia un oggetto chiamato* `target` *dalla classe* `CallMe`. *Alla riga 28 istanzia (senza referenziarlo), un oggetto della classe* `Caller`, *al cui costruttore passa l'istanza* `target` *e la stringa "Hello" . A questo punto il thread principale si è spostato nell'istanza appena creata dalla classe* `Caller`, *per eseguirne il costruttore (righe da 14 a 18). In particolare setta come variabili d'istanza, sia l'oggetto* `target` *sia la stringa "Hello" , quest'ultima referenziata come* `msg`. *Inoltre crea e fa partire un thread al cui costruttore viene passato l'oggetto* `this`. *Poi il thread principale, ritorna alla riga 29 istanziando un altro oggetto della classe* `Caller`. *Al costruttore di questo secondo oggetto viene passato la stessa istanza* `target`, *che era stata passata al primo, e la stringa* `msg`. *Prevedibilmente, il costruttore di questo oggetto appena istanziato, setterà le variabili d'istanza* `target` *e* `msg` *(con la stringa "synchronized"), e creerà e farà partire un thread, il cui campo d'azione sarà il metodo* `run` *di questo secondo oggetto* `Caller`. *Presumibilmente, il thread principale creerà un terzo oggetto* `Caller`, *che avrà come variabili d'istanza, sempre lo stesso oggetto* `target` *e la stringa "world" . Anche in questo oggetto viene creato e fatto partire un thread il cui campo d'azione sarà il metodo* `run` *di questo terzo oggetto* `Caller`. *Subito dopo, il thread principale "muore" , e lo scheduler dovrà scegliere quali dei tre thread in stato ready (pronto per l'esecuzione), dovrà essere eseguito [fig. 1 dell'applet]. Avendo tutti i thread la stessa priorità 5 di default, verrà scelto quello che è da più tempo in attesa. Ecco che allora il primo thread creato (che si trova nell'oggetto dove* `msg = "Hello"` *), eseguirà il proprio metodo* `run`, *chiamando il metodo* `call` *sull'istanza* `target`. *Quindi, il primo thread si sposta nel metodo* `call` *dell'oggetto* `target` *(righe da 2 a 9), mettendosi a "dormire" per un secondo[fig. 3], dopo aver stampato una parentesi quadra d'apertura e la*

stringa "Hello" [fig. 2]. A questo punto il secondo thread si impossessa del processore ripetendo in manera speculare le azioni del primo thread. Dopo la chiamata al metodo `call`*, anche il secondo thread si sposta nel metodo* `call` *dell'oggetto* `target`*, mettendosi a "dormire" per un secondo [fig. 5], dopo aver stampato una parentesi quadra d'apertura e la stringa "synchronized" [fig. 4]. Il terzo thread quindi, si sposterà nel metodo* `call` *dell'oggetto* `target`*, mettendosi a "dormire" per un secondo [fig. 7], dopo aver stampato una parentesi quadra d'apertura e la stringa "world" [fig. 6]. Dopo poco meno di un secondo, si risveglierà il primo thread che terminerà la sua esecuzione stampando una parentesi quadra di chiusura ed andando da capo (metodo* `println` *alla riga 8) [fig. 8]. Anche gli altri due thread si comporteranno allo stesso modo, negli attimi successivi [fig. 9 e10], producendo l'output non sincronizzato.*

L'applicazione ha una durata che si può quantificare in circa un secondo.

Per ottenere l'output sincronizzato, basta decommentare il modificatore `synchronized` *anteposto alla dichiarazione del metodo* `call` *(riga 2). Infatti, quando un thread inizia ad eseguire un metodo dichiarato sincronizzato, anche in caso di chiamata al metodo* `sleep`*, non lascia il codice a disposizione di altri thread. Quindi, sino a quando non termina l'esecuzione del metodo sincronizzato, il secondo thread non può eseguire il codice dello stesso metodo. Ovviamente lo stesso discorso si ripete con il secondo ed il terzo thread. L'applicazione quindi ha una durata quantificabile in circa tre secondi e produce un output sincronizzato.*

In alternativa, lo stesso risultato si può ottenere decommentando le righe 20 e 22. In questo caso la keyword `synchronized` *assume il ruolo di comando, tramite la sintassi:*

```
synchronized (nomeOggetto)

{...blocco di codice sincronizzato...}
```

e quando un thread si trova all'interno del blocco di codice, valgono le regole di sincronizzazione sopra menzionate.

Si tratta di un modo di sincronizzare gli oggetti che da un certo punto di vista può risultare più flessibile, anche se più complesso e chiaro. Infatti è possibile utilizzare un metodo di un oggetto in maniera sincronizzata o meno a seconda del contesto. Tuttavia, una situazione in cui un metodo possa essere considerato sincronizzato o meno non è mai richiesta né desiderabile. Si raccomanda al lettore di evitare quindi di utilizzare il comando `synchronized` *ogni volta che ciò risulta possibile.*

Monitor e Lock

Esiste una terminologia ben precisa riguardo la sincronizzazione dei thread. Nei vari testi che riguardano l'argomento, viene definito il concetto di monitor di un oggetto. In Java ogni oggetto ha associato il proprio monitor, se contiene del codice sincronizzato. A livello concettuale un monitor è un oggetto utilizzato come blocco di mutua esclusione per i thread, il che significa che solo un thread può "entrare" in un monitor in un determinato istante.

Java non implementa fisicamente il concetto di monitor di un oggetto, ma questo è facilmente associabile alla parte sincronizzata dell'oggetto stesso. In pratica, se un thread `t1` *entra in un metodo sincronizzato* `ms1` *di un determinato oggetto* `o1`*, nessun altro thread potrà entrare in nessun metodo sincronizzato dell'oggetto* `o1`*, sino a quando* `t1`*, non avrà terminato l'esecuzione del metodo* `ms1` *(ovvero non avrà abbandonato il monitor dell'oggetto) [8]. In particolare si dice che il thread* `t1`*, ha il "lock" dell'oggetto* `o1`*, quando è entrato nel suo monitor (parte sincronizzata).*

È bene conoscere questa terminologia, per interpretare correttamente la documentazione ufficiale. Tuttavia, l'unica attenzione che deve avere il programmatore è l'utilizzo della keyword `synchronized`*.*

La comunicazione fra Thread

Nell'esempio precedente abbiamo visto come la sincronizzazione di thread che condividono gli stessi dati sia facilmente implementabile. Purtroppo, le situazioni che si presenteranno dove bisognerà gestire la sincronizzazione di thread, non saranno sempre così semplici. Come vedremo nel prossimo esempio, la sola keyword `synchronized` *non sempre basta a risolvere i problemi di sincronizzazione fra thread. Descriviamo lo scenario del prossimo esempio. Vogliamo creare una semplice applicazione che simuli la situazione economica ideale, dove c'è un produttore che produce un prodotto, e un consumatore che lo consuma. In questo modo, il produttore non avrà bisogno di un magazzino. Ovviamente le attività del produttore e del consumatore saranno eseguite da due thread lanciati in attività parallele.*

```
// Classe Magazzino ****************************

1   public class WareHouse{
2     private int numberOfProducts;
3     private int idProduct;
4     public synchronized void put(int idProduct) {
5       this.idProduct = idProduct;
6       numberOfProducts++;
```

```
7       printSituation("Produced " + idProduct);
8     }
9     public synchronized int get() {
10      numberOfProducts--;
11      printSituation("Consumed " + idProduct);
12      return idProduct;
13    }
14    private synchronized void printSituation(String msg)
15      System.out.println(msg +"\n" + numberOfProducts
16      + " Product in Warehouse");
17    }
18  }
19
20  //classe Produttore ***************************
21
22    public class Producer implements Runnable {
23      private WareHouse wareHouse;
24      public Producer(WareHouse wareHouse) {
25        this.wareHouse = wareHouse;
26        new Thread(this, "Producer").start();
27      }
28      public void run() {
29        for (int i = 1; i <= 10; i++) {
30          wareHouse.put(i);
31        }
32      }
33    }
34
35  //classe Consumatore ************************
36  public class Consumer implements Runnable {
37    private WareHouse wareHouse;
```

```
38    public Consumer(WareHouse wareHouse) {
39      this.wareHouse = wareHouse;
40      new Thread(this, "Consumer").start();
41    }
42    public void run() {
43      for (int i = 0; i < 10;) {
44        i = wareHouse.get();
45      }
46    }
48  }
49
50  //classe del main *************************
51  public class IdealEconomy {
52    public static void main(String args[]) {
53      WareHouse wareHouse = new WareHouse();
54      new Producer(wareHouse);
55      new Consumer(wareHouse);
56    }
57  }
```

Output Solaris 8

```
C:\TutorialJavaThread\Code>java IdealEconomy

Produced 1

1 Product in Warehouse

Produced 2

2 Product in Warehouse
```

```
Produced 3

3 Product in Warehouse

Produced 4

4 Product in Warehouse

Produced 5

5 Product in Warehouse

Produced 6

6 Product in Warehouse

Produced 7

7 Product in Warehouse

Produced 8

8 Product in Warehouse

Produced 9

9 Product in Warehouse

Produced 10

10 Product in Warehouse
```

```
Consumed 10

9 Product in Warehouse
```

Analisi di `IdealEconomy`

Visto l'output prodotto dal codice l'identificatore della classe `IdealEconomy`*, suona un tantino ironico...* [9]*. Come al solito cerchiamo di immaginare il runtime dell'applicazione, supponendo di lanciare l'applicazione su di un sistema Unix (caso più semplice). La classe* `IdealEconomy`*, fornisce il metodo* `main`*, quindi partiremo dalla riga 53, dove viene istanziato un oggetto* `WareHouse` *(letteralmente "magazzino")* [10]*. Nelle successive due righe, vengono istanziati un oggetto* `Producer` *ed un oggetto* `Consumer`*, ai cui costruttori viene passato lo stesso oggetto* `WareHouse`*, (già si intuisce che il magazzino sarà condiviso fra i due thread). Sia il costruttore di* `Producer` *sia il costruttore di* `Consumer`*, dopo aver settato come variabile d'istanza l'istanza comune di* `WareHouse`*, creano un thread (con nome rispettivamente "Producer" e "Consumer") e lo fanno partire. Una volta che il thread principale ha eseguito entrambi i costruttori, muore, e poiché è stato fatto partire per primo, il thread Producer, passa in stato di esecuzione all'interno del suo metodo* `run`*. Da questo metodo viene chiamato il metodo sincronizzato* `put`*, sull'oggetto* `wareHouse`*. Essendo questo metodo sincronizzato, ne è garantita l'atomicità dell'esecuzione, ma nel contesto corrente ciò non basta a garantire un corretto comportamento dell'applicazione. Infatti, il thread Producer chiamerà il metodo* `put` *per 10 volte [riga 29], per poi terminare il suo ciclo di vita, e lasciare l'esecuzione al thread Producer. Questo eseguirà un'unica volta il metodo get dell'oggetto* `wareHouse`*, per poi terminare il suo ciclo di vita e con esso il ciclo di vita dell'applicazione.*
Sino ad ora, non abbiamo visto ancora dei meccanismi chiari per far comunicare i thread. Il metodo `sleep`*, le priorità e la parola chiave* `synchronized`*, non rappresentano meccanismi sufficienti per fare comunicare i thread. Contrariamente a quanto ci si possa aspettare, questi meccanismi non sono definiti nella classe* `Thread`*, bensì nella classe* `Object` [11]*. Trattasi di metodi dichiarati* `final` *nella classe* `Object` *e pertanto ereditati e da tutte le classi e non modificabili (applicando l'override). Possono essere invocati in un qualsiasi oggetto all'interno di codice sincronizzato. Questi metodi sono:*

- `wait()`*: dice al thread corrente (cioè che legge la chiamata a questo metodo) di abbandonare il monitor e porsi in pausa finché qualche altro thread non entra nello stesso monitor e chiama* `notify()`
- `notify()`*: richiama dallo stato di pausa il primo thread che ha chiamato* `wait()` *nello stesso oggetto*

- `notifyAll()`*: richiama dalla pausa tutti i thread che hanno chiamato* `wait()` *in quello stesso oggetto. Viene fra questi eseguito per primo quello a più alta priorità.*

Allo scopo di far correttamente comportare il runtime dell'applicazione `IdealEconomy`*, andiamo a modificare solamente il codice della classe* `WareHouse`*. In questo modo sarà la stessa istanza di questa classe (ovvero il contesto di esecuzione condiviso dai due thread), a stabilire il comportamento corretto dell'applicazione. In pratica l'oggetto* `wareHouse` *bloccherà (*`wait()`*), il thread Producer una volta realizzato un* `put` *del prodotto, dando via libera al* `get` *del Consumer. Poi notificherà (*`notify()`*) l'avvenuta consumazione del prodotto al Producer, e bloccherà (*`wait()`*) il* `get` *di un prodotto (che non esiste ancora) da parte del thread Consumer. Poi, dopo che il Producer avrà realizzato un secondo* `put` *del prodotto, verrà prima avvertito (*`notify()`*) il Consumer, e poi bloccato il Producer (*`wait()`*). Ovviamente il ciclo si ripete per 10 iterazioni. Per realizzare l'obiettivo si introduce la classica tecnica del flag (*`boolean empty`* che vale true se il magazzino è vuoto). Di seguito troviamo il codice della nuova classe* `WareHouse`*:*

```
1   public class WareHouse{
2     private int numberOfProducts;
3     private int idProduct;
4     private boolean empty = true; // magazzino vuoto
5     public synchronized void put(int idProduct) {
6       if (!empty) // se il magazzino non è vuoto...
7         try {
8           wait(); // fermati Producer
9         }
10        catch (InterruptedException exc) {
11          exc.printStackTrace();
12        }
13      this.idProduct = idProduct;
14      numberOfProducts++;
15      printSituation("Produced " + idProduct);
16      empty = false;
17      notify(); // svegliati Consumer
18    }
```

```
19    public synchronized int get() {
20      if (empty) // se il magazzino è vuoto...
21        try {
22          wait(); // bloccati Consumer
23        }
24        catch (InterruptedException exc) {
25          exc.printStackTrace();
26        }
27      numberOfProducts--;
28      printSituation("Consumed " + idProduct);
29      empty = true; // il magazzino ora è vuoto
30      notify(); // svegliati Producer
31      return idProduct;
32    }
33    private synchronized void printSituation(String msg) {
34      System.out.println(msg +"\n" + numberOfProducts +
35      " Product in Warehouse");
36    }
37  }
```

Immaginiamo il runtime relativo a questo codice. Il thread Producer chiamerà il metodo `put` *passandogli* `1` *alla prima iterazione. Alla riga 6 viene controllato il flag* `empty`*, siccome il suo valore è* `true`*, non sarà chiamato il metodo* `wait()`*. Quindi viene settato l'*`idProduct`*, incrementato il* `numberOfProducts`*, e chiamato il metodo* `printSituation`*. Poi, il flag* `empty` *viene settato a* `true` *(dato che il magazzino non è più vuoto) e viene invocato il metodo* `notify()`*. Quest'ultimo non ha nessun effetto dal momento che non ci sono altri thread in attesa nel monitor di questo oggetto. Poi riviene chiamato il metodo* `put` *dal thread Producer con argomento* `2`*. Questa volta il controllo alla riga 6 è verificato (*`empty` *è* `false` *cioè il magazzino non è vuoto), e quindi il thread Producer va a chiamare il metodo* `wait()` *rilasciando il lock dell'oggetto. A questo punto il thread Consumer va ad eseguire il metodo* `get`*. Il controllo alla riga 20, fallisce perchè* `empty` *è* `true`*, e quindi non viene chiamato il metodo* `wait()`*. Viene decrementato il* `numberOfProducts` *a* `0`*, chiamato il metodo* `printSituation`*, viene settato il flag* `empty` *a* `true`*. Poi viene*

chiamato il metodo `notify()` *che toglie il thread Producer dallo stato di* `wait()`*. Infine viene ritornato il valore di* `idProduct`*, che è ancora* `1`*. Poi viene richiamato il metodo* `get`*, ma il controllo alla riga 20 è verificato e il thread Consumer si mette in stato di* `wait()`*. Quindi il thread Producer continua la sua esecuzione da dove si era bloccato cioè dalla riga 8. Quindi viene settato l'*`idProduct` *a* `2`*, incrementato di nuova ad* `1` *il* `numberOfProducts`*, e chiamato il metodo* `printSituation`*. Poi, il flag* `empty` *viene settato a* `true` *e viene chiamato il metodo* `notify()` *che risveglia il thread Consumer...*

Conclusioni:

La gestione dei thread in Java, può considerarsi semplice, soprattutto se paragonata alla gestione dei thread in altri linguaggi. Inoltre, una classe (`Thread`*), un'interfaccia (*`Runnable`*), una parola chiave (*`synchronized`*) e tre metodi (*`wait()`*,* `notify()` *e* `notifyAll()`*), rappresentano il nucleo di conoscenza fondamentale per gestire applicazioni multi-threaded. In realtà le situazioni multi-threaded che bisognerà gestire nelle applicazioni reali, non saranno tutte così semplici come nell'ultimo esempio. Tuttavia la tecnologia Java fa ampio uso del multi-threading. La notizia positiva, è che la complessità del multi-threading, è gestita dalla tecnologia stessa. Consideriamo ad esempio la tecnologia Java Servlet (http://java.sun.com/products/servlet/). La tecnologia Java Servlet nasce come alternativa alla tecnologia C.G.I. (Common Gateway Interface) ma è in grado di gestire il multi- threading in maniera automatica, allocando thread diversi per servire ogni richiesta del client. Quindi, lo sviluppatore è completamente esonerato dal dover scrivere codice per la gestione dei thread. Anche altre tecnologie Java (Java Server Pages...), gestiscono il multi-threading automaticamente, ed alcune classi della libreria Java (java.nio.channels.SocketChannel, java.util.Timer...), semplificano enormemente il rapporto tra lo sviluppatore ed i thread. Ma senza conoscere l'argomento, è difficile utilizzare questi strumenti in maniera "consapevole" .*

Note:

[1]: *Il multi-threading non deve essere confuso con il multi-tasking. Possiamo definire "task" i processi che possono essere definiti "pesanti" , per esempio Word ed Excel. In un sistema operativo che supporta il multi-tasking, e possibile lanciare più task contemporaneamente. La precedente affermazione, può risultare scontata per molti lettori, ma negli anni '80 l' "Home Computer" , aveva spesso come sistema DOS, chiaramente non multi-tasking. I task hanno spazi di indirizzi separati, e la comunicazione fra loro è limitata.*

I thread possono essere definiti come processi "leggeri" , che condividono lo stesso spazio degli indirizzi e lo stesso processo pesante in cooperazione. I thread hanno quindi la caratteristica fondamentale di poter "comunicare" al fine di ottimizzare l'esecuzione dell'applicazione in cui sono definiti. *[torna indietro]*

[2]: *Non considereremo altri thread (meno gestibili) eventualmente creati della JVM come quello della Garbage Collection.* *[torna indietro]*

[3]: *Come vedremo più avanti il concetto di priorità, NON è la chiave per gestire i thread. Infatti, limitandoci alla sola gestione delle priorità. la nostra applicazione multi-threaded potrebbe comportarsi in maniera differente su sistemi diversi. Pensiamo solo al fatto che non tutti i sistemi operativi utilizzano una scala da 1 a 10 per le priorità, e che per esempio Unix e Windows hanno thread scheduler con filosofie completamente differenti.* *[torna indietro]*

[4]: *La chiamata al metodo* `start` *per eseguire il metodo* `run`*, fa sì che il thread* `t` *vada prima o poi ad eseguire il metodo* `run`*. Invece, una eventuale chiamata del metodo* `run`*, non produrrebbe altro che una normale esecuzione dello stesso da parte del thread principale* `ct`*: non ci sarebbe multi-threading.* *[torna indietro]*

[5]: *Notiamo come il metodo* `sleep` *sia statico. Infatti, viene mandato "a dormire" il thread che esegue il metodo.* *[torna indietro]*

[6]: *E' anche possibile (ma non consigliato) creare un thread che utilizza un* `CounterThread` *come oggetto* `Runnable`**:**

```
Thread t = new Thread(new CounterThread());

t.start();
```

[torna indietro]

[7]: *Tale comportamento è però non deterministico, e quindi l'output prodotto, cambierà anche radicalmente ad ogni esecusione dell'applicazione.* *[torna indietro]*

[8]: *C'è da dire che nell'occasione in cui abbiamo è stato generato tale output, siamo stati addirittura fortunati.Oppure, come vedremo nel prossimo paragrafo, non viene chiamato il metodo* `wait`*.*

[torna indietro]

[9]: *L'output in questione è stato generato su di un sistema Unix, (comportamento preemptive), ed è l'output obbligato per ogni esecuzione dell'applicazione. C'è da dire che se l'applicazione fosse stata lanciata su Windows, l'output avrebbe potuto variare drasticamente da esecuzione a esecuzione. Su Windows l'output "migliore" , sarebbe proprio quello è standard su Unix. Il lettore può lanciare l'applicazione più volte per conferma su un sistema Windows.* *[torna indietro]*

[10]: *La scelta dell'identificatore magazzino, può anche non essere condivisa da qualcuno. Abbiamo deciso di pensare ad un magazzino, perchè il nostro obbiettivo è quello di tenerlo sempre vuoto.* *[torna indietro]*

[11]: *In realtà esistono dei metodi nella classe Thread che realizzano una comunicazione tra thread:* `suspend()` *e* `resume()`*. Questi però per sono attualmente deprecati, per colpa della facilità dell'alta probabilità di produrre "DeadLock" . Il Deadlock è una condizione di errore difficile da risolvere, in cui due thread stanno in reciproca dipendenza in due oggetti sincronizzati. (Esempio veloce: il thread* `t1` *si trova nel metodo sincronizzato* `m1` *dell'oggetto* `o1` *(di cui quindi possiede il lock), il thread* `t2` *si trova nel metodo sincronizzato* `m2` *dell'oggetto* `o2` *(di cui quindi possiede il lock). Se il thread* `t1` *prova a chiamare il metodo* `m2`*, si bloccherà in attesa che il thread* `t2` *rilasci il lock dell'oggetto* `o2`*. Se poi anche il thread* `t2` *prova a chiamare il metodo* `m1`*, si bloccherà in attesa che il thread* `t1` *rilasci il lock dell'oggetto* `o1`*. L'applicazione rimarrà bloccata in attesa di una interruzione da parte dell'utente.)*

Come si può intuire anche dai solo identificatori, i metodi `suspend()` *e* `resume()`*, più che realizzare una comunicazione tra thread di pari dignità, implicava l'esistenza di thread "superiori"che avevano il compito di gestirne altri. Poiché si tratta di metodi deprecati, non aggiungeremo altro.*

[torna indietro]

Bibliografia:

- Appunti delle lezioni del Prof. Giuseppe "Geppo" Coppola, (grazie di tutto Consulente!)
- "Il Manuale Java" Patrick Naughton, Mc Graw-Hill

- P.Heller, S.Roberts "The Complete Java 2 Certification Study Guide" Sybex
- Bruce Eckel "Thinking In Java 2nd Edition" MindView Publishing: http://www.bruceeckel.com
- VVAA: "The Java Tutorial" http://www.java.sun.com
- James Jaworsky "Java 2 Tutto e Oltre" Apogeo
