Thread in Java

Fino a qualche tempo fa, prima dell'avvento dei moderni sistemi operativi, era ragionevole descrivere un computer come una macchina in grado di eseguire un programma per volta utilizzando, a tale scopo, una CPU dedicata allo svolgimento delle necessarie computazioni e una memoria che si occupasse di conservare le informazioni relative al programma in esecuzione in un determinato istante.

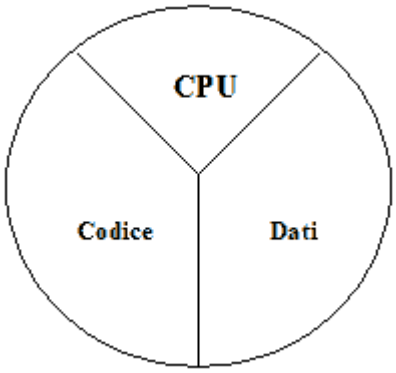
Oggi, come sappiamo, sui nostri sistemi è possibile eseguire svariate operazioni contemporaneamente (o, per lo meno, questa è la sensazione che si ha!). Ad esempio, potremmo ascoltare musica, scrivere una e-mail ed eseguire una scansione con l'antivirus senza preoccuparci di dover completare un'operazione prima di eseguirne una successiva.

Ma, se si escludono i sistemi che lavorano con più processori in parallelo, in fondo la struttura di un computer è rimasta basata su un singolo microprocessore e una memoria. Come si spiega, dunque, un simile comportamento?

Mettiamo da parte l'hardware e ragioniamo da un punto di vista legato alla programmazione. In base a quanto detto precedentemente, eseguire più di un programma contemporaneamente è, in fondo, la stessa cosa di ad avere un computer (e, quindi, una CPU) adibito per ogni singolo programma.

Un Thread, può essere considerato, allora come una CPU virtuale che incapsuli al suo interno i dati e il codice di un particolare programma, come rappresentato in figura:

Figura 1. La rappresentazione di un thread



In termini più generici, avvalendoci delle definizioni più classiche possiamo dire che:

1. Un Processo è un programma in esecuzione
2. Un Thread è un processo che appartiene ad un programma o ad un altro processo.

In Java, la classe ad hoc che implementa una CPU virtuale è la java.lang.Thread. E' importante però, prima di vedere il codice java all'opera, fissare bene i seguenti due punti:

A. Due o più thread possono condividere, indipendentemente dai dati, il codice che essi eseguono. Questo avviene quando tali thread eseguono il loro codice da istanze della stessa classe.
B. Due o più thread possono condividere, indipendentemente dal codice, i dati su cui eseguono delle operazioni. Questo avviene quando tali thread condividono l'accesso ad un oggetto comune.

Come creare un Thread

Ci sono, fondamentalmente, due metodi per creare un thread in Java. Quello che ci sentiamo di consigliare, soprattutto per la sua struttura Object Oriented, è basato sul seguente costruttore:

```
public Thread (Runnable target)
```

ovvero fa uso di un parametro di tipo Runnable, che costituisce la classe che si desidera esegua del codice in modo indipendente dal processo che la manda in esecuzione. Runnable è un'interfaccia che contiene il seguente metodo

```
void run()
```

che dovrà, pertanto, essere implementato dalla classe da "dare in pasto" al costruttore del thread. Vediamo un esempio per chiarire meglio quanto detto:

Listato 1. Esempio di implementazione di un thread

```
public class SimpleThread
{
 Public static void main(String[] args)
 {
  SimpleRunner r = new SimpleRunner();
  Thread t = new Thread(r);
  t.start();
 }
}

Class SimpleRunner implements Runnable
{
 int i;

 public void run()
 {
  i = 0;

  while (true)
  {
   System.out.println("Ciao " + i++);
   if ( i == 20)
   {
    break;
   }
  }
 }
}
```

Come si può osservare, il main() crea un'istanza r della classe SimpleRunner. Tale istanza avrà accesso ai propri dati che, in questo caso, sono rappresentati semplicemente dalla variabile intera i. Poiché l'istanza r viene passata al costruttore della classe Thread, si evince che la variabile i rappresenterà il dato sul quale opererà il thread t quando andrà in esecuzione.

Ogni thread inizia sempre attraverso l'invocazione del metodo run() dell'istanza di tipo Runnable che il thread stesso carica in fase di costruzione (nel nostro esempio r). Il metodo run(), a sua volta, viene invocato quando si effettua una chiamata al metodo start() del thread.

Un ambiente di programmazione multithread consente di creare più thread basati sulla medesima istanza di tipo Runnable. Ad esempio, sarà possibile scrivere:

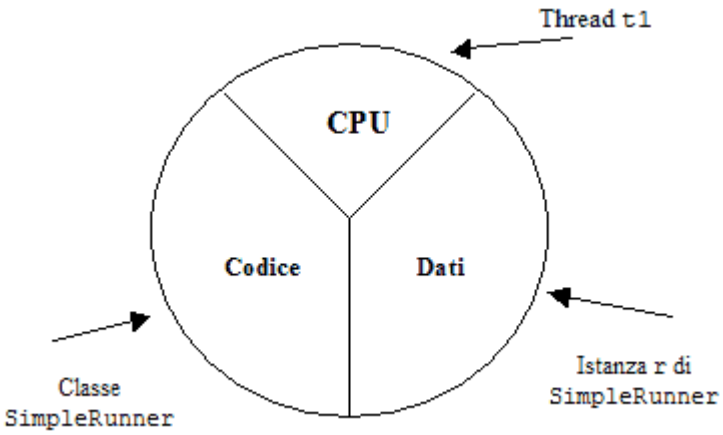
Listato 2. Dichiarare due istanze di thread

```
Thread t1 = new Thread (r); Thread t2 = new Thread (r);
```

In tal modo avremo creato due thread (t1 e t2) che condividono lo stesso codice e gli stessi dati.

Lo schema seguente, riassume il processo di creazione di un thread, relativamente all'esempio prima visto.

Figura 2. La creazione di un Thread



Gli stati di un thread

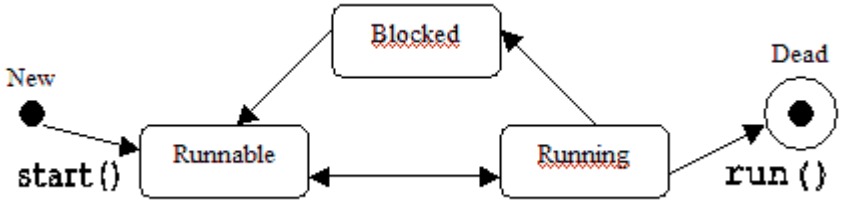
Abbiamo visto come fare per mandare in esecuzione un thread. E' importante sottolineare una cosa: quando viene invocato il metodo start()su un thread non è assolutamente detto che venga fatto partire il thread stesso immediatamente dopo tale invocazione. Quello che sicuramente avviene, in questo caso, è che il thread assume lo stato di Runnable, ovvero è pronto per essere eseguito.

In generale, è importante sapere che possono esserci molti thread nello stato Runnable ma soltanto uno è quello che, in un determinato istante, sarà in esecuzione, ovvero si troverà nello stato Running.

Un thread in esecuzione continua a rimanere nello stato Running fino a quando esso non cessi di essere Runnable oppure finché non sopraggiunga un altro thread con priorità maggiore che abbia il proprio stato a Runnable.

Quando un thread cessa di essere Runnable, si dice che esso passa allo stato Blocked. Esistono molteplici ragioni per le quali un thread potrebbe passare dallo stato Runnable a quello Blocked. Ad esempio, la causa potrebbe essere una semplice chiamata al metodo Thread.sleep(), che impone che il thread corrente interrompa la propria esecuzione per un periodo di tempo prefissato. Il seguente State Diagram illustra i concetti appena esposti:

Figura 2. Diagramma degli stati per un thread



Poiché i thread Java non garantiscono una suddivisione paritetica del tempo tra tutti i thread in gioco, è necessario agire a livello di programmazione per assicurare che il codice di un singolo thread conceda a tutti gli altri thread un'opportunità per essere eseguiti. Un simile risultato si ottiene utilizzando il metodo sleep(), come mostrato nel codice seguente:

Listato 3. Mettere un thread in attesa con sleep()

```
public class MyRunner implements Runnable
{
 public void run()
 {
  while (true)
  {
   // Esegue svariate azioni
   // ...
   // Concede una opportunità di esecuzione agli altri thread
   try
   {
    Thread.sleep(10);
   }
   catch (InterruptedException exc)
   {
    // Lo stato di sleeping è stato interrotto
    // da un altro thread
   }
  }
 }
}
```

Come si può notare dal codice, è necessario utilizzare un costrutto try...catch per poter invocare il metodo Thread.sleep(). Infatti, è sempre possibile che un altro thread invochi il

metodo interrupt() del thread che si trova in sleeping, interrompendone lo stato di pausa attraverso un'eccezione di tipo InterruptedException.

Il metodo sleep(), come si evince dal codice, è un metodo statico che opera sempre sul thread corrente (ovvero quello in esecuzione in quel momento) e non su un thread specifico.

Come terminare un thread

Innanzitutto è importante sapere che quando un thread termina la sua esecuzione, lo stesso thread non può essere eseguito nuovamente.

Una tecnica elegante utilizzata per stoppare un thread è quella di utilizzare un flag, verificato dal metodo run(), che aiuti a stabilire quando il metodo run() stesso debba essere completato. Vediamo un esempio:

Listato 4. Classe con metodo per interrompere l'esecuzione del thread

```
public class SimpleRunner implements Runnable
{
 private boolean stopThread = false;

 public void run()
 {
  while (! stopThread)
  {
   // Esegue qualcosa fino a quando la
   // variabile stopThread è false
  }
  // esecuzione di eventuali operazioni di "pulizia"
//...
 }

 public void stopRunning()
 {
  stopThread = true;
 }
}

public class ThreadController
{
 private SimpleRunner r = new SimpleRunner();
 private Thread t = new Thread(r);

 public void startThread()
 {
  t.start();
 }

 public void stopThread()
 {
  r.stopRunning();
```

```
 }
}
```

I metodi per controllare il comportamento dei thread

Vediamo, adesso, come procedere quando si vogliono ricavare informazioni o effettuare controlli sui thread.

Un thread è contraddistinto da una priorità, ovvero un valore intero che ne indichi l'importanza rispetto ad altri (abbiamo già visto che un thread con priorità più alta rispetto ad un altro che si trovi in stato di sleeping può interromperne lo stato stesso). La classe Thread definisce il metodo `getPriority()` per ricavare la priorità di un thread ed il corrispettivo `setPriority()` per impostarla. Se non viene definita alcuna priorità, verrà assegnato per default il valore `Thread.NORM_PRIORITY`. Altri valori predefiniti sono: `Thread.MIN_PRIORITY` e `Thread.MAX_PRIORITY`.

Un metodo utile per riuscire a stabilire la condizione di un thread in un determinato istante è: `isAlive()`. Non bisogna farsi, però, ingannare e ritenere che tale metodo restituisca `true` soltanto se il thread è correntemente in esecuzione (ovvero in stato running). Infatti, un thread è "vivo" semplicemente se ne è stato invocato il metodo `start()` ed il thread stesso non ha ancora completato il suo ciclo di istruzioni. Quindi, sia che un thread si trovi nello stato sleeping, runnable o blocked il metodo `isAlive()` restituirà sempre il valore true.

Abbiamo visto in azione il metodo `sleep()`. Altri due metodi sono in grado di mettere un thread in attesa. La loro definizione è:

- `public final void` **`join()`**
- `public static void` **`yield()`**

Il primo metodo viene richiamato su un thread specifico e ha lo scopo di mettere in attesa il thread attualmente in esecuzione fino a quando il thread su cui è stato invocato il metodo join() non termini. Vediamo un esempio per chiarire meglio il concetto:

Listato 5. Esempio di uso di join()

```
public static void main (String[] args)
{
 Thread t = new Thread (new SimpleRunner());
 t.start();

 // Vengono eseguite delle operazioni in parallelo con altri thread
 ...
 ...

// A questo punto il thread corrente entra in attesa fino a quando il
// thread t termina
```

```
 try
 {
  t.join();
 }
 catch (InterruptedException exc)
 {

 }

 // A questo punto il thread corrente è in grado di continuare
 ...
 ...
}
```

Il metodo statico `yield()`, mette temporaneamente in pausa il thread corrente e consente ad altri thread in stato Runnable (qualora ve ne siano) di avere una chance per essere eseguiti. Nel caso non vi fossero altri thread con tali requisiti, il metodo `yield()` non avrà alcun effetto.

La Sincronizzazione

Abbiamo detto, in precedenza, che due o più thread possono condividere le stesse risorse. Tale condizione, però, se non attentamente gestita potrebbe essere fonte di grossi problemi in fase di esecuzione di un programma. Vediamo un esempio che ci aiuti a capire il problema.

Supponiamo di avere la seguente classe che implementi uno stack:

Listato 6. Classe SimpleStack

```
public class SimpleStack
{
 int index = 0;
 char [] data = new char[6];

 public void push(char c)
 {
  data [index] = c;
  index++;
 }

 public char pop()
 {
  index--;
  return data[index];
 }
}
```

Come si può notare analizzando il codice non vi è alcun controllo per eventuali errori di overflow (se si cerca, ad esempio, di inserire un elemento sullo stack quando questo contenga già 6 elementi) o di underflow (se si cerca di eliminare un elemento dalla pila quando la pila stessa è già vuota).

La proprietà index, inoltre, contiene sempre il valore che dovrà essere utilizzato al momento di effettuare un'operazione di push o di pop.

Si supponga, adesso che due thread agiscano su una singola istanza della classe SimpleStack. Uno dei due thread effettua delle operazioni di push e l'altro, in modo più o meno indipendente, elimina i dati dallo stack attraverso operazioni di pop.

Se i due thread agissero in modo sempre ordinato, ovvero eseguendo sempre prima un'operazione di push seguita da una pop, la gestione della pila non causerà mai problemi. Ma, prevedere un comportamento del genere è assolutamente impossibile quando si ha a che fare con i thread e, pertanto, potrebbero presentarsi delle situazioni anomale. Vediamone una possibile:

Supponiamo che ad un certo punto dell'esecuzione, lo stack da noi creato si trovi nella seguente situazione (il carattere '^' è utilizzato come freccia per indicare il successivo indice disponibile):

```
| a | b |   |   |   |   |

          ^


index = 2
```

Definiamo un primo thread con il simbolo t1 ed un secondo con t2. Il thread t1 invoca, quindi, il metodo push(), per inserire il carattere 'c'. Viene effettuata l'istruzione:

```
data[index] = 'c';
```

ma prima di incrementare il valore di index e portarlo a 3, t1 va nello stato sleeping e t2 passa nello stato running. Il nuovo stato dello stack è, pertanto:

```
| a | b | c |   |   |   |

          ^


index = 2
```

Come si può notare, vi è già un'incongruenza dato che la proprietà index contiene un valore che punta ad una cella non vuota dello stack. Vediamo cosa accade proseguendo l'esecuzione.

t2 esegue, quindi, un'operazione di pop al termine della quale si otterrà:

```
| a | b | c |   |   |   |

      ^


index = 1
```

```
valore restituito: 'b'
```

In quell'istante, t1 riprende la sua esecuzione da dove l'aveva interrotta e, pertanto, incrementa il valore di index, che adesso (a causa dell'intervento precedente di t2) varrà 2:

```
| a | b | c |   |   |   |


          ^


index = 2
```

In pratica, la successiva cella vuota risulterebbe quella in cui è contenuto il carattere 'c'.

Se da questo momento in poi i due thread t1 e t2 proseguissero in modo perfettamente alternato le loro operazioni (iniziando da un'operazione di pop), avremmo una situazione tale per cui il carattere 'c' non verrebbe mai restituito attraverso il metodo pop() e il carattere 'b' risulterà presente due volte sullo stack.

Come fare, allora, per evitare problemi del genere? In Java ogni oggetto ha un flag associato ad esso, denominato lock flag. La parola chiave synchronized permette l'interazione con questo flag e fornisce un accesso esclusivo al codice che gestisce dei dati condivisi. Basterà, dunque, modificare i metodi dello stack nel seguente modo:

Listato 7. Sincronizzazione dello stack

```
public void push(char c)
{
 synchronized (this)
 {
  data [index] = c;
  index++;
 }
}

public char pop()
{
 synchronized (this)
 {
  index--;
  return data[index];
 }
}
```

Grazie a tale intervento, adesso, i dati condivisi verranno utilizzati in modi sincronizzato dai singoli thread senza rischiare di risultati errati o inconsistenti.