Paradigmi di programmazione (sperimentazioni) 2000-01

JAVA - Thread

Alberto Martelli

THREAD

Thread: flusso sequenziale di controllo (esecuzione di istruzioni) in un programma.

Nello stesso programma si possono far partire più *thread* che sono eseguiti concorrentemente.

Tutti i *thread* condividono le stesse variabili del programma, a differenza dai *processi* che hanno ciascuno il proprio contesto (*lightweight process*).

Nei computer a singola CPU la concorrenza viene simulata con una politica di *scheduling* che alterna l'esecuzione dei singoli *thread*.

Una applicazione Java che usa i thread può eseguire più attività contemporaneamente. Esempio: aggiornare l'informazione grafica sullo schermo e accedere alla rete.

Abbiamo già visto che quando un main crea una finestra, viene attivato un **thread** di interfaccia utente, diverso da quello del main.

In alcuni casi i thread possono procedere in modo indipendente uno dall'altro (comportamento asincrono), in altri devono essere sincronizzati fra loro (es. produttore - consumatore).

Come si crea un thread

Si definisce una classe che eredita da **Thread** e che implementa un metodo **run().**
Quando si crea un oggetto di questa classe si esegue il suo metodo **start()** per far partire il *thread*.

```
class MiaClasse extends Thread {
   public void run() {
      System.out.println("Sono il thread " + getName());
}}

....
MiaClasse t1 = new MiaClasse();
MiaClasse t2 = new MiaClasse();
t1.start();
t2.start();
```

Per avviare un thread si deve eseguire il metodo **start()**, che si occupa di inizializzare un nuovo thread e poi chiama **run()**.

Non è possibile chiamare direttamente il metodo **run()** perché questo non creerebbe un nuovo thread.

L'ereditarietà è usata in modo poco naturale.

Non ha molto senso dire che **MiaClasse** *è un* **Thread**.

Quello che si intende è che è possibile attivare un nuovo thread facendogli eseguire il metodo **run()** della **MiaClasse**.

Altro modo di creare un thread

Si definisce una classe che implementa l'interfaccia **`Runnable`** che possiede il metodo **`run`**.

```
class Esempio implements Runnable {
   public void run() {...}}
```

Per attivare un thread:

```
Esempio es = new Esempio();
Thread t = new Thread(es),
t.start();
```

```
class MiaClasse implements Runnable {
   public void run() {
      System.out.println("Sono il thread " +
        Thread.currentThread().getName());
}}

....
MiaClasse mt = new MiaClasse();
Thread t1 = new Thread(mt);
Thread t2 = new Thread(mt);
t1.start();
t2.start();
```

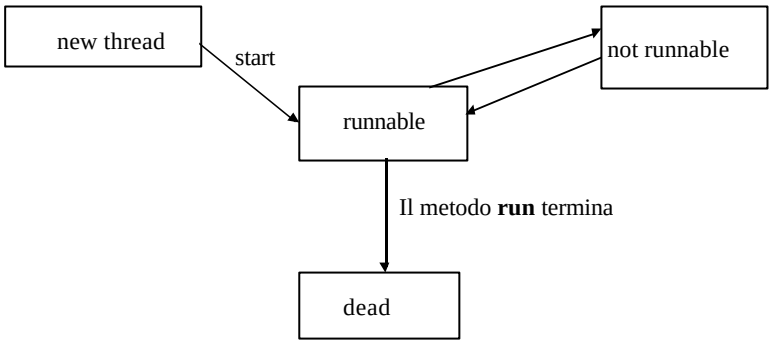


```
class MiaClasse implements Runnable {
   int i = 0;
   public void run() {
      i++;
      System.out.println(i);
}}

....
MiaClasse mt = new MiaClasse();
Thread t1 = new Thread(mt);
Thread t2 = new Thread(mt);
t1.start();
t2.start();
```

Viene stampato prima 1 e poi 2.
Infatti esiste un unico oggetto della **MiaClasse**, legato alla variabile **mt**, che ha una variabile locale **i**. I thread **t1** e **t2** eseguono entrambi il metodo **run** di questo oggetto, incrementando la stessa variabile **i**.

Ciclo di vita di un thread



Un thread **runnable** *può* essere eseguito. L'effettiva esecuzione dipende dalla politica dello scheduler.

Dopo essere stato attivato, un thread può bloccarsi se

- chiama il metodo *sleep*
- esegue una *wait* (riparte con *notify*)
- sta aspettando una operazione di I/O

Il thread si ferma quando la *run* termina.

Esistono anche dei metodo *suspend*, *resume* e *stop*, ma sono *deprecati*.

E' possibile assegnare ai thread una **priorità** da 1 a 10.

Il thread eseguibile con la priorità maggiore rimane in esecuzione fino a quando:

- cede il controllo chiamando il metodo *yield*;
- smette di essere eseguibile;
- un thread di priorità superiore diventa eseguibile.

Se ci sono più thread con la stessa priorità, Java non garantisce la politica con cui i thread saranno gestiti. Se si vuole dare la possibilità a tutti i thread di essere attivati, è conveniente far eseguire ogni tanto una *sleep* ad ogni thread.

E' possibile definire *gruppi di thread* mediante la classe **ThreadGroup**.

In questo modo è possibile gestire tutti i thread del gruppo in modo omogeneo.

Sincronizzazione

Il meccanismo di sincronizzazione di Java si basa sulla nozione di *monitor*.

Per ogni classe in Java è possibile definire dei metodi **synchronized**.

I metodi **synchronized** realizzano delle *sezioni critiche*.

Se una classe contiene dei metodi **synchronized**, Java associa un **lock** ad ogni oggetto della classe.

Quando un thread chiama un metodo sincronizzato, l'oggetto diventa bloccato (locked). Altri thread che tentino di accedere allo stesso oggetto chiamando metodi sincronizzati rimangono bloccati fino a quando il thread precedente non rilascia l'oggetto, terminando l'esecuzione del metodo.

E' possibile bloccare un oggetto senza usare un metodo sincronizzato, mediante un *blocco sincronizzato*.

```
synchronized (obj) {
      ... codice del blocco sincronizzato ...}
```

blocca l'oggetto **obj** per tutta l'esecuzione del blocco.

Da un metodo sincronizzato si possono chiamare i metodi:

wait() - sblocca l'oggetto e mette il thread che lo ha eseguito in una coda di attesa associata all'oggetto;

notify() - risveglia un thread a caso fra quelli in attesa, dandogli la possibilità di competere per il lock e di riprendere l'esecuzione dal punto in cui si era messo in wait;

notifyAll() - risveglia tutti i thread in attesa (ne entra solo uno per volta).

Sono metodi di **Object** e quindi vengono ereditati da qualunque classe. Se si tenta di chiamarli da un metodo non sincronizzato, si ha un errore a runtime.

Es. **produttore-consumatore** (dal Tutorial).
Come realizzare un buffer di un elemento.

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }
```

```
    public synchronized void put(int value) {
           while (available == true) {
               try {
                   wait();
               } catch (InterruptedException e) { }
           }
           contents = value;
           available = true;
           notifyAll();
       }
   }

   // fine di CubbyHole
```



```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                               + " got: " + value);
        }
    }
}
```



```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```