19.
Introduzione al multi-threading

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

I thread

- I **thread**, o *processi leggeri*, sono flussi di esecuzione all'interno di un processo in corso
- In altre parole, un processo può essere suddiviso in vari thread, ciascuno dei quali rappresenta un flusso di esecuzione indipendente dagli altri

- I thread appartenenti allo stesso processo condividono quasi tutte le risorse, come la **memoria** e i file aperti, tranne:
  - il program counter
  - lo stack
- Il program counter e lo stack sono proprio quelle risorse che consentono ad un thread di avere un flusso di esecuzione indipendente

I thread in Java

- Java è stato il primo tra i linguaggi di programmazione maggiormente utilizzati ad offrire un **supporto nativo** ai thread
- Ad esempio, per supportare i thread i linguaggi C/C++ necessitano di librerie esterne, spesso fornite dal sistema operativo (come la libreria per i thread POSIX)
- Siccome la virtual machine di Java funge anche da sistema operativo per i programmi Java, essa offre in maniera nativa il supporto ai thread

- Nota: il C++11 ha introdotto il supporto ai thread nella sua libreria standard

La classe Thread

- Esamineremo due modi alternativi di creare un thread
- Entrambi i modi sono supportati dalla **classe Thread**
- Per evitare confusione tra i thread e gli oggetti della classe Thread, chiameremo “thread di esecuzione” i primi e “oggetti Thread” i secondi

Oggetti Thread e thread di esecuzione

- In Java, ad ogni thread di esecuzione è associato un oggetto Thread
- Il viceversa non è sempre vero: un oggetto Thread può non avere un corrispondente thread di esecuzione, perché quest'ultimo non è ancora partito, oppure perché è già terminato

Applicazioni e thread

- Una applicazione Java termina quando **tutti** i suoi thread sono terminati
- Ogni applicazione Java parte con almeno un thread, detto thread principale (*main thread*), che esegue il metodo main della classe di partenza
- Anche al thread principale è associato, in maniera automatica, un oggetto Thread; in seguito vedremo come ottenere un riferimento a questo oggetto

Creazione di un thread

- Il primo modo di creare un thread di esecuzione consiste nei seguenti passi:

  1) Creare una classe X che estenda Thread
  2) Affinché il nuovo thread di esecuzione faccia qualcosa, la classe X deve effettuare l'overriding del metodo “run”, la cui intestazione in Thread è semplicemente

```
public void run()
```

  3) Istanziare la classe X
  4) Invocare il metodo **start** dell'oggetto creato

- Naturalmente, all'occorrenza il procedimento può essere semplificato utilizzando una classe X anonima

Esempio di creazione di un thread di esecuzione

- Ad esempio, creiamo un thread di esecuzione che stampa i numeri da 0 a 9, con una pausa di un secondo tra un numero e il successivo
- A questo scopo, creiamo una classe che estende Thread, il cui metodo run svolge il compito prefissato

```
public class MyThread extends Thread {
     @Override
     public void run() {
          for (int i=0; i<10 ;i++) {
               System.out.println(i);
               try {
                    Thread.sleep(1000);
               } catch (InterruptedException e) {
                    return;
               }
          }
     }
}
```

metodo bloccante

- A questo punto, istanziamo la classe MyThread e facciamo partire il corrispondente thread di esecuzione

```
MyThread t = new MyThread();
t.start();
```

metodo non bloccante

Esempio (continua)

- Non abbiamo dotato la classe MyThread di un costruttore in quanto **Thread ha un costruttore senza argomenti**, che è sufficiente per i nostri scopi

- Osserviamo che prima di chiamare il metodo start, c'è un oggetto di tipo Thread a cui non corrisponde (ancora) nessun thread di esecuzione

- L'intestazione di start in Thread è semplicemente

```
public void start()
```

- La chiamata a start non è bloccante; per definizione, il nuovo thread di esecuzione svolge le sue operazioni in parallelo al resto del programma

- **Non è consentito invocare start più di una volta** sullo stesso oggetto thread, anche se la prima esecuzione del thread è terminata

Esempio (continua)

- Il nuovo thread di esecuzione esegue automaticamente il metodo **run** dell'oggetto thread corrispondente

- Il metodo run viene anche detto l'*entry point* del thread, perché è il primo metodo che viene eseguito
- In questo senso, il metodo main è l'entry point del thread principale

- Quando il metodo run termina, anche il thread di esecuzione termina

Il metodo sleep

- Nell'esempio abbiamo usato anche un'altro metodo della classe Thread:

```
public static void sleep(long millis) throws InterruptedException
```

- Tale metodo statico mette in attesa il thread corrente (cioè, quello che invoca sleep) per un dato numero di millisecondi
- Se l'attesa viene interrotta, e vedremo in seguito come, il metodo lancia l'**eccezione verificata** InterruptedException
- Questa è una caratteristica comune a tutti i metodi cosiddetti “bloccanti”, che cioè possono mettere in attesa un thread

- Domanda: perché secondo voi è stato deciso che l'eccezione lanciata fosse verificata?

- Quando si cattura l'eccezione InterruptedException, è buona norma terminare il thread di esecuzione corrente (si veda la slide “La disciplina delle interruzioni”)
- Se ci si trova nel metodo run di un oggetto thread, per terminare il thread corrente è sufficiente utilizzare “return”

Esercizio 1

[27/3/08, #3]

Implementare un metodo statico **delayIterator** che prende come argomenti un iteratore "i" ed un numero intero "n", e restituisce un nuovo iteratore dello stesso tipo di "i", che restituisce gli stessi elementi di "i", ma in cui ogni elemento viene restituito (dal metodo next) dopo un ritardo di "n" secondi.

Suggerimento: se possibile, utilizzare classi anonime.

Esempio d'uso:

```
List<Integer> l = new LinkedList<Integer>();
l.add(3);
l.add(4);
l.add(177);

Iterator<Integer> i = delayIterator(l.iterator(), 2);
while (i.hasNext()) {
    System.out.println(i.next());
}
```

Output: il programma stampa il contenuto della lista, mostrando ciascun valore dopo 2 secondi di ritardo.

Altri metodi di Thread

- Esaminiamo altri metodi utili della classe Thread

```
public static Thread currentThread()
```

- Restituisce l'oggetto thread corrispondente al thread di esecuzione che l'ha invocato
- Con questo metodo è possibile anche ottenere un riferimento all'oggetto thread corrispondente al thread principale (quello che esegue inizialmente il metodo main)

Altri metodi di Thread

```
public final void join() throws InterruptedException
```

- Il metodo join interagisce con due thread (sia oggetti, sia thread di esecuzione):
  - thread 1: il thread che lo chiama, cioè il flusso di esecuzione che invoca join
  - thread 2: il thread sul quale è chiamato, cioè il thread corrispondente all'oggetto puntato da this

  Esempio:
  - t.join() (dove t è una variabile di tipo Thread)
  - Thread 1: il thread corrente, quello che esegue t.join()
  - Thread 2: t
  - Analogo a pthread_join(t)

- Il metodo join **mette in attesa il thread 1 fino alla terminazione del thread 2**
- Se il thread 2 non è ancora partito, oppure è già terminato, il metodo join ritorna immediatamente

- Pertanto, si tratta di un metodo **bloccante**
- Come tutti i metodi bloccanti, lancia l'eccezione verificata InterruptedException se l'attesa viene interrotta

- Questo metodo svolge un compito analogo alla system call **waitpid** dei sistemi Unix, nonché della funzione **pthread_join** dello standard POSIX thread

Esercizio 2

[3/9/2012, #2] Elencare tutte le sequenze di output possibili per il seguente programma:

```
public class MyThread extends Thread
{
    private int id;
    private Thread other;

    public MyThread(int n, Thread t) {
       id = n;
       other = t;
    }

    public void run() {
       try {
          if (other!=null)
             other.join();
       } catch (InterruptedException e) {
          return;
       }
       System.out.println(id);
    }
}
```

Nel **main**:

```
Thread t1 = new MyThread(1,null);
Thread t2 = new MyThread(2,null);
Thread t3 = new MyThread(3,t1);
Thread t4 = new MyThread(4,t2);
t1.start();
t2.start();
t3.start();
t4.start();
```

Esercizio 2

[3/9/2012, #2] Elencare tutte le sequenze di output possibili per il seguente programma:

```
public class MyThread extends Thread
{
    private int id;
    private Thread other;

    public MyThread(int n, Thread t) {
       id = n;
       other = t;
    }

    public void run() {
       try {
          if (other!=null)
             other.join();
       } catch (InterruptedException e) {
          return;
       }
       System.out.println(id);
    }
}
```

Nel **main**:

```
Thread t1 = new MyThread(1,null);
Thread t2 = new MyThread(2,null);
Thread t3 = new MyThread(3,t1);
Thread t4 = new MyThread(4,t2);
t1.start();
t2.start();
t3.start();
t4.start();
```

Soluzione:

Tutte le permutazioni di 1 2 3 4 in cui 1 precede 3 e 2 precede 4.

Interrompere un thread

- E' spesso utile interrompere le operazioni di un thread
- A tale scopo, ogni thread è dotato di un **flag booleano** chiamato *stato di interruzione*, inizialmente falso
- I metodi bloccanti, come sleep e join, vengono interrotti non appena lo stato di interruzione diventa vero
- Il seguente metodo della classe Thread imposta a *vero* lo stato di interruzione del thread sul quale è chiamato

```
public void interrupt()
```

- Quindi, nonostante il suo nome, interrupt non ha un effetto *diretto* su un thread di esecuzione
- In particolare, se tale thread non sta eseguendo un'operazione bloccante, la chiamata ad interrupt non ha nessun effetto immediato
- Tuttavia, la successiva chiamata bloccante troverà lo stato di interruzione a *vero* ed uscirà immediatamente lanciando l'apposita eccezione

- E' possibile conoscere lo stato di interruzione di un thread chiamando

```
public boolean isInterrupted()
```

- Tale metodo restituisce l'attuale stato di interruzione di questo thread, senza modificarlo

La disciplina delle interruzioni

- Una applicazione dovrebbe sempre essere in grado di terminare tutti i suoi thread su richiesta
- Infatti, in un ambiente interattivo l'utente potrebbe richiedere la chiusura dell'applicazione in qualunque momento
- Per ottenere questo risultato, tutti i thread dovrebbero rispettare la seguente disciplina relativamente alle interruzioni

- Se una chiamata bloccante lancia l'eccezione InterruptedException, il thread dovrebbe interpretarla come una **richiesta di terminazione**, e reagire assecondando la richiesta
- Se un thread non utilizza periodicamente chiamate bloccanti, dovrebbe invocare periodicamente *isInterrupted* e terminare se il risultato è vero

- Avvertenze:
  - Queste regole valgono soprattutto per i thread che hanno una durata potenzialmente illimitata, come quelli basati su un ciclo infinito
  - In questo caso, invece di usare "while (true)" è possibile utilizzare

    ```
    while (!Thread.currentThread().isInterrupted())
    ```
  - Naturalmente, è anche possibile segnalare un'interruzione al thread in altro modo, ad esempio utilizzando lo stato di un oggetto condiviso
  - Nella prossima lezione si tratterà il tema della comunicazione tra thread

Esercizio 3

[26/6/06, traccia B, #5]

Implementare la classe **Interruptor,** il cui compito è quello di interrompere un dato thread dopo un numero fissato di secondi.

Ad esempio, se t è un riferimento ad un oggetto Thread, la linea

```
Interruptor i = new Interruptor(t, 10);
```

crea un nuovo thread di esecuzione che interrompe il thread t dopo 10 secondi.

L'interfaccia Runnable

- Abbiamo visto come creare thread di esecuzione istanziando sottoclassi della classe Thread
- Questo metodo ha lo svantaggio che la sottoclasse di Thread che creiamo non può estendere alcuna altra classe
- In alternativa, è possibile creare un thread di esecuzione tramite una nostra classe che implementi l'interfaccia Runnable, il cui contenuto è il seguente

```
public interface Runnable {
     public void run();
}
```

- Come si vede, l'interfaccia contiene solo un metodo run del tutto analogo a quello della classe Thread
- Tale metodo sarà l'entry point per il nuovo thread di esecuzione
- Per creare il thread, utilizziamo il seguente costruttore della classe Thread, passandogli come argomento un oggetto di una nostra classe che implementa Runnable

```
public Thread(Runnable r)
```

- Naturalmente, per far partire il nuovo thread, è sempre necessario chiamare il metodo start

Thread creati con Runnable

- E' possibile usare lo stesso oggetto Runnable per creare più thread: tutti eseguiranno lo stesso metodo run
- Quando scriviamo il metodo run di un oggetto Runnable, è facile dimenticare che non ci troviamo in una classe che estende Thread
- Quindi, non è possibile scrivere semplicemente

`isInterrupted()` oppure `sleep(1000)`

- ma bisogna scrivere

`Thread.currentThread().isInterrupted()` e `Thread.sleep(1000)`

Supporto ai thread in C++11

- A partire dal C++11, è presente un supporto ai thread nella libreria standard
- Header file “thread” e classe “thread”

Esempio:

```
#include <thread>
#include <iostream>

using namespace std;

void foo(int n, char *args[]) {
   for (int i=0; i<n; i++) {
      cout << i << "\t" << args[i] << endl;
   }
}

int main(int argc, char *argv[]) {
   thread t1(foo, argc, argv);
   thread t2(foo, argc, argv);
   thread t3(foo, argc, argv);
   t1.join();
   t2.join();
   t3.join();
}
```

Notare similitudini e differenze con Java

Esercizio 4: periodicJob

Implementare il metodo statico **periodicJob**, che accetta un Runnable r e un periodo p espresso in millisecondi e fa partire un'esecuzione di r ogni p millisecondi.

Il metodo periodicJob non deve essere bloccante.

Esempio d'uso:

```
Runnable r = new Runnable() {
   public void run() {
        System.outprintln(“Ciao”);
   }
};
periodicJob(r, 2000);
```

Risultato:

il programma stampa a video “Ciao” ogni 2 secondi.

Esercizio 5

[8/9/08, #3]

Si consideri la seguente interfaccia.

```
public interface RunnableWithArg<T> {
  void run(T x);
}
```

Un oggetto *RunnableWithArg* è simile ad un oggetto *Runnable*, tranne per il fatto che il suo metodo *run* accetta un argomento.

Si implementi una classe ***RunOnSet*** che esegue il metodo *run* di un oggetto *RunnableWithArg* su tutti gli oggetti di una data collezione, *contemporaneamente*.

[Esempio d'uso sulla slide successiva]

Esercizio 5

Esempio d'uso:

```
Collection<Integer> s = new HashSet<Integer>();
s.add(3); s.add(13); s.add(88);

RunnableWithArg<Integer> r = new RunnableWithArg<Integer>() {
    public void run(Integer i) {
  System.out.println(i/2);
    }
};
Thread t = new RunOnSet<Integer>(r, s);
t.start();
```

Un possibile output dell'esempio d'uso:

```
1
6
44
```

Tabella riassuntiva

- Riassumiamo qui i metodi della classe Thread esaminati, tutti pubblici

• Thread()	costruttore senza argomenti
• Thread(Runnable r)	costruttore che accetta un *Runnable*
• void start()	crea e avvia il corrispondente thread di esecuzione
• void run()	l'entry point del thread
• void join() throws I.E.	attende la terminazione di questo thread
• void interrupt()	imposta a vero lo stato di interruzione di questo thread
• boolean isInterrupted()	restituisce lo stato di interruzione di questo thread
• **static** Thread currentThread()	restituisce l'oggetto thread del thread di esecuzione corrente
• **static** void sleep(long m) throws I.E.	attende *m* millisecondi