# Programmazione in Java
# Parte II

## Lezione 8

**Dott. Marco Faella**

# 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

- Java è l'unico tra i linguaggi di programmazione maggiormente utilizzati ad offrire un supporto nativo ai thread
- Ad esempio, per supportare i thread il linguaggio C/C++ necessita 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

- In Java, i thread sono supportati dall'omonima 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
- In Java, ad ogni thread di esecuzione è associato un oggetto thread, mentre il viceversa non è sempre vero

- 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

- Per creare un thread di esecuzione seguiamo i 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) chiamare il metodo `start` dell'oggetto creato

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

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

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

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

# 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 chiama 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”

# 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)

```
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
- Il metodo join **mette in attesa il thread 1 fino alla terminazione del thread 2**
- Se il thread 2 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

# 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

# Esercizi (esame 26/6/2006, 27/3/2008)

1) 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.

2) 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.

Viene valutato positivamente l'uso di classi anonime.

# 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

# Tabella riassuntiva

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

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

# Sincronizzazione tra thread

- Se due thread tentano di modificare contemporaneamente lo stesso oggetto, l'interleaving sostanzialmente casuale può far si che l'operazione lasci l'oggetto in uno stato incoerente
- E' necessario garantire che solo un thread alla volta possa modificare tale oggetto
- Questa proprietà prende il nome di *mutua esclusione*
- La soluzione classica al problema prevede l'uso di mutex
- Si consulti un libro sui sistemi operativi per un'introduzione ai mutex

- Un mutex è un semaforo binario che supporta le operazioni base di *lock* e *unlock*
- Java integra i mutex nel linguaggio stesso
  - ad ogni oggetto, indipendentemente dal tipo, è associato un mutex e una corrispodente lista d'attesa
  - la parola chiave ***synchronized*** permette di utilizzare implicitamente tali mutex

- In queste lezioni, talvolta chiameremo “x.mutex” il mutex associato all'oggetto “x”
- Questa è una notazione puramente didattica, che non trova corrispondenza nel linguaggio Java

- Il modificatore synchronized si può applicare ad un metodo, oppure ad un blocco di codice
- Si noti che non si può applicare synchronized ad un campo o altra variabile

# Metodi sincronizzati

- Consideriamo il caso di un metodo a cui sia applicato il modificatore synchronized
- In tal caso, diremo che il metodo è *sincronizzato*

```
public synchronized int f(int n) { ... }
```

- Supponiamo che “x.f(3)” sia una chiamata a tale metodo

- L'effetto del modificatore synchronized è il seguente:
  - Prima di entrare nel metodo “f”, il thread corrente tenta di acquisire il mutex di “x”
    - informalmente, è come se il thread chiamasse x.mutex.lock()
  - Se il mutex è già impegnato, il thread viene messo in attesa che si liberi
  - Quando esce dal metodo “f”, il thread rilascia il mutex di “x”
    - informalmente, è come se il thread chiamasse x.mutex.unlock()

- In altre parole, quando un thread chiama un metodo sincronizzato “f” di un dato oggetto, altri thread che chiamino **qualunque** metodo sincronizzato dello **stesso oggetto** devono aspettare che il primo thread esca dalla chiamata a “f”
- Questo garantisce che solo un thread alla volta possa eseguire i metodi sincronizzati di ciascun oggetto

# Blocchi sincronizzati

- La parola chiave synchronized può anche introdurre un blocco di codice
- In questo caso, parleremo di blocco (di codice) sincronizzato
- Usato in questo modo, synchronized richiede come argomento l'oggetto del quale vogliamo acquisire il mutex
- Ad esempio, il seguente frammento di codice:

```
Integer n = 0;
synchronized (n) {
...
}
```

- corrisponde informalmente a (il codice seguente non è Java, ma è solo esemplificativo):

```
Integer n = 0;
n.mutex.lock();
...
n.mutex.unlock();
```

- Si noti che i mutex acquisiti dai blocchi sincronizzati sono gli **stessi** che sono utilizzati anche dai metodi sincronizzati
- Quindi, se un thread sta eseguendo un blocco che è sincronizzato sull'oggetto “x”, gli altri thread devono aspettare per eseguire eventuali metodi sincronizzati di “x”