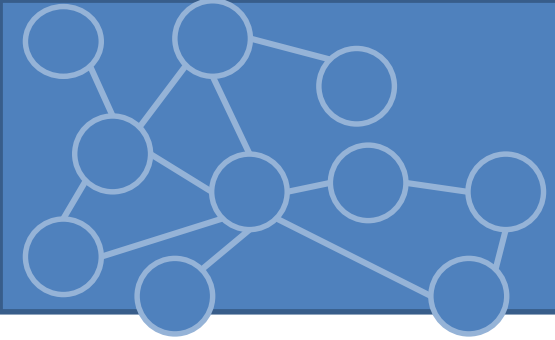
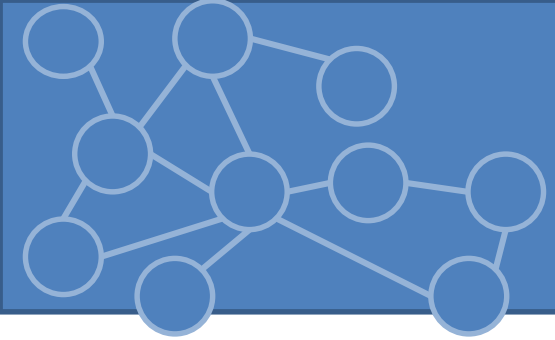


Laboratorio Reti di Calcolatori

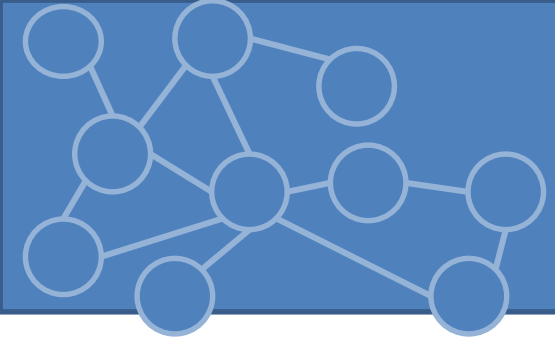
Laurea Triennale in Comunicazione Digitale

Anno Accademico 2012/2013



Thread e processi

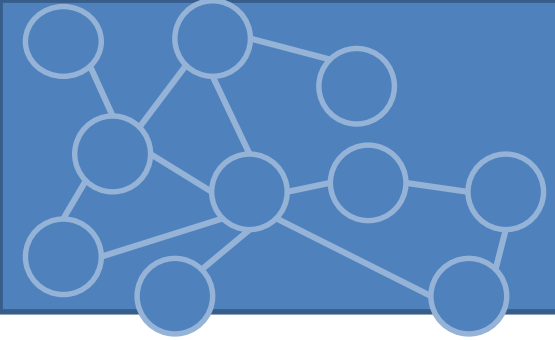
- Da software ci si attende la concorrenza. Java è stato progettato per supportare concorrenza sia a basso livello sia ad alto livello (*java.util.concurrent*)
- Nella programmazione concorrente 2 unità base di esecuzione: processi e thread. **Java è orientato verso i thread**
- Sempre processi e thread attivi anche su single core. Un solo thread in esecuzione in un dato momento ma tempo di processing distribuito tra i processi e thread sfruttando il time slicing
- Con avvento di processori multicore aumentato uso e potenzialità di processi e thread concorrenti



- Processi
  - ambiente di esecuzione che possiede un insieme privato e completo di risorse a run-time. In particolare una spazio di memoria.
  - Comunicazione tra processi resa disponibile da SO che supportano IPC (Inter Process Communication) come socket e pipe. Per comunicazione nello stesso sistema e sistemi differenti
  - JVM eseguita come processo singolo. Si possono creare processi usando l'oggetto ProcessBuilder

```
ProcessBuilder pb = new ProcessBuilder(‘’comando’’,’’arg1’’);
Process p = pb.start();
```

- Threads
  - Processi leggeri: meno risorse richieste
  - Esistono all'interno di un processo e ne condividono le risorse (memoria e file aperti)
  - Ogni Java application ha almeno un thread (main thread) che crea ulteriori threads



In Java ogni thread associato con un'istanza della classe Thread. Posso essere utilizzati per creare applicazioni concorrenti

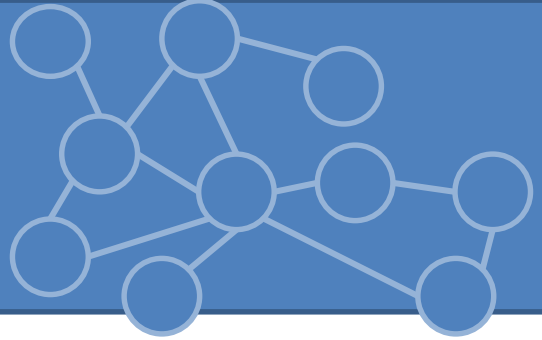
1. Istanzio l'oggetto ogni volte l'applicazione deve iniziare un task asincrono. Creazione e gestione diretta
2. Passo i task ad un esecutor per astrarre la gestione dei thread

Due metodi per creare un'istanza di Thread:

1. Implementare interfaccia ***Runnable*** che prevede il solo metodo ***run()*** che contiene il codice da eseguire nel thread. Oggetto Runnable passato al costruttore di Thread
2. Estendo la classe Thread sovrascrivendo il metodo ***run()***

In entrambi i casi viene invocato il metodo ***Thread.start ()*** per lanciare il nuovo thread

codice



Sleep

Thread.sleep determina la sospensione dell'esecuzione del thread per un periodo specificato

static void sleep(millisecondi)

static void sleep(millisecondi, nanosecondi)

- Non c'è garanzia sulla precisione a causa dei vincoli posti da SO + il periodo di sleep può essere terminato da interrupts
- Solleva ***InterruptedException*** quando altro thread interrompe il thread corrente mentre lo sleep è attivo

Interrupt

Interrupt: segnale al thread che dovrebbe sospendere esecuzione e fare altro. È programmatore che decide come thread risponde a interrupt (di solito termina).

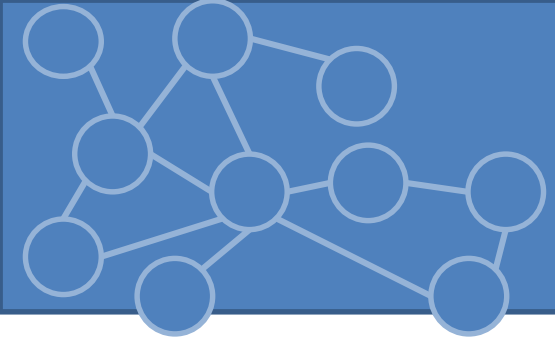
- Thread A invia un interrupt a B invocando il metodo interrupt su oggetto Thread B

Come thread supporta la sua interruzione

1. Se uso frequente di metodi che sollevano InterruptedException, catturo eccezione ed esco dal run
2. Se uso non frequente devo periodicamente invocare **Thread.interrupted()** che restituisce true se è stato ricevuto un interrupt (codice)

Interrupt status flag

Meccanismo di interrupt implementato da un flag interno. Invocare *interrupt()* setta il flag, *interrupted()* resetta il flag mentre ***isInterrupted()*** non cambia lo stato.

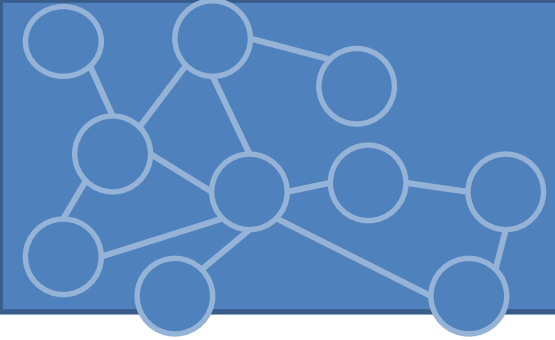


Metodo ***join()*** permette ad un thread di attendere il completamento di un altro thread. Se t è un thread in esecuzione allora

t.join()

Mette in pausa il thread corrente fino a che t termina

- Sensibile all'interrupt



Errori di concorrenza

Thread comunicano condividendo l'accesso a campi o riferimenti ad oggetti.

- Forma di comunicazione efficiente ma determina due tipi di errori

Thread Interference

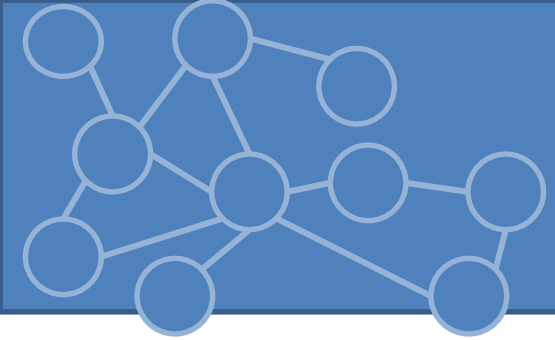
Due operazioni in thread diversi che agiscono sullo stesso dato si sovrappongono. Le operazioni non sono atomiche ma consistono in una sequenza di istruzioni. Le sequenze si sovrappongono

Errori di consistenza nella memoria

Thread diversi hanno una visione inconsistente dello stesso dato.

Per evitare si deve capire la relazione happens-before = le scritture in memoria fatta da un'istruzione devono essere visibili a un'altra specifica istruzione.

Es: sincronizzazione, *start(), join()*



Sincronizzazione

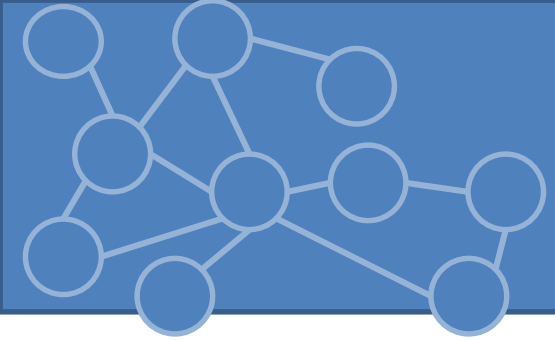
Due costrutti per la sincronizzazione

Metodi sincronizzati

- Aggiungo la parola chiave ***synchronized*** alla dichiarazione del metodo

1. Non è possibile che due chiamate al metodo sincronizzato sull'oggetto si sovrappongano. Altri thread che invocano metodi sincronizzati sullo stesso oggetto vengono sospesi.
2. Quando un metodo sincronizzato termina, determina la relazione happens-before rispetto a chiamate successive di altri metodi sincronizzati

- Costruttori non possono essere sincronizzati

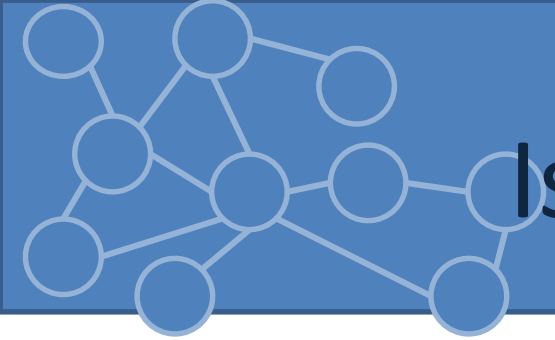


Sincronizzazione si basa su entità interna: *intrinsic lock* o *monitor lock*. Agisce su entrambi gli errori di concorrenza visti.

- Ogni oggetto ha un intrinsic lock. Thread che vuole accesso esclusivo e consistente ai campi dell'oggetto acquisisce il lock prima di accedere e rilascia il lock quando finisce. Finchè il thread possiede il lock nessun altro thread può averlo.

Metodo sincronizzato

- Thread acquisisce il lock per l'oggetto con quel metodo
  - Nel caso di metodo statico acquisisce il lock dell'oggetto Class associato alla classe



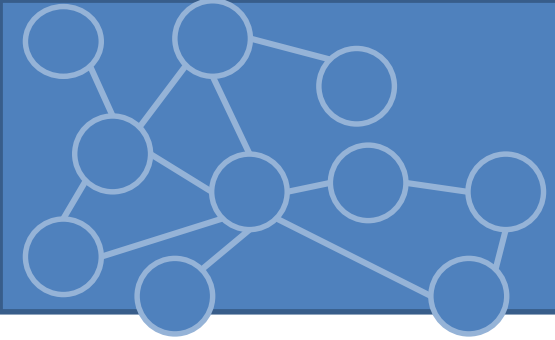
Istruzioni sincronizzate

Nelle istruzioni sincronizzate devo definire l'oggetto che fornisce l'intrinsic lock.

```
public void addName(String name) {
   synchronized(this) {
      lastName = name;
      nameCount++;
   }
   nameList.add(name); => VOGLIO EVITARE INVOCAZIONI SINCRONIZZATE SU METODI DI ALTRI OGGETTI
}
```

Per sincronizzazione più fine

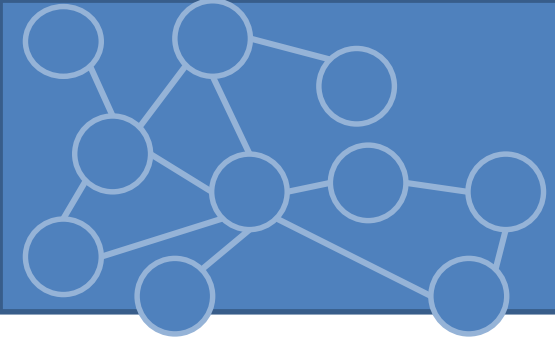
- Thread può acquisire un lock che già possiede = **reentrant synchronization:** codice già sincronizzato invoca metodo che contiene codice sincronizzato e entrambi usano lo stesso lock.



Deadlock: due o più sono per sempre bloccati in attesa che altri rilascino la risorsa (codice)

Starvation: thread non riesce ad avere un accesso regolare alla risorsa condivisa e non procede nel task. La risorsa condivisa è inutilizzabile a causa di thread ‘’voraci’’

Livelock: Thread risponde in base all’azione di un altro thread il quale risponde ad azione del primo. Sono bloccati in maniera diversa rispetto al deadlock



Oggetti immutabili

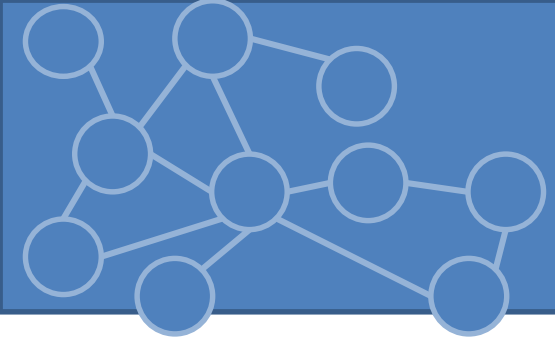
Oggetto immutabile: oggetto il cui stato non può essere modificato dopo la sua costruzione.

- Non soffrono di thread interference o inconsistenza

Regole per rendere oggetto immutabile

1. Non definisco metodi di set
2. Tutti i campi final e private
3. Non permettere ad una sottoclasse di sovrascrivere i metodi
4. Se i campi si riferiscono ad oggetti mutabili, non permettere che oggetti vengano modificati

(codice)



Executors

Finora stretta connessione tra il thread e il task, ma su larga scala meglio separare la creazione e la gestione dei thread dal resto dell'applicazione.

Approccio permesso da ''esecutori''

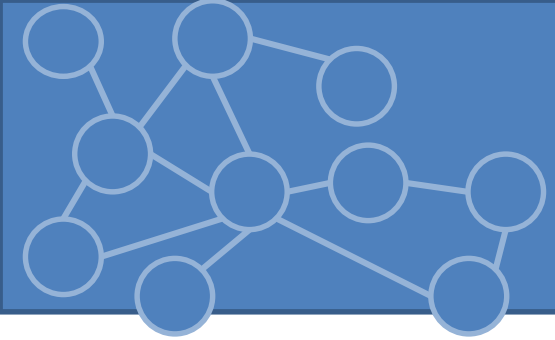
java.util.concurrent definisce tre interfacce per gli esecutori:

- ***Executor***: supporta l'esecuzione di un nuovo task
- ***ExecutorService***: sottointerfaccia di Executor e migliora la gestione del ciclo di vita dei task
- ***ScheduledExecutorService***: sottointerfaccia di ExecutorService che supporta esecuzione futura e periodica di task

Interfaccia Executor

- Solo il metodo ***execute*** che rimpiazza lo start() di un Thread ma meno specifico.

  e.execute(Runnable r)



Thread Pools

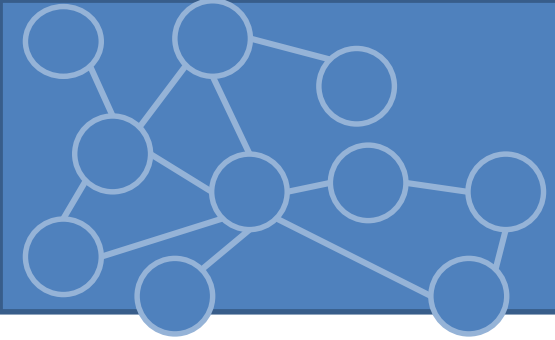
La maggior parte delle implementazioni in java.util.concurrent usano una ‘’riserva’’ (pool) di worker threads che esistono a prescindere dai task Runnable

- Minimizza l’overhead dovuto alla creazione di un thread (gestione della memoria)

Fixed thread pool: numero specificato di thread in esecuzione. Tasks assegnati utilizzando una coda interna.

Creazione di un Executor

- Metodi factory della classe ***java.util.concurrent.Executors***
- Classi ***ThreadPoolExecutor*** o ***ScheduledThreadPoolExecutor***



Esercizi

- Si crei un thread, utilizzando entrambi i metodi precedentemente illustrati, che legga un file e lo visualizzi in *System.out*
- Si crei una classe che riceve da linea di comando una serie di file e li visualizza contemporaneamente su *System.out*
- Si crei una classe che implementa il metodo copieMultiple(String filename, int m) che crea m copie dello stream passato come argomento.
- Si creino due stream di input e si assegni la lettura a due thread diversi. Si attenda nel main che entrambi abbiamo terminato la lettura.
- Si implementi una classe che decomprime un file .zip sfruttando il multithread.
- Nella precedente implementazione si faccia uso di un thread pool
- Si renda la classe SynchronizedRGB immutabile seguendo le 4 regole illustrate nelle slide