

JAVA – Programmazione concorrente

Metodi Avanzati di Programmazione
Laurea Triennale in Informatica
Università degli Studi di Bari Aldo Moro
Docente: Pierpaolo Basile

Programmazione concorrente

- Un calcolatore può eseguire più task contemporaneamente
 - Riprodurre musica
 - Navigare sul web
 - Scrivere un documento
- JAVA mette a disposizione una serie di classi per la programmazione concorrente
 - `java.util.concurrent`

Processi e Thread

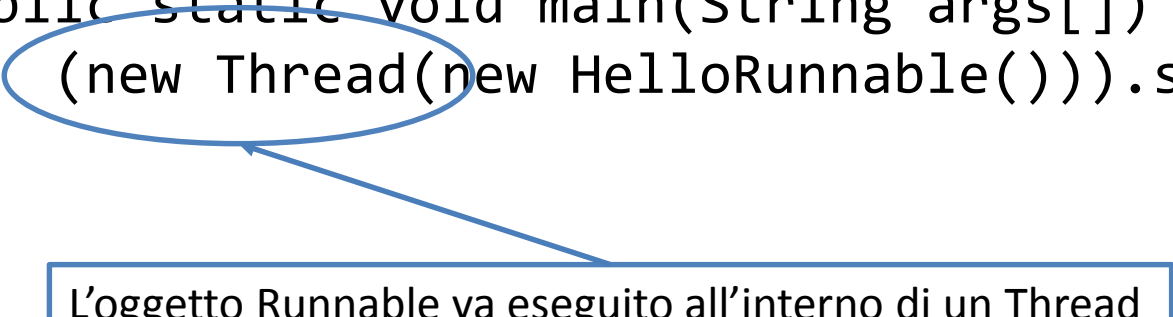
- Anche un calcolatore con una singola CPU esegue più processi contemporaneamente
 - *time slicing*: il tempo della CPU è suddiviso tra più processi
- Processi: in genere rappresentano una singola applicazione
- Thread: sono unità di esecuzione meno complesse dei processi
 - un processo può essere composto da più thread
 - i thread all'interno dello stesso processo condividono le stesse risorse (es. memoria e I/O)

Runnable e Thread

- La classe Thread di JAVA implementa tutte le funzionalità di un singolo thread
- Può essere creata in due modi:
 - Creare una classe che implementa l'interfaccia Runnable: questa interfaccia prevede un singolo metodo run() dove va inserito il codice che deve eseguire il thread
 - Creare una classe che estende la classe Thread: anche la classe Thread ha un metodo run() in quanto implementa Runnable

Runnable

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```



L'oggetto Runnable va eseguito all'interno di un Thread

Thread

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
  
}
```

Runnable e Thread (note)

- In entrambi i casi è necessario invocare il metodo `start()` di `Thread` per far partire il Thread
- La classe `Thread` mette a disposizione dei metodi specifici per la gestione e sincronizzazione dei thread
 - oltre ad avere alcuni metodi statici utili
- L'interfaccia `Runnable` è più generica in quanto svincolata dalla classe `Thread`
 - l'utilizzo di `Runnable` evita di far ereditare la vostra classe da `Thread` (in questo modo la classe è libera di ereditare da qualche altra classe)

Sospendere l'esecuzione

- Thread.sleep sospende l'esecuzione del thread corrente per un certo intervallo di tempo
 - in questo modo si rende disponibile del tempo per l'esecuzione di altri thread
- L'intervallo di tempo può essere espresso in millisecondi o millisecondi+nanosecondi
 - il calcolo del tempo dipende dal sistema operativo è può non essere esatto

Sleep (esempio)

```
public class SleepMessages {  
    public static void main(String args[]) throws InterruptedException {  
        String importantInfo[] = {  
            "Info 1",  
            "Info 2",  
            "Info 3",  
            "Info 4"  
        };  
  
        for (int i = 0; i < importantInfo.length; i++) {  
            //sospendi per 4 secondi (4000 millisecondi)  
            Thread.sleep(4000);  
            //Stampa il messaggio  
            System.out.println(importantInfo[i]);  
        }  
    }  
}
```

sleep può generare un'eccezione se il thread viene interrotto da un altro thread durante lo sleep

Interrompere l'esecuzione

- Un thread può essere interrotto chiamando il metodo `interrupt`
 - ogni thread dovrebbe implementare il suo metodo `interrupt`
 - è buona norma che l'`interrupt` di un thread coincida con la sua terminazione
 - si può catturare l'eccezione `InterruptedException` e interrompere l'esecuzione

Esempio interrupt 1

- Riprendendo l'esempio precedente
 - catturiamo l'eccezione durante lo sleep

```
for (int i = 0; i < importantInfo.length; i++) {  
    // Pause for 4 seconds  
    try {  
        Thread.sleep(4000);  
    } catch (InterruptedException e) {  
        // We've been interrupted: no more messages.  
        return;  
    }  
    // Print a message  
    System.out.println(importantInfo[i]);  
}
```

se interrotto esco dal metodo run

Esempio interrupt 2

- Ma se non ci sono metodi che generano InterruptedException?
 - controlliamo periodicamente se qualche altro thread abbia invocato l'interruzione

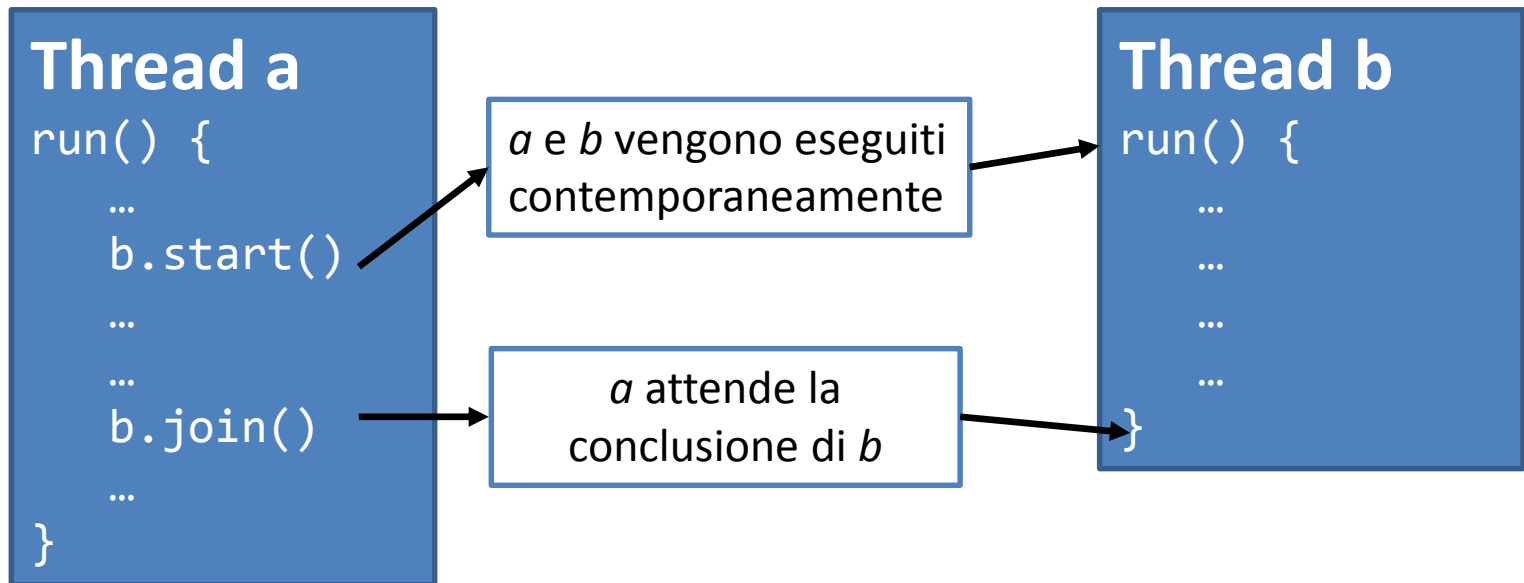
```
for (int i = 0; i < inputs.length; i++) {  
    //do something  
    if (Thread.interrupted()) {  
        // We've been interrupted  
        return;  
    }  
}
```

restituisce true in caso di
interruzione

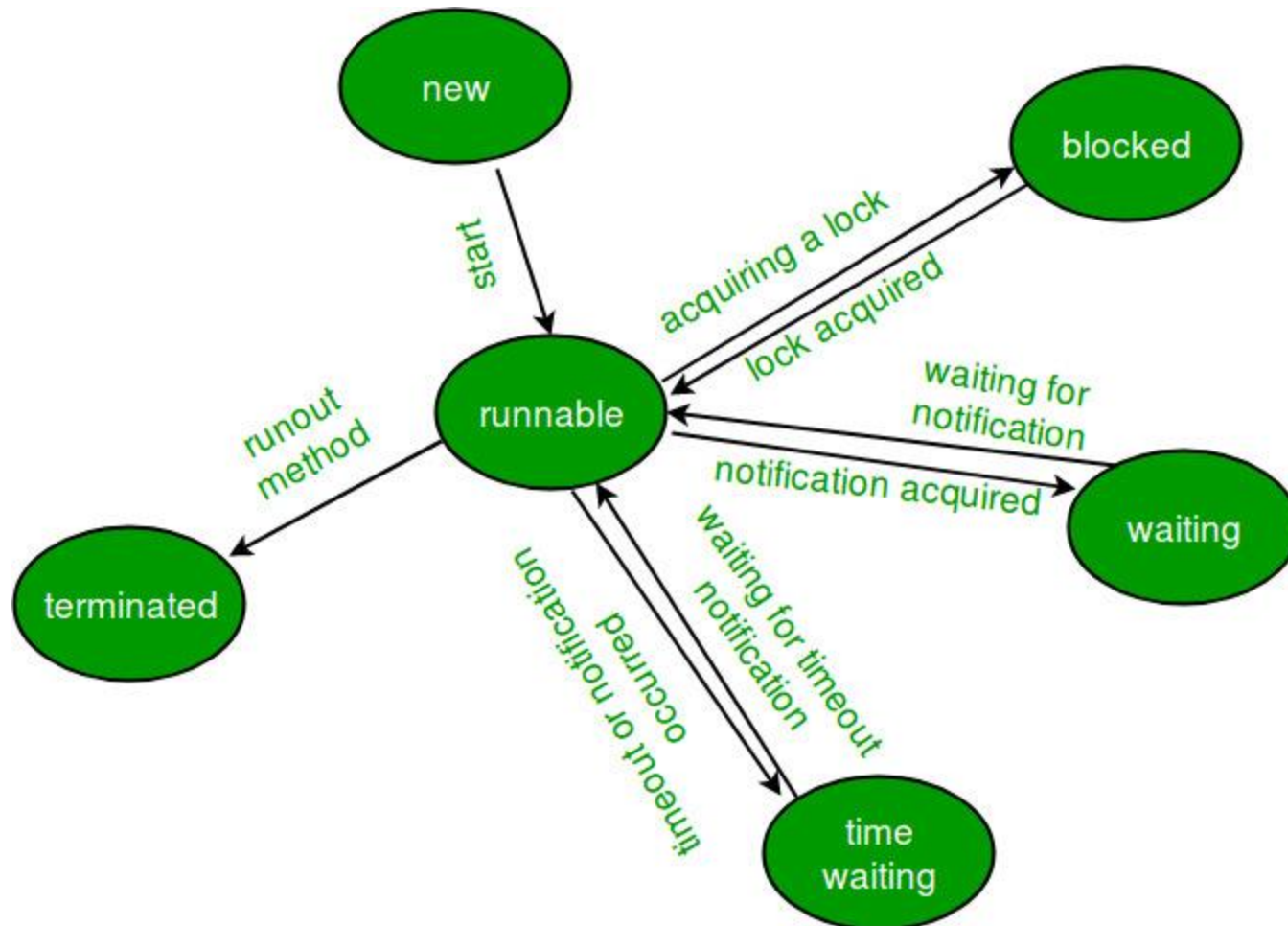
Il metodo join

- La classe Thread ha un metodo *join*
 - la chiamata del metodo *join* mette in pausa il thread corrente fino a quando il thread sul quale si è chiamato *join* non termina
 - Il metodo *join* come *sleep* permette di specificare un tempo massimo di attesa
 - Anche il metodo join può essere interrotto da un `InterruptedException`

Il metodo join



Ciclo di vita di un Thread



Esempio sull'uso dei Thread...

- Creare due thread
 - Uno stampa 100 numeri pari
 - Uno stampa 100 numeri dispari
 - Quello che stampa i numeri pari va più veloce di quello che stampa i numeri dispari
- Creare una classe main che esegue i due thread
 - mentre il thread pari è in esecuzione deve stampare un messaggio ogni secondo
 - al termine del thread pari deve aspettare la fine del thread dispari

...Esempio sull'uso dei Thread...

```
public class Pari implements Runnable {  
    public void run() {  
        int i = 0;  
        for (int k=0;k<100;k++) {  
            try {  
                System.out.println(i);  
                i += 2;  
                Thread.sleep(50);  
            } catch (InterruptedException ex) {  
                return;  
            }  
        }  
    }  
}
```

...Esempio sull'uso dei Thread...

```
public class Dispari implements Runnable {
    public void run() {
        int i = 1;
        for (int k=0;k<100;k++) {
            try {
                System.out.println(i);
                i += 2;
                Thread.sleep(200);
            } catch (InterruptedException ex) {
                return;
            }
        }
    }
}
```

...Esempio sull'uso dei Thread

```
public class ThreadEs1 {  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread pari = new Thread(new Pari(), "Pari");  
        Thread dispari = new Thread(new Dispari(), "Dispari");  
        pari.start();  
        dispari.start();  
        while (pari.isAlive()) {  
            Thread.sleep(1000);  
            System.out.println("Attendo...");  
        }  
        System.out.println(pari.getName()+" è terminato, attendo che  
"+dispari.getName()+" termini...");  
        dispari.join();  
    }  
}
```

Sincronizzazione

- L'unico modo che hanno i Thread per comunicare e accedere alle stesse risorse
 - è un meccanismo efficiente
 - comporta due problematiche: thread interference and memory consistency errors
 - la sincronizzazione può risolvere queste problematiche
 - ma introduce problemi di thread contention quando più thread vogliono utilizzare le stesse risorse

Thread Interference

- Avviene quando due operazioni su due thread differenti agiscono sullo stesso dato (interleave)
- Siccome le operazioni non sono atomiche il risultato che si ottiene sul dato è difficilmente prevedibile

Thread Interference

```
class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
}
```

l'operazione c++ (c--) non è atomica:

1. Recupera il valore di c
2. Incrementa (decrementa) il valore recuperato
3. Assegna a c il nuovo valore

Thread Interference

- Supponiamo che il thread A chiami increment e il thread B chiami decrement
 1. Thread A: Retrieve c
 2. Thread B: Retrieve c
 3. Thread A: Increment retrieved value; result is 1
 4. Thread B: Decrement retrieved value; result is -1
 5. Thread A: Store result in c; c is now 1
 6. Thread B: Store result in c; c is now -1
- c sarà uguale a -1 e l'operazione di A viene persa

Memory Consistency Errors

- Si verifica quando due thread hanno una visione inconsistente dello stesso dato
- Supponiamo che la variabile *int counter = 0* sia accessibile a due Thread A e B
- A incrementa counter: *counter++*
- B stampa counter: *System.out.println(counter)*
- Se A e B sono eseguiti contemporaneamente può accadere che B stampi 0 se l'incremento non è ancora terminato

La sincronizzazione dei metodi

- La sincronizzazione dei metodi permette di ovviare sia al *thread interference* che al *memory inconsistency*
- Un metodo sincronizzato può essere chiamato da un solo thread per volta
 - gli altri thread restano in attesa
- Ciò garantisce che tutto quello che viene fatto in un metodo viene portato a compimento prima che un altro thread chiami lo stesso metodo

Sincronizzazione metodi (esempio)

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```

Sincronizzazione istruzioni

- E' possibile richiedere la sincronizzazione su porzioni di istruzioni
 - questo permette di sincronizzare solo porzioni di codice e non un intero metodo
 - quando un blocco di codice è sincronizzato solo un thread per volta può accedere
- E' sensato sincronizzare solo le porzioni di codice che lo richiedono in questo modo si velocizza l'esecuzione

Sincronizzazione istruzioni (esempio 1)

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

In questo caso sincronizziamo solo la modifica di lastName e di nameCount. L'istruzione add non richiede sincronizzazione

Stiamo inserendo un Lock sull'intero oggetto

Sincronizzazione istruzioni (esempio 2)

```
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
    public void inc2() {  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```

Non blocchiamo tutto l'oggetto (this) ma creiamo due oggetti «fittizi» sui quali possiamo richiedere in contemporanea un Lock, in questo modo inc1 e inc2 possono essere chiamati contemporaneamente da due thread differenti

Programmazione concorrente (problemi)

- **Deadlock:** due o più thread sono bloccati in modo indefinito perché ognuno attende la fine dell'altro
- **Starvation:** un thread non riesce ad accedere ad alcune risorse perché sono utilizzate avidamente da altri thread
- **Livelock:** un thread A in genere agisce in risposta di un altro thread B, se B agisce in risposta di un altro thread C allora i thread proseguiranno in maniera discontinua

THE END

