

PROGRAMMAZIONE MULTITHREAD IN JAVA - PRIMA PARTE -

Alessandro Ricci
a.ricci@unibo.it

SOMMARIO

- Introduzione alla programmazione multithread
 - thread e multithreading
 - richiami (da Sistemi Operativi)
 - programmazione multithread e paradigmi di programmazione
 - software and the concurrency revolution
- Programmazione multithread in Java
 - API di base: classe Thread, interfaccia Runnable
 - Esempi: parallelismo & multi-core programming
 - Terminazione di un thread
- Multithreading e GUI
 - come realizzare interfacce grafiche reattive

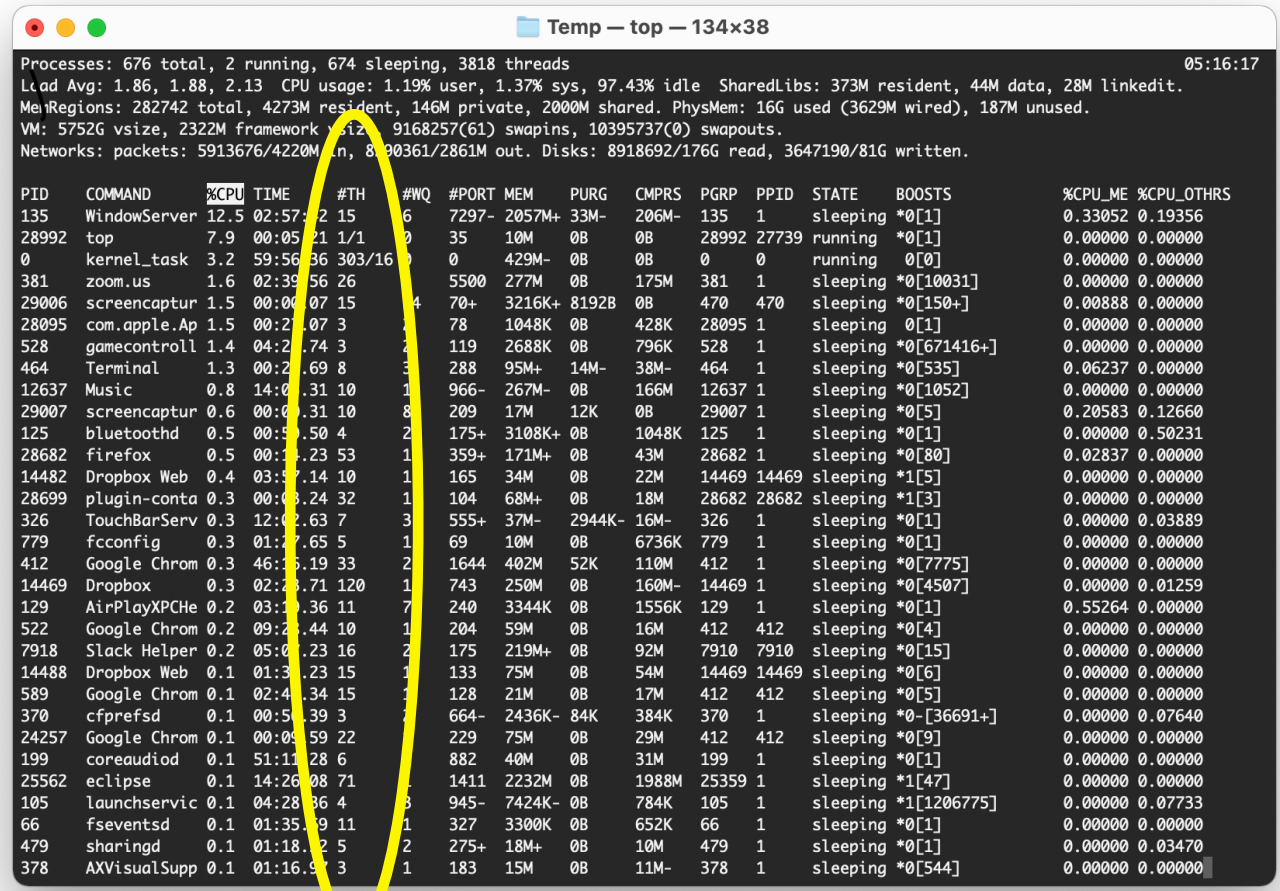
INTRODUZIONE ALLA PROGRAMMAZIONE MULTITHREAD

PROGRAMMAZIONE MULTITHREAD

- Oggigiorno la maggior parte delle applicazioni e sistemi software, sia standalone, sia di rete, è **multithread**
 - un'applicazione è realizzata come un processo con più thread, che rappresentano attività di tipo diverso in esecuzione *concorrentemente*
 - ovvero: la loro esecuzione si sovrappone nel tempo
- Esempi:
 - Web Browser: un thread per visualizzare immagini / testo, uno per recuperare dati via connessioni di rete
 - Word Processor: un thread per visualizzare testo e grafica, uno per raccogliere input (da tastiera..) dell'utente, un altro ancora per controllare errori grammaticali del documento
 - Web Server: un thread per ogni client che sta interagendo con il sito
 - Video-giochi- ...

PERVASIVITA'

- Con il programma top è possibile osservare quanti thread sono creati nell'ambito di un processo UNIX



Processes: 676 total, 2 running, 674 sleeping, 3818 threads
Load Avg: 1.86, 1.88, 2.13 CPU usage: 1.19% user, 1.37% sys, 97.43% idle SharedLibs: 373M resident, 44M data, 28M linkedit.
MemRegions: 282742 total, 4273M resident, 146M private, 2000M shared. PhysMem: 16G used (3629M wired), 187M unused.
VM: 5752G vsize, 2322M framework vsize, 9168257(61) swpins, 10395737(0) swapouts.
Networks: packets: 5913676/4220M in, 890361/2861M out. Disks: 8918692/176G read, 3647190/81G written.

PID	COMMAND	%CPU	TIME	#TH	#WO	#PORT	MEM	PURG	CMRPS	PGRP	PPID	STATE	BOOSTS	%CPU_ME	%CPU_OTHRS
135	WindowServer	12.5	02:57:42	15	6	7297-	2057M+	33M-	206M-	135	1	sleeping	*0[1]	0.33052	0.19356
28992	top	7.9	00:05:21	1/1	0	35	10M	0B	0B	28992	27739	running	*0[1]	0.00000	0.00000
0	kernel_task	3.2	59:56:36	303/16	0	0	429M-	0B	0B	0	0	running	0[0]	0.00000	0.00000
381	zoom.us	1.6	02:39:56	26	0	5500	277M	0B	175M	381	1	sleeping	*0[10031]	0.00000	0.00000
29006	screen captur	1.5	00:00:07	15	4	70+	3216K+	8192B	0B	470	470	sleeping	*0[150+]	0.00888	0.00000
28095	com.apple.Ap	1.5	00:21:07	3	0	78	1048K	0B	428K	28095	1	sleeping	0[1]	0.00000	0.00000
528	gamecontroll	1.4	04:21:74	3	0	119	2688K	0B	796K	528	1	sleeping	*0[671416+]	0.00000	0.00000
464	Terminal	1.3	00:21:69	8	3	288	95M+	14M-	38M-	464	1	sleeping	*0[535]	0.06237	0.00000
12637	Music	0.8	14:00:31	10	1	966-	267M-	0B	166M	12637	1	sleeping	*0[1052]	0.00000	0.00000
29007	screen captur	0.6	00:00:31	10	8	209	17M	12K	0B	29007	1	sleeping	*0[5]	0.20583	0.12660
125	bluetoothd	0.5	00:50:50	4	2	175+	3108K+	0B	1048K	125	1	sleeping	*0[1]	0.00000	0.50231
28682	firefox	0.5	00:11:23	53	1	359+	171M+	0B	43M	28682	1	sleeping	*0[80]	0.02837	0.00000
14482	Dropbox Web	0.4	03:57:14	10	1	165	34M	0B	22M	14469	14469	sleeping	*1[5]	0.00000	0.00000
28699	plugin-conta	0.3	00:00:24	32	1	104	68M+	0B	18M	28682	28682	sleeping	*1[3]	0.00000	0.00000
326	TouchBarServ	0.3	12:02:63	7	3	555+	37M-	2944K-	16M-	326	1	sleeping	*0[1]	0.00000	0.03889
779	fcconfig	0.3	01:27:65	5	1	69	10M	0B	6736K	779	1	sleeping	*0[1]	0.00000	0.00000
412	Google Chrom	0.3	46:15:19	33	2	1644	402M	52K	110M	412	1	sleeping	*0[7775]	0.00000	0.00000
14469	Dropbox	0.3	02:20:71	120	1	743	250M	0B	160M-	14469	1	sleeping	*0[4507]	0.00000	0.01259
129	AirPlayXPCHe	0.2	03:10:36	11	7	240	3344K	0B	1556K	129	1	sleeping	*0[1]	0.55264	0.00000
522	Google Chrom	0.2	09:20:44	10	1	204	59M	0B	16M	412	412	sleeping	*0[4]	0.00000	0.00000
7918	Slack Helper	0.2	05:00:23	16	2	175	219M+	0B	92M	7910	7910	sleeping	*0[15]	0.00000	0.00000
14488	Dropbox Web	0.1	01:30:23	15	1	133	75M	0B	54M	14469	14469	sleeping	*0[6]	0.00000	0.00000
589	Google Chrom	0.1	02:40:34	15	3	128	21M	0B	17M	412	412	sleeping	*0[5]	0.00000	0.00000
370	cfprefsd	0.1	00:50:39	3	0	664-	2436K-	84K	384K	370	1	sleeping	*0-[36691+]	0.00000	0.07640
24257	Google Chrom	0.1	00:00:59	22	0	229	75M	0B	29M	412	412	sleeping	*0[9]	0.00000	0.00000
199	coreaudiod	0.1	51:11:28	6	0	882	40M	0B	31M	199	1	sleeping	*0[1]	0.00000	0.00000
25562	eclipse	0.1	14:26:08	71	1	1411	2232M	0B	1988M	25359	1	sleeping	*1[47]	0.00000	0.00000
105	launchservic	0.1	04:28:36	4	0	945-	7424K-	0B	784K	105	1	sleeping	*1[1206775]	0.00000	0.07733
66	fseventsd	0.1	01:35:09	11	1	327	3300K	0B	652K	66	1	sleeping	*0[1]	0.00000	0.00000
479	sharingd	0.1	01:18:12	5	2	275+	18M+	0B	10M	479	1	sleeping	*0[1]	0.00000	0.03470
378	AXVisualSupp	0.1	01:16:57	3	1	183	15M	0B	11M-	378	1	sleeping	*0[544]	0.00000	0.00000

- Come è possibile verificare, la maggior parte dei processi / applicazioni è multithreaded

CONCORRENZA VS. PARALLELISMO

- **Esecuzione Concorrente**

- l'esecuzione di due attività si sovrappone nel tempo
 - una inizia prima che l'altra sia terminata
- non implica necessariamente l'esecuzione su processori (fisici o logici) distinti
 - può essere lo stesso processore (=> scheduling)

- **Esecuzione Parallela**

- l'esecuzione si sovrappone nel tempo, su processori distinti

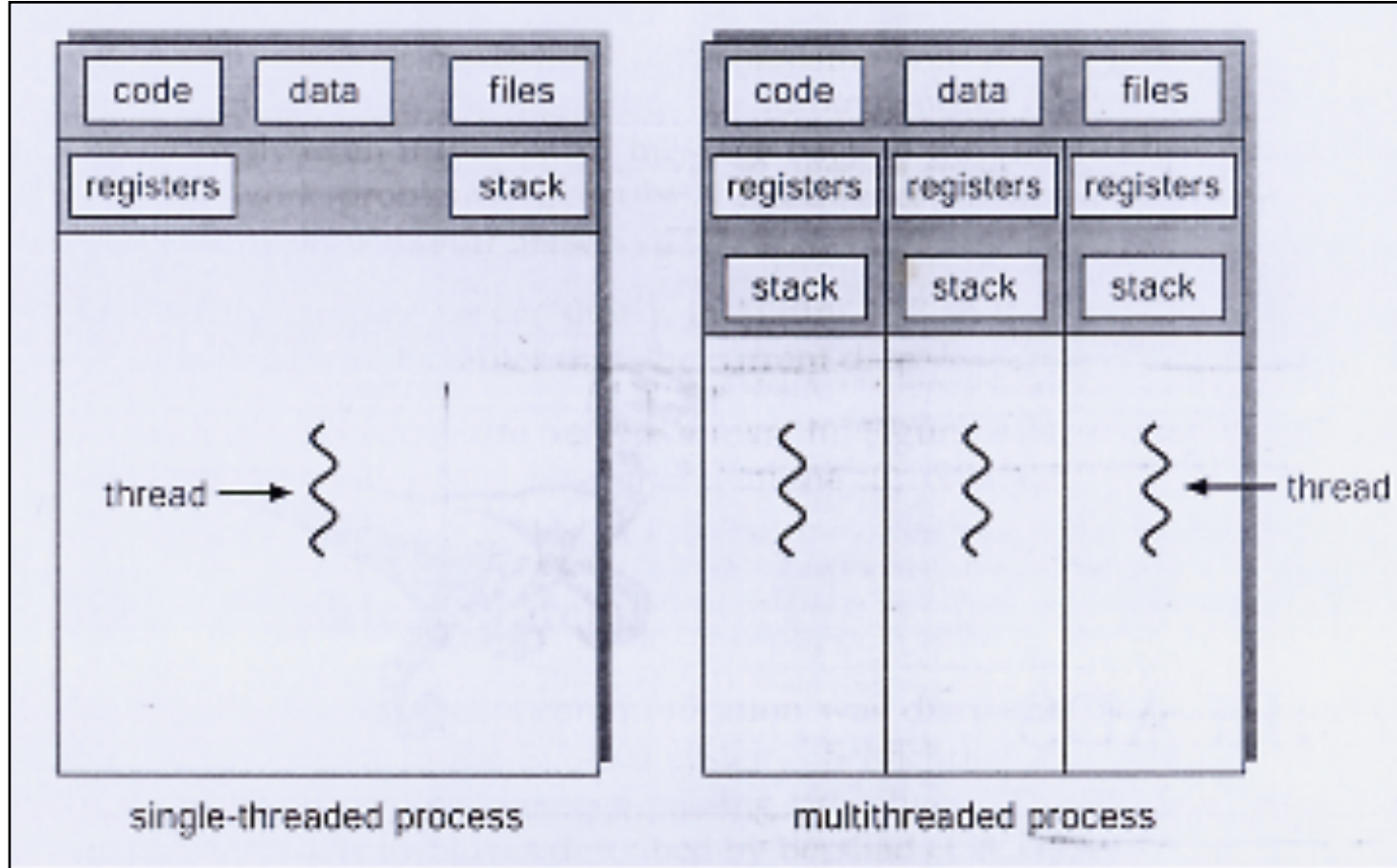
- **Esecuzione Distribuita**

- l'esecuzione avviene su processori che non condividono memoria e comunicano via rete

THREAD NEI S.O.: RICHIAMI

- Nei SO che supportano i thread, un thread diviene l'entità di base con cui si rappresenta un'attività in esecuzione su una CPU, un flusso di controllo
- Un thread nei SO moderni è caratterizzato da:
 - un thread ID
 - program counter
 - set di registri
 - un proprio stack
- In un medesimo processo possono essere creati e mandati in esecuzione più thread:
 - tutti i thread di un processo ne condividono il codice, dati e tipicamente altre risorse del sistema operativo (es: file aperti, segnali,..).
 - quindi a differenza di un processo (pesante) con un solo flusso di esecuzione, processi con più thread possono eseguire più attività simultaneamente.
- Il *context switch* fra thread è molto più leggero di un context switch a livello di processi

THREAD NEI S.O.: RICHIAMI



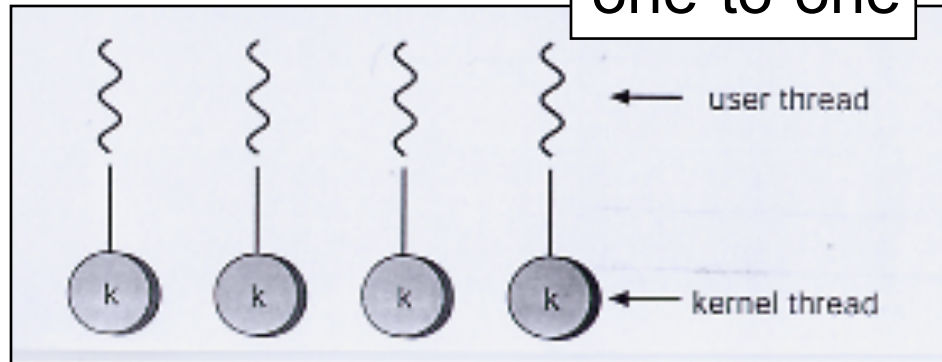
- Si parla di **multithreaded programming** per indicare la programmazione di sistemi mediante l'uso di thread.

KERNEL THREADS E USER THREADS

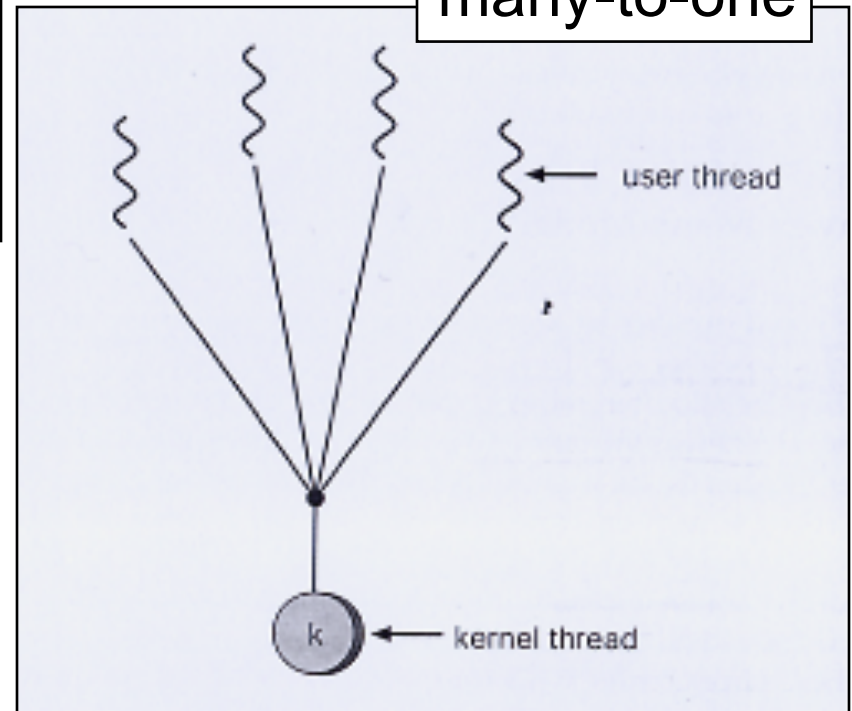
- Il supporto multithread può essere a due livelli
 - **user thread**
 - thread forniti allo user level, "simulato" mediante apposite librerie, senza il supporto diretto del kernel
 - **kernel thread**
 - thread forniti direttamente dal kernel del SO
 - è il caso di tutti i sistemi operativi moderni
- Vari approcci sono utilizzati al fine di mappare thread allo user level in thread al kernel level. I principali sono:
 - **Many-to-one**
 - più user thread vengono mappati su un kernel-thread. Gestione efficiente (user level), ma tutti i thread sono bloccati se si esegue una chiamata di sistema
 - **One-to-One**
 - uno user thread è mappato esattamente su un kernel thread.
 - **Many-to-Many**
 - più user thread vengono mappati su un insieme più piccolo o uguale di kernel thread.

KERNEL THREADS E USER THREADS

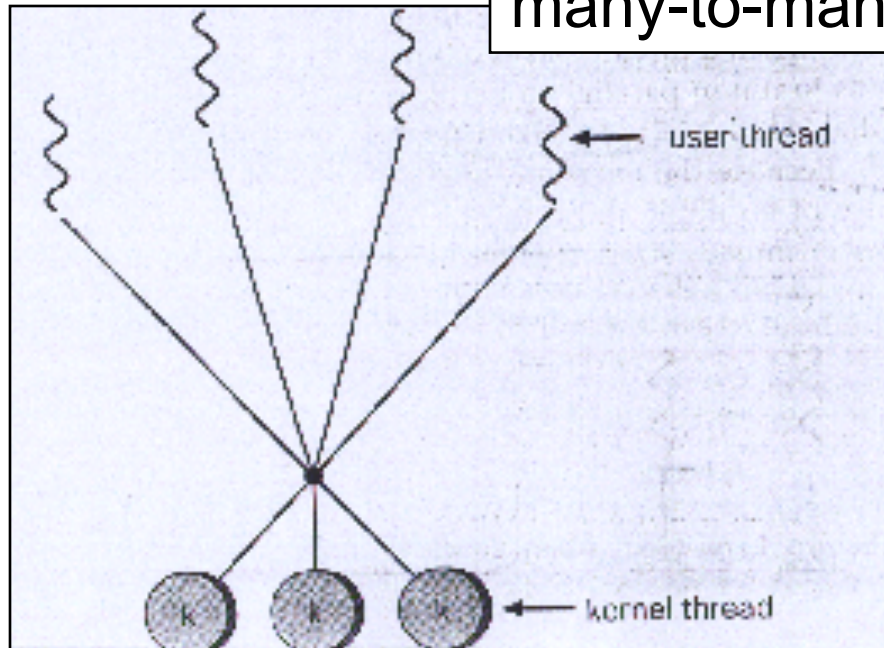
one-to-one



many-to-one



many-to-many



BENEFICI DEL MULTITHREADING

- I benefici dall'utilizzo di thread vengono in genere classificati in quattro categorie principali:
 - **reattività (responsiveness)**
 - la possibilità di creare thread dedicati all'interazione con l'utente, concorrentemente all'esecuzione di operazioni a lungo termine, è fondamentale per creare interfacce utente (UI) adeguate
 - **condivisione di risorse (resource sharing)**
 - thread di uno stesso processo condividono dati e codice, che non devono quindi essere replicati
 - **performance (economy)**
 - la creazione e lo switch di contesto a livello di thread richiede molte meno risorse temporali e spaziali che non la creazione e switch fra processi.
 - **sfruttamento di architetture multiprocessore/multicore**
 - i benefici dell'uso di thread multipli si sentono ancor più su architetture multiprocessore, dove thread distinti possono essere messi in esecuzione concorrente su CPU diverse, aumentando notevolmente la concorrenza.

IMPORTANZA DELLA PROGRAMMAZIONE CONCORRENTE OGGI

- ***Think concurrent!***
 - programmazione concorrente non solo come insieme di meccanismi per aumentare le performance e/o reattività delle applicazioni..
 - ma come strumento concettuale e insieme di principi utili nella progettazione di applicazioni, sistemi
- Impatto a livello di paradigma
 - **problem solving**
 - come affrontare e pensare alla soluzione di problemi
 - **design e sviluppo di programmi d sistemi**
 - modularità, incapsulamento, riusabilità, estendibilità...
- “Software & concurrency revolution” by H. Sutter and J. Larus (2005) (bibliografia)

TERMINOLOGIA

- **Programmazione *parallela* (parallel programming)**
 - l'esecuzione di programmi (o parti di programma) si sovrappone nel tempo, andando in esecuzione su processori fisici separati
- **Programmazione *concorrente* (concurrent programming)**
 - l'esecuzione di programmi (o parti di programma) si sovrappone nel tempo, senza necessariamente andare in esecuzione su processori fisici separati
- **Programmazione *asincrona* (asynchronous programming)**
 - esecuzione di computazioni in modo asincrono, non bloccante
- **Programmazione *distribuita* (distributed programming)**
 - quando i processori sono distribuiti in rete e non c'è condivisione fisica della memoria
- **Programmazione *multithread***
 - quando le parti in esecuzione concorrente sono rappresentate da thread

DESIGN TASK-ORIENTED

- Organizzazione *task-oriented* dei programmi
 - un **task** rappresenta a livello logico un compito ben definito che può essere svolto in modo parzialmente o totalmente concorrente ad altri compiti
 - nel caso di prog. multithreaded => eseguito da thread
- Individuazione e gestione delle dipendenze fra i task
 - esempi:
 - risorse condivise e utilizzate in compiti diversi, informazioni che servono per svolgere un certo task e prodotte da un altro task
 - dipendenze temporali: un certo task deve essere svolto dopo che altri task sono stati completati...
 - nel caso di prog. multithreaded => porta all'uso di meccanismi per la sincronizzazione, coordinazione fra thread
- Decomposizione di task in sotto-task
 - division of labor

PROGRAMMAZIONE CONCORRENTE E OOP

- Visione a livello di paradigma
 - **come integrare programmazione concorrente e OOP ?**
 - storicamente gli oggetti in OOP erano stati pensati per esser logicamente autonomi, concorrenti...
 - con comunicazione basata su scambio di messaggi
 - <http://worrydream.com/EarlyHistoryOfSmalltalk/>
 - come integrare programmazione multithread e OOP?
- **OOP + Multithreading**
 - linguaggi OOP puri + librerie per multithreading
 - esempi: linguaggi C/C++ e libreria PThread su sistemi POSIX, oppure libreria Win32 per sistemi Windows
 - linguaggi OOP estesi con astraz. di prima classe
 - modelli ad **attori**, oggetti concorrenti, agenti...
 - ibrido: linguaggio con meccanismi abilitanti + supporto a livello di libreria
 - Java, Objective C, C# (e CLR lang)

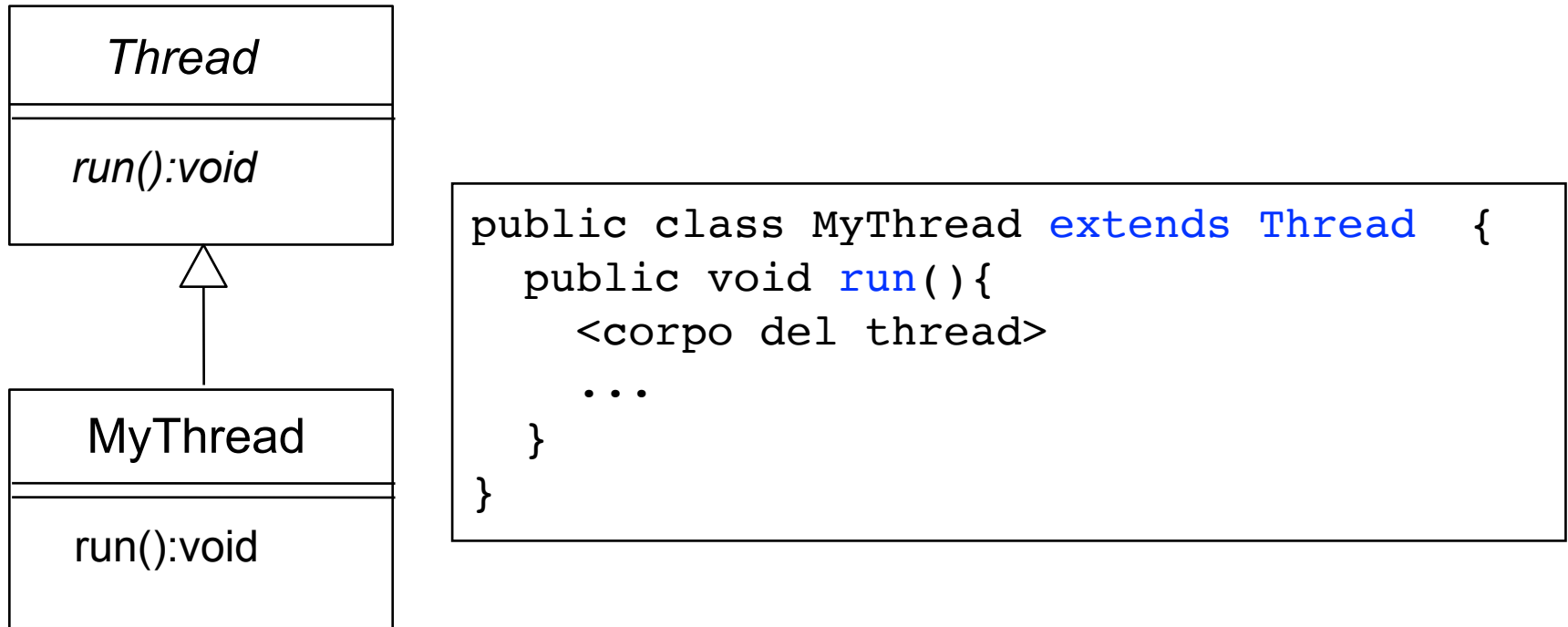
MULTITHREAD PROGRAMMING IN JAVA

PROGRAMMAZIONE MULTITHREAD IN JAVA

- Java è uno dei pochi linguaggi che fornisce supporto per i thread direttamente a livello di linguaggio, cercando di modellare tale nozione in termini di oggetto (classe).
 - la JVM si occupa quindi della creazione e gestione dei thread: come vengono mappati sui kernel thread dipende dal sistema (tipicamente one-to-one)
 - supporto esteso nella versione JDK5.0 con l'introduzione di una nuova libreria (**java.util.concurrent**)
- Un thread è rappresentato dalla classe astratta **Thread**, caratterizzata dal metodo astratto **run**, che definisce il comportamento (attività) del thread.
 - un thread concreto si definisce estendendo la classe Thread, ed specificando il comportamento del metodo run.
 - a tempo di esecuzione, un thread viene creato come un normale oggetto Java, e mandato in esecuzione invocando il metodo **start**.

LA CLASSE Thread

- la classe Thread è fornita direttamente nel package java.lang.



- Il codice eseguito dallo thread è specificato nel metodo **run**
 - il codice effettivo eseguito dipende dall'implementazione specifica descritta nel metodo run della classe derivata

ESEMPIO: UN CLOCK

- Thread Clock che visualizza in standard output la data e l'ora completi, ogni step millisecondi, con step specificato in fase di costruzione

```
public class Clock extends Thread {
    private int step;

    public Clock(int step){
        this.step=step;
    }
    public void run(){
        while (true) {
            System.out.println(new Date());
            try {
                sleep(step);
            } catch (Exception ex){
            }
        }
    }
}
```

LANCIO (“SPAWNING”) DI UN THREAD

- Esecuzione del metodo **start** sull'oggetto thread
 - viene creato un nuovo flusso di controllo che manda in esecuzione il metodo run
- Esempio dell'orologio

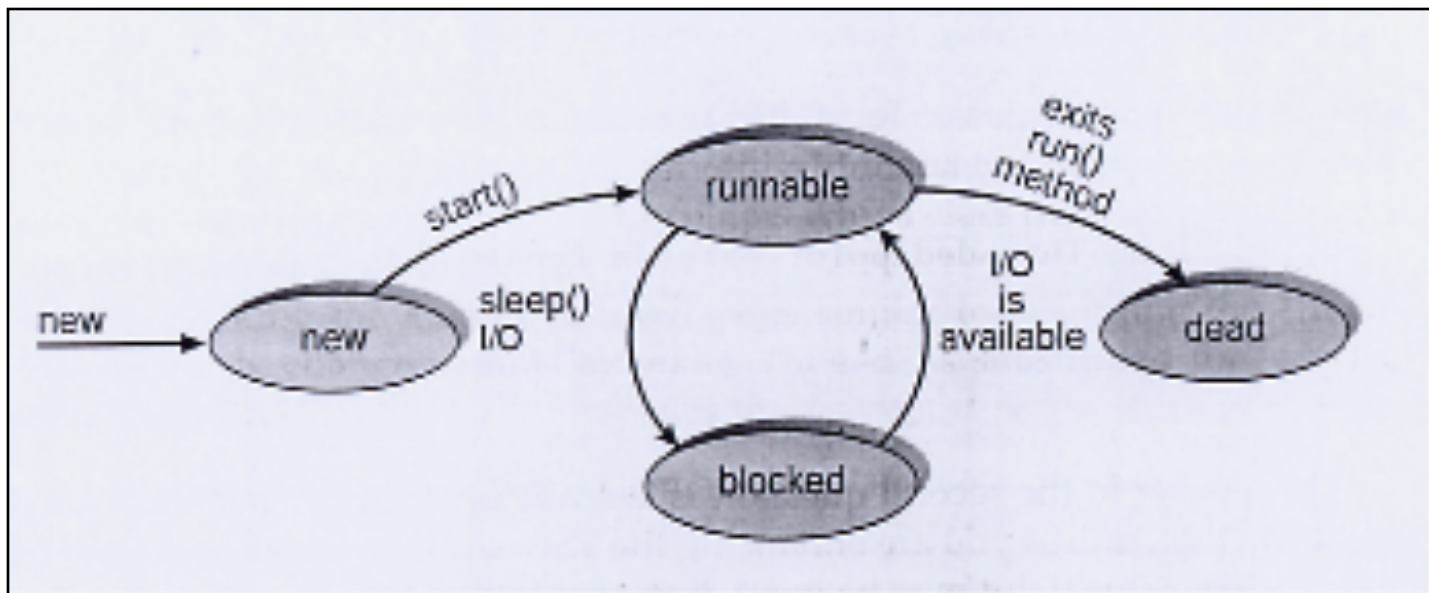
```
package oop.concur;
import java.io.*;
public class TestClock {
    static public void main(String[] args) throws Exception {
        Clock clock = new Clock(1000);
        clock.start();
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        String input = null;
        do {
            input = reader.readLine();
            System.out.println("eco: " + input);
        } while (!input.equals("exit"));
        System.exit(0);
    }
}
```

INTERFACCIA DELLA CLASSE Thread

- Costruttori/metodi significativi della classe Thread sono:
 - **Thread**(String name)
 - costruisce il thread di nome name
 - void Thread.**sleep**(long ms)
 - metodo statico per addormentare il thread corrente di ms millisecondi
 - ~~void destroy()~~
 - distrugge il thread
 - ~~void setPriority~~(int priority)
 - cambia la priorità di esecuzione del thread
 - String getName()
 - ottiene il nome del thread
 - boolean isAlive()
 - verifica se il thread è 'vivo'
 - void **interrupt**()
 - interrompe l'attesa del thread (nel caso fosse in sleep o wait)
 - Thread Thread.**currentThread**()
 - metodo statico per recuperare il riferimento al thread corrente
- Metodi deprecati
 - ~~stop, suspend, resume~~

STATI DI UN THREAD

- Un thread in Java può trovarsi in uno dei seguenti stati:
 - **NEW**: appena creato (con new)
 - **RUNNABLE**: elegibile di essere eseguito dalla JVM oppure direttamente in esecuzione. L'invocazione del metodo `start()` alloca memoria per il nuovo thread nella JVM, quindi viene invocato il metodo `run()`, che provoca il cambiamento dello stato del thread da NEW a RUNNABLE.
 - **BLOCKED**: stato in cui si trova il thread se esegue una operazione bloccante o sospensiva, come una operazione di I/O, oppure operazioni quali `sleep()`
 - **DEAD**: stato in cui si trova il thread quando termina il corpo del metodo `run()`

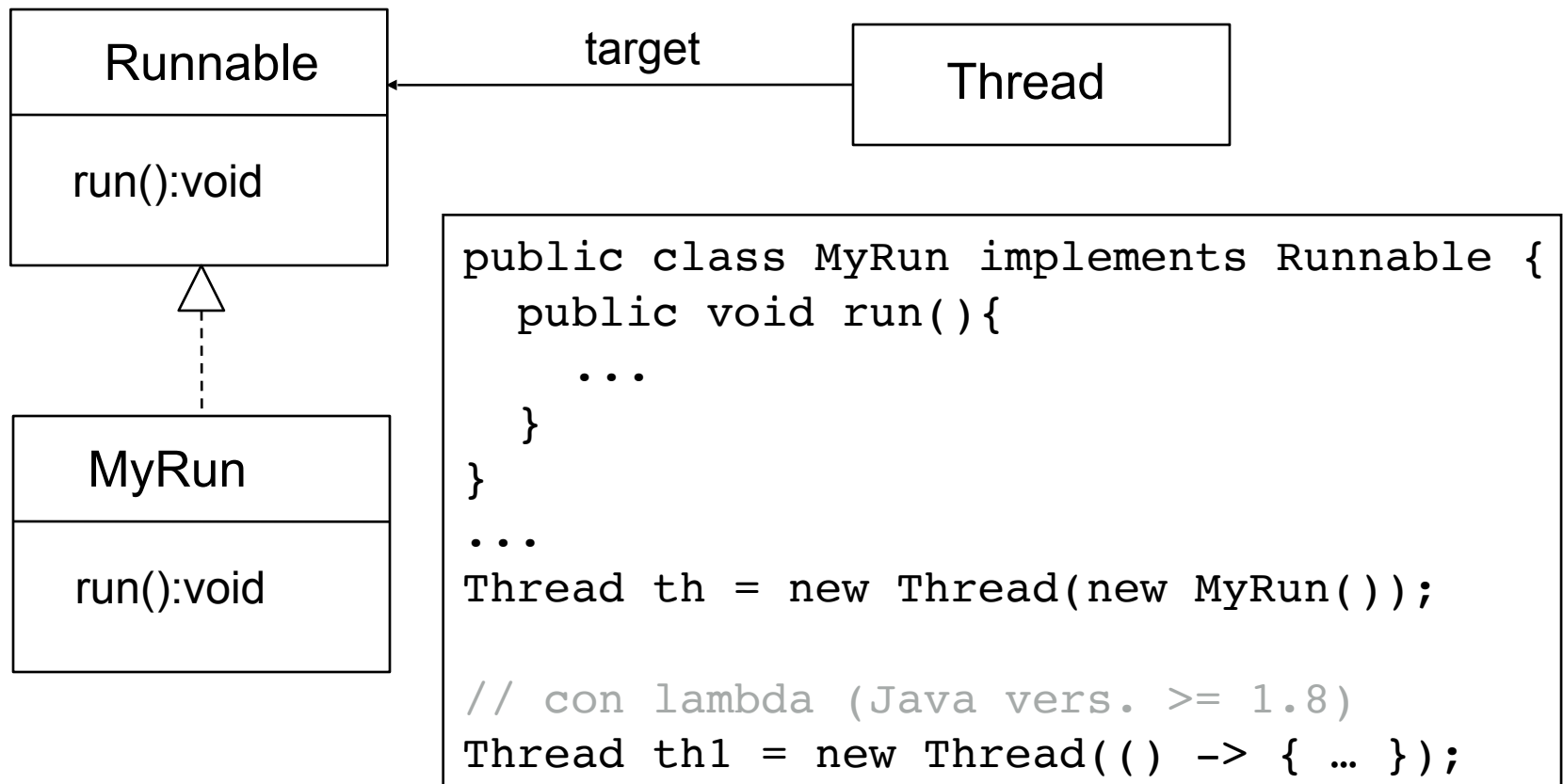


NOTE

- Thread e main app
 - un'applicazione Java ha sempre almeno un flusso di controllo in esecuzione
 - è il main thread, che esegue il metodo statico main
 - in realtà ha più di un thread
 - garbage collector
 - profiler / debugger listeners, RMI listeners, etc
- start() vs. run()
 - cosa succede se invochiamo il metodo run() anziché il metodo start() per eseguire un thread?

INTERFACCIA Runnable

- Esiste anche un secondo modo per definire un thread, basato su interfacce, utile quando la classe che funge da thread è già parte di una gerarchia di ereditarietà e non può derivare da Thread



CLOCK WITH RUNNABLE

- Esempio Clock usando un'espressione lambda per definire il task:

```
new Thread(new Runnable() {  
    public void run() {  
        while (true) {  
            System.out.println(new Date());  
            try {  
                Thread.sleep(2000);  
            } catch (Exception ex){}  
        }  
    }  
}).start();
```

- Approccio utile in particolare quando il thread rappresenta un compito di durata finita e limitata, da svolgere in modo asincrono rispetto al thread che lo lancia

MONITORING THREADS: JConsole TOOL

- Java Monitoring and Management Console, a tool grafico fornito con il JDK
 - usa il supporto nativo della JVM per monitoraggio performance e uso risorse
 - Java Management Extension (JMX) technology
 - <http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>
- Tra le varie funzionalità, permette di monitorare i thread lanciati

THINK CONCURRENT

- Fra i principali benefici della programmazione concorrente => aumentare le performance sfruttando al meglio le risorse HW parallele
 - nel caso della programmazione multithread questo significa *ripensare alla soluzione di problemi e la progettazione di algoritmi/strutture dati / sistemi in termini di insiemi (eventualmente dinamici) di thread ognuno dei quali svolge una parte del compito complessivo e cooperano opportunamente al fine di ottenere l'obiettivo finale*
- A design level
 - thread come **componenti attivi**, che eseguono con un *flusso di esecuzione autonomo* un certo compito o *task* ben definito
 - vs. oggetti come componenti passivi
 - nuovo livello per concepire *l'organizzazione* di un programma
 - ogni componente attivo => *modulo* che *incapsula* un flusso di controllo
 - incapsulamento stato + comportamento + controllo
 - in qualsiasi applicazione non banale i componenti attivi devono opportunamente interagire e coordinarsi
 - prossimo modulo

ESEMPIO: CONCURRENT SORTING

- Consideriamo come esempio il problema dell'ordinamento degli elementi di un vettore
 - algoritmi: QuickSort, MergeSort, BubbleSort, HeapSort...
 - tutti sono sequenziali
- Soluzione multithread? Quale design? Un esempio
 - suddivisione del lavoro in più **worker** come componenti attivi ognuno dei quali si occupa dell'ordinamento di una porzione del vettore (usando algoritmi tradizionali)
 - thread principale funge da **master** che si occupa di lanciare i worker e attendere che abbiano completato il loro lavoro, per effettuare un merge
 - complessità lineare

UN PRIMO MECCANISMO DI SINCRONIZZAZIONE: JOIN

- Un thread può attendere la terminazione di un altro thread invocandone il metodo join

```
...  
MyThread thread = new MyThread();  
thread.start();  
System.out.println("Waiting for thread termination...");  
thread.join();  
System.out.println("Thread completed.");
```

- Forme più articolate di sincronizzazione verranno discusse nel prossimo modulo

ESEMPIO CONCURRENT SORTING CON MASTER + 2 WORKER THREADS

- Master Thread

```
public class SortingMaster extends Thread {  
  
    private int[] vectorToSort;  
  
    public SortingMaster(int[] v) {  
        vectorToSort = v;  
    }  
  
    public void run() {  
        int middle = vectorToSort.length / 2;  
        SortingWorker w1 = new SortingWorker(0, middle, vectorToSort);  
        SortingWorker w2 = new SortingWorker(middle, vectorToSort.length, vectorToSort);  
        w1.start();  
        w2.start();  
        try {  
            w1.join();  
            w2.join();  
            merge(vectorToSort, 0, middle, middle, vectorToSort.length);  
        } catch (InterruptedException ex) {}  
    }  
}
```

ESEMPIO CONCURRENT SORTING CON MASTER + 2 WORKER THREADS

- Worker Thread

```
class SortingWorker extends Thread {  
  
    private int minIndex, maxIndex;  
    private int[] data;  
  
    public SortingWorker(int minIndex, int maxIndex, int[] data){  
        this.minIndex = minIndex; this.maxIndex = maxIndex; this.data = data;  
    }  
  
    public void run(){  
        Arrays.sort(data, minIndex,maxIndex);  
    }  
}
```

CONFRONTO PERFORMANCE

- Test performance
 - SequentialSortTest
 - ConcurSortTest

```
% java -cp bin oop.concur.SequentialSortTest
Generating array...
Array generated.
Sequential sorting (100000000 elements)...
Done. Time elapsed: 8735 ms
SORTING OK.
```

```
% java -cp bin oop.concur.ConcurSortTest
Generating array...
Array generated.
Concurrent sorting (100000000 elements)...
Thread-2 starting 50000000-100000000
Thread-1 starting 0-50000000
Thread-2 done 50000000-100000000
Thread-1 done 0-50000000
Done. Time elapsed: 4575 ms
SORTING OK.
```


NOTE

- Come generalizzare la strategia a 4, 8, N thread/processi?
- Peso della parte sequenziale della strategia (merge...)
 - quale strategia per migliorare questo aspetto?

ESEMPIO FALLING WORDS

- Esempio TestFallingWords nel materiale in cui si fa “precipitare” una frase parola per parola, con velocità diverse, dalla parte alta dello schermo (console) alla parte bassa
 - si suddivide il compito considerando un worker thread(“WordFallingAgent”) dedicato al compito specifico di far precipitare la singola parola con una certa velocità
 - il risultato complessivo si ottiene mandando in esecuzione concorrentemente tutti i thread, uno per ogni parola
 - strategia più semplice rispetto a dover usare un solo thread che si occupa di tutte le parole

COMPORTAMENTO CPU-BOUND O IO-BOUND DEI THREAD

- Il comportamento di un thread può essere CPU-bound o IO-bound a seconda esegua esclusivamente operazioni di calcolo/computazione oppure esegua anche interazioni (ad esempio di I/O) che coinvolgano la sua sospensione/blocco da parte del sistema operativo
- La natura CPU-bound o I/O bound impatta notevolmente sulle performance in generale
- Esempi
 - esempio BouncingBalls (nel materiale)
 - esempio InfiniteLoops

ULTERIORI ASPETTI DI BASE

- Terminazione di un thread
- Ottenere il numero di processori a disposizione

TERMINAZIONE DI UN THREAD

- La terminazione di un thread (thread cancellation) consiste nella terminazione della sua attività prima del suo completamento. Il thread da terminare prende in genere il nome di target thread.
 - Ad esempio si pensi ad un insieme di thread con il medesimo compito di ricerca di informazioni in un archivio: non appena uno trova le informazioni, gli altri possono essere fermati.
 - Oppure al thread relegato al caricamento di una pagina in un Web Browser, al momento in cui si preme il pulsante di stop.
- In genere si considerano due tipi di terminazioni:
 - asynchronous cancellation
 - un thread ne termina immediatamente il target thread
 - **deferred cancellation**
 - il target thread controlla periodicamente se deve terminare
- I problemi relativi alla terminazione di un thread sono per lo più legati alle risorse che tale thread può aver bloccato o comunque averne possesso: in particolare ciò è problematico nel caso di asynchronous cancellation.

TERMINAZIONE DI UN THREAD IN JAVA

- La terminazione di un thread può essere sia di tipo asincrono, sia deferred.
 - nel primo caso - *deprecato* - la terminazione avviene invocando il metodo **stop()**
 - nel secondo caso la terminazione avviene controllando nel metodo `run()` stesso che non si siano verificate le condizioni per la terminazione.
- Un esempio consiste nell'utilizzo del metodo `isInterrupted` che valuta se è stata richiesta l'interruzione del thread (mediante metodo `interrupt`): in tal caso si fa in modo di terminare il metodo `run`.

DEFERRED CANCELLATION: ESEMPIO

```
public class StoppableClock extends Thread {
    private int step;
    private volatile boolean stopped;

    public Clock(int step){
        this.step = step;
        this.stopped = false;
    }

    public void run(){
        while (!stopped) {
            System.out.println(new Date());
            try {
                sleep(step);
            } catch (Exception ex){
                System.out.println("Interrupted!");
            }
        }
    }

    public void forceStop(){
        stopped = true;
        interrupt();
    }
}
```

ATTRIBUTO VOLATILE

- L'attributo volatile specificato per un campo implica che ogni thread che voglia accedere al campo in lettura, legga effettivamente il valore corrente in memoria e non una eventuale versione che tiene in cache/nello stack
 - questo potrebbe capitare in seguito ad ottimizzazioni operate dal compilatore e comporta problemi in caso di accessi concorrenti..
- In particolare:
 - l'accesso in lettura e scrittura di campi di tipo boolean è garantito essere atomico a livello di JVM
 - questo vale anche per int, non vale invece per double e long
 - per cui - nell'esempio - non ci sono corse critiche nell'accesso concorrente da parte del thread stesso nel metodo run (...! stopped...) e di un altro thread che invoca il metodo forceStop
 - ciò che invece può capitare - se non si specifica volatile - è che il valore di stopped letto in run non sia quello effettivamente in memoria, ma una versione in cache
 - eventualmente errata, se il campo è stato aggiornato da forceStop..

INTERRUPT

- Il metodo `interrupt` ha effetto sul thread chiamato nel caso esso sia bloccato su metodi come `sleep`, `wait` (prossimo modulo) che vengono quindi sbloccati generando un'eccezione di tipo `InterruptedException`
 - che è parte della signature del metodo
- Nell'esempio, nel caso in cui il thread `StoppableClock` sia bloccato sulla `sleep`, questo viene immediatamente sbloccato e quindi può uscire dal ciclo `while`

OTTENERE NUMERO PROCESSORI

- Funzionalità fornita dalla classe Runtime presente in java.lang
 - metodo **availableProcessors()**
 - restituisce il numero di processori HW logici presenti nel sistema
 - logici => incluso hyper-threading
 - da invocare sull'unico oggetto Runtime presente nel sistema (pattern singleton)

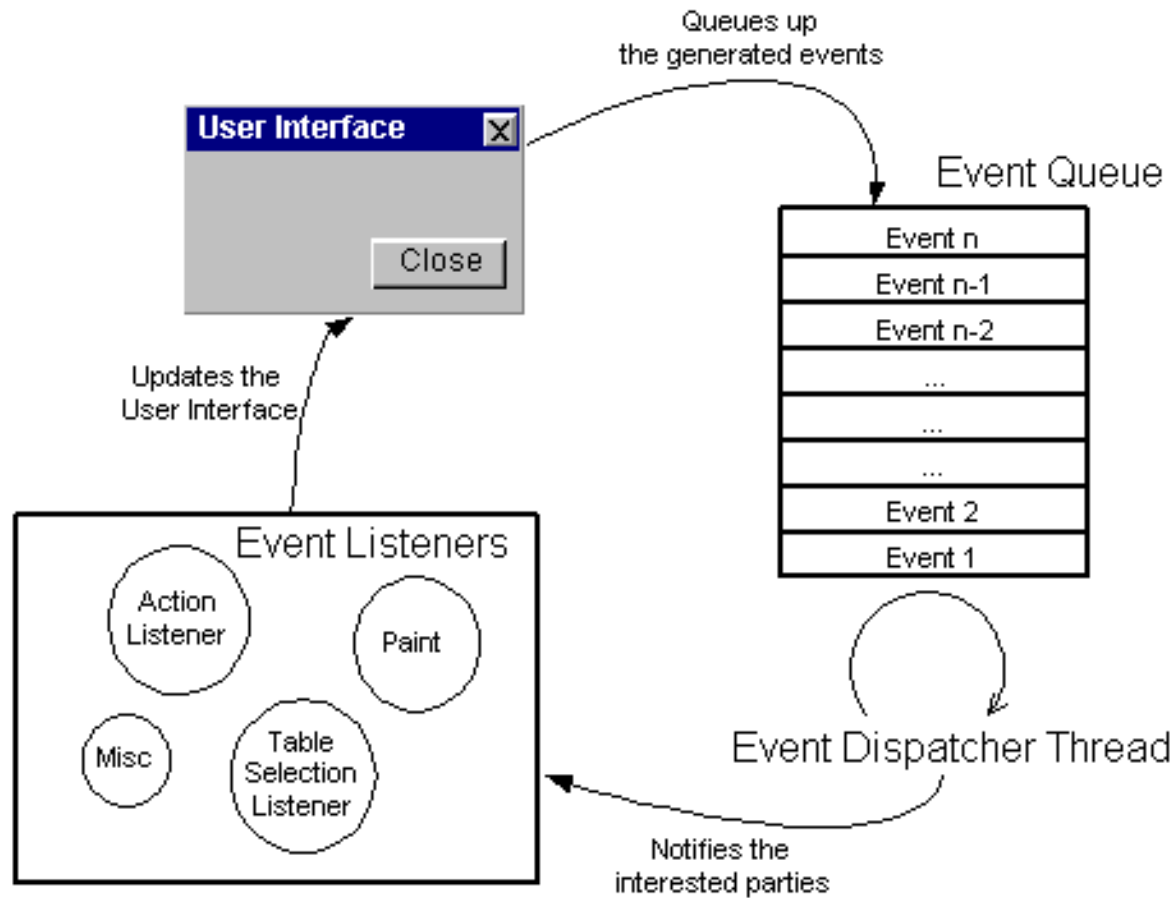
```
public class PrintAvailableProcs {  
    public static void main(String[] args) {  
        int nprocs = Runtime.getRuntime().availableProcessors();  
        System.out.println("Available processors: "+nprocs);  
    }  
}
```

MULTITHREADING E INTERFACCE GRAFICHE

MULTITHREADING E INTERFACCE GRAFICHE

- I sottosistemi grafici tipicamente usano un unico thread per processare gli eventi che concernono i componenti della GUI
 - tipicamente gestiti con una singola coda di eventi
 - **architettura di controllo event-loop**
- Un esempio è dato dalla libreria Java **Swing**
 - EDT (Event Dispatcher Thread)
 - creato inizializzato alla prima creazione di un componente Swing
 - elabora i vari eventi di qualsiasi frame o più in generale componente, chiamando opportunamente i listener agganciati
- Stessa architettura anche per **JavaFX**
- Conseguenze importanti
 - se l'esecuzione da parte del thread di un metodo del listener è particolarmente pesante dal punto di vista computazionale, l'interfaccia grafica “perde reattività”
 - caso estremo: se l'esecuzione da parte del thread di un metodo di un listener entra in loop infinito, tutta l'interfaccia grafica rimane bloccata

GUI EVENT LOOP



[Figura tratta da <https://www.javaworld.com/article/2073477/swing-gui-programming/customize-swingworker-to-improve-swing-guis.html>]

ESEMPIO: MANDIAMO IN STALLO Swing...

```
class MyFrame extends JFrame implements ActionListener {

    public MyFrame(){
        super("Test Swing thread");
        setSize(120,60);
        setVisible(true);
        JButton button = new JButton("test");
        button.addActionListener(this);
        getContentPane().add(button);
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent ev){
                System.exit(-1);
            }
        });
    }

    public void actionPerformed(ActionEvent e) {
        while (true) {}
    }

}

public class TestSwingThread {
    static public void main(String[] args){
        new MyFrame();
        new MyFrame();
    }
}
```

THREAD-SAFETY

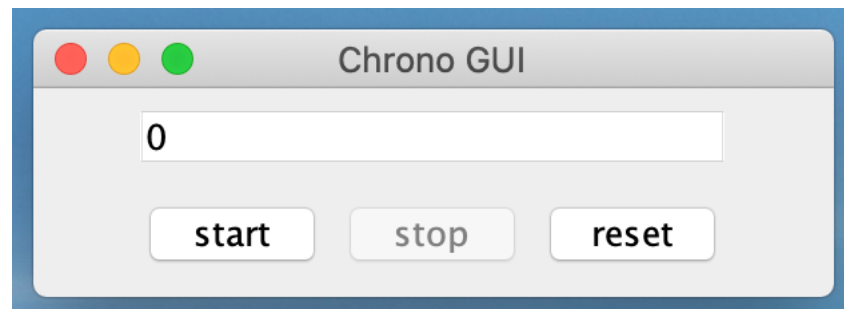
- In Swing - come nella maggior parte dei GUI toolkit - le classi che rappresentano componenti dell'interfaccia non sono thread-safe
 - questo perché si presuppone che l'unico thread che vi acceda sia l'EDT
- Questo implica che se altri thread devono interagire con tali componenti, non lo possono fare direttamente, ma devono interagire con l'EDT
 - i metodi `invokeXXX` in `SwingUtilities` lo fanno inserendo il compito specificato (istanza di una classe che implementa `Runnable`) in una coda che acceduta dall'EDT
 - analogo all'invio di un messaggio
 - **`invokeLater`** => accoda e non attende lo svolgimento del compito
 - **`invokeAndWait`** => accoda e attende che l'EDT abbia svolto il compito

PROGRAMMAZIONE MULTITHREAD PER APPLICAZIONI INTERATTIVE: QUADRO

- L'utilizzo di thread diviene dunque indispensabile per la realizzazione di applicazioni interattive e reattive, in cui l'interfaccia grafica deve rispondere con una certa prontezza all'input dell'utente
- Per realizzare interfacce reattive e corrette, valgono le seguenti regole:
 - non utilizzare mai il thread che gestisce gli eventi dell'interfaccia grafica per svolgere compiti pesanti
 - usare quindi thread separati per l'esecuzione di attività che possono comportare un certo impegno di risorse spazio temporali si devono
 - i thread possono essere creati on-the-fly nei listener, oppure si utilizzano pool di thread pre-esistenti e schemi produttore-consumatore
 - qualora thread diversi dall'EDT debbano interagire con oggetti/componenti dell'interfaccia grafica, non si devono invocare direttamente i metodi di tali oggetti ma si devono utilizzare i metodi `invokeLater` and `invokeAndWait` in `SwingUtilities`

ESEMPIO: COSTRUIAMO UN CRONOMETRO...

- Problema
 - si vuole pilotare un conteggio mediante una interfaccia grafica dotata di tre pulsanti (start, stop, reset) e un edit box dove visualizzare il conteggio
 - alla pressione di start, il conteggio deve partire e venire incrementato ogni decimo di secondo, alla pressione di stop il conteggio si ferma e reset lo riporta a zero



ARCHITETTURA BASATA SU MVC + THREAD

- Adottiamo un'architettura basata sul pattern model view controller (MVC) estesa per integrare opportunamente i thread per mantenere la reattività dell'interfaccia
 - Model
 - ChronoCount
 - View
 - ChronoView
 - ChronoFrame (Swing specific)
 - Controller
 - Controller (interfaccia) e ChronoController (impl.)
 - ChronoAgent (thread)
- Due soluzioni proposte
 - nella prima soluzione (oop.concur.chrono) la responsabilità dell'aggiornamento della View è a carico del controller
 - nella seconda soluzione (oop.concur.chrono2), viene applicato in modo completo il pattern MVC e la View funge da osservatore del model

NOTE

- Il controller è costituito da due parti: una passiva (ChronoController) e una parte attiva (ChronoAgent)
 - I metodi della parte passiva vengono chiamati dall'EDT per notificare eventi dalla GUI (start, stop, reset)
 - La parte passiva crea e manda in esecuzione la parte attiva, che provvede a far partire il processo di conteggio con un flusso di controllo proprio (ChronoAgent)
- In entrambe le soluzioni, il flusso di controllo che chiama i metodi della parte view (ChronoView) è quello del ChronoAgent, un flusso di controllo diverso da quello dell'EDT
 - motivo per cui nei metodi in ChronoView si delega il compito all'EDT mediante il metodo `SwingUtilities.invokeLater`

BIBLIOGRAFIA PER APPROFONDIMENTI

- Concurrent Programming in Java: Design Principles and Pattern, 2/E
- Doug Lea - Addison-Wesley Professional
 - testo di riferimento per la programmazione concorrente in Java
- Java Concurrency in Practice - Brian Goetz et al - Addison Wesley
Software and the Concurrency Revolution - by Herb Sutter, James Larus - ACM Queue - <http://queue.acm.org/detail.cfm?id=1095421>
- Multithreaded toolkits: A failed dream? - Graham Hamilton - <http://news.jchk.net/article/a2c3e767100ce327f46c1ad9d0e89e7db19ab227>