

Lezione 1

Introduzione al corso

JAVA Thread Programming

Laboratorio di

Programmazione di Reti - B

14/09/2021

Federica Paganelli

federica.paganelli@unipi.it

Introduzione al corso

Informazioni utili

- Riferimenti
 - Federica Paganelli (federica.paganelli@unipi.it)
 - Ricevimento: venerdì 14:00-16:00 e su appuntamento (fissare tramite e-mail) sul canale [Ricevimento](#)
- Moodle <https://elearning.di.unipi.it/course/view.php?id=259>
 - pubblicazione contenuti
 - forum, chats...
 - assignments

Orario

- In presenza
 - Martedì 11:00 - 12:45 Aula A1
 - Presentazione concetti
 - Martedì 14:15 - 16:00 Aula A1 BYOD*
 - Esercitazione in aula, correzione esercizi e assignment
- Teams
 - Canale Lezioni LABORATORIO DI RETI B del Team 274AA 20/21 - RETI DI CALCOLATORI E LABORATORIO DI RETI - DISPARI [INF-L]
[https://teams.microsoft.com/l/channel/19%3a9ad256ba43d24cc5bc854851c6cecbdd%40thread.tacv2/Lezioni%2520LABORATORIO%2520DI%2520RETI%2520\(B\)?groupId=6dab3cb8-663c-48f9-b3d7-53e484ca2f60&tenantId=c7456b31-a220-47f5-be52-473828670aa1](https://teams.microsoft.com/l/channel/19%3a9ad256ba43d24cc5bc854851c6cecbdd%40thread.tacv2/Lezioni%2520LABORATORIO%2520DI%2520RETI%2520(B)?groupId=6dab3cb8-663c-48f9-b3d7-53e484ca2f60&tenantId=c7456b31-a220-47f5-be52-473828670aa1)

Informazioni utili

- esperienze pratiche in laboratorio
 - obiettivo: verifica esercizi assegnati nelle lezioni teoriche
 - se si consegna entro 14 giorni dalla data dell'assegnazione l'80% degli esercizi, sarà possibile discuterli all'esame e, se la discussione è positiva, ottenere un bonus di 2 punti.

Informazioni utili

- **modalità di esame** (modulo di Laboratorio):
 - **Consegna progetto + prova orale**
 - l'esame del modulo di laboratorio (discussione del progetto + orale) può essere sostenuto indipendentemente da quello del modulo teorico di Reti.
 - per accedere alla prova orale conclusiva è necessario ottenere una valutazione sufficiente del progetto.
 - Le specifiche di progetto saranno consegnate all'inizio di dicembre e rimarranno valide fino a settembre 2022 (novembre per appello straordinario).
- **Orale**
 - discussione del progetto + domande su tutti argomenti trattati nelle lezioni teoriche di laboratorio
 - l'esame si tiene su appuntamento, nel periodo temporale previsto per gli appelli di esame.

INFORMAZIONI UTILI

Prerequisiti

- corso di Programmazione 2, conoscenza del linguaggio JAVA, in particolare:
 - packages
 - gestione delle eccezioni
 - collections
 - generics
- Modulo di reti: conoscenza TCP/IP
 - Linguaggio di programmazione di riferimento: JAVA 8
 - Ambiente di sviluppo di riferimento: Eclipse

Materiale didattico

- Lucidi delle lezioni (su Moodle)
- Testi consigliati (non obbligatori)
 - Bruce Eckel – [Thinking in JAVA](#) – Volume 3 - Concorrenza e Interfacce Grafiche
 - B. Goetz, [JAVA Concurrency in Practice](#), 2006
 - Esmond Pitt [Fundamental Networking in JAVA](#)
- Materiale di Consultazione
 - Harold, JAVA Network Programming 3rd edition O'Reilly 2004.
 - K.Calvert, M.Donhaoo, TCP/IP Sockets in JAVA, Practical Guide for Programmers
 - Costrutti di base Horstmann, Concetti di Informatica e Fondamenti di Java 2
 - JAVA NIO O'Reilly

Programma preliminare del corso

- Threads

- creazione ed attivazione di threads
- meccanismi di gestione di pools di threads
- mutua esclusione, lock implicite ed esplicite
- il concetto di monitor: sincronizzazione di threads su strutture dati condivise: `synchronized`, `wait`, `notify`, `notifyall`
- concurrent collections

- Stream ed IO

- Streams: tipi di streams, composizione di streams
- meccanismi di serializzazione: serializzazione standard di JAVA, JSON

- Non blocking IO

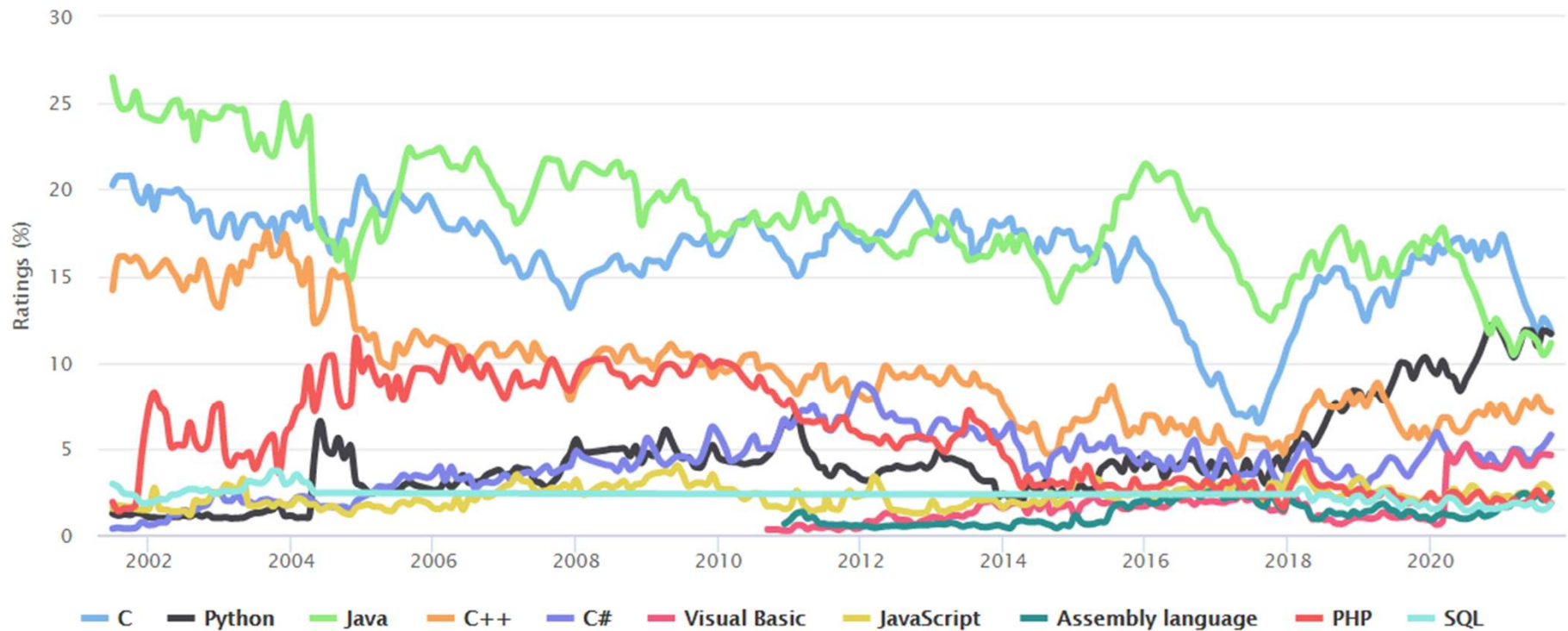
- Channels, buffers

Programma preliminare del corso

- Programmazione di rete a basso livello
 - connection oriented Sockets
 - connectionless sockets: UDP, multicast
 - Non blocking IO e sockets, selectors
- Oggetti Distribuiti
 - definizione di oggetti remoti
 - il meccanismo di Remote Method Invocation in JAVA
 - dynamic code loading
 - problemi di sicurezza
 - il meccanismo delle callbacks
- Stile architetturale REST
 - Vincoli dello stile architetturale REST
 - REST e HTTP
 - Esempio sviluppo REST server

TIOBE Programming Community Index

Source: www.tiobe.com



- indice TIOBE: misura la popolarità dei linguaggi di programmazione in funzione del numero di ricerche contenenti il nome del linguaggio
- Java uno dei top-3 linguaggi: utile studiarlo!

Evoluzione di Java



23 JAN 1996 - JAVA 1

First public release. The stable version Java 1.0.2 is called Java 1.

8 DEC 1998 - JAVA 1.2

Swing, JIT Compiler, Collections

6 FEB 2002 - JAVA 1.4

Assertions, RegEx Improvements, Image IO API, XML Parsers, XSLT Processors, Preferences API

1995 - JDK BETA

The first beta version of Java. Developed by James Gosling at Sun Microsystems.

19 FEB 1997 - JAVA 1.1

Inner Classes, Java Beans, JDBC, RMI

8 MAY 2000 - JAVA 1.3

HotSpot JVM, JNDI, JPDA

30 SEP 2004 - JAVA 5

Generics API, Varargs, for-each loop, Autoboxing, Enum, Annotations, Static Imports

Evoluzione di Java



Ultima versione JAVA 16
In questo corso faremo
riferimento a JAVA8

In Questo Corso: Classi In Blu

- 1.0.2 prima versione stabile, rilasciata il 23 gennaio del 1996
 - AWT Abstract Window Toolkit, applet
 - Java.lang (supporto base per concorrenza), Java.io, Java.util
 - Java.net (socket TCP ed UDP, Indirizzi IP, ma non RMI)
- 1.1: RMI, Reflections,....
- 1.2: Swing (grafica), RMI-IIOP, ...
- 1.4: regular expressions, assert, NIO, IPV6
- JAVA 5: una vera rivoluzione generics, concorrenza,....
- 7: acquisizione da parte di Oracle: framework fork and join
- 8: Lambda Expressions
- ...e così via

Programmazione concorrente in Java

Concurrent Programming: Motivazione

- Concetto di base: gestire più di un task alla volta (il programma principale interrompe quello che stava facendo, gestisce un altro problema e poi riprende l'esecuzione precedente)
 - Es. Applicazioni con interfaccia utente
- La programmazione concorrente permette di partizionare il programma in sequenze di operazioni (task) separate, eseguite in modo indipendente (non sequenziale).
- Queste porzioni di programma appaiono come in esecuzione in parallelo (singolo processore) – sono un modo di allocare il tempo di un singolo processore. Ma se sono a disposizione più processore i task possono essere eseguiti in parallelo.

Concurrent Programming: Motivazioni

- concurrent programming: una storia iniziata molti anni fa, diventata un “hot topic”

legge di Moore (Gordon Moore, co-fondatore di INTEL):

"the number of transistors that can be fit onto a square inch of silicon doubles every ~~12~~ 24 months”*

https://it.wikipedia.org/wiki/Legge_di_Moore

- riprogettazione dei processori verso sistemi multicore
- come distribuire il carico tra i diversi core?
 - nuovi linguaggi: Scala, Akka, nuove librerie di JAVA, GO,...
 - nuove metodologie di sviluppo del software

Problema: risorse condivise, necessità di meccanismi per la gestione della concorrenza (lock)

* Questo non sembra essere avvenuto negli ultimi anni

<https://www.tomshw.it/hardware/la-legge-di-moore-e-morta-meglio-ci-sara-piu-innovazione/>

java.util.concurrent framework

- JAVA < 5 built in for concurrency: lock implicite, wait, notify e poco più.
- java.util.concurrent
 - un toolkit general purpose per lo sviluppo di applicazioni concorrenti.
- no more “reinventing the wheel”!
- definire un insieme di utility che risultino:
 - standardizzate
 - facili da utilizzare e da capire
 - high performance
 - utili in un grande insieme di applicazioni per un vasto insieme di programmatori, da quelli più esperti a quelli meno esperti.

java.util.concurrent framework

sviluppato in parte da Doug Lea, disponibile come insieme di librerie JAVA non standard prima della integrazione in JAVA 5.0.

tra i package principali:

- `java.util.concurrent`
 - executor, concurrent collections, semaphores,...
 - `java.util.concurrent.atomic`
 - `AtomicBoolean`, `AtomicInteger`,...
- `java.util.concurrent.locks`
 - `Condition`
 - `Lock`
 - `ReadWriteLock`

JAVA 5 CONCURRENT FRAMEWORK

- **Executors**
 - `Executor`
 - `ExecutorService`
 - `ScheduledExecutorService`
 - `Callable`
 - `Future`
 - `ScheduledFuture`
 - `Delayed`
 - `CompletionService`
 - `ThreadPoolExecutor`
 - `ScheduledThreadPoolExecutor`
 - `AbstractExecutorService`
 - `Executors`
 - `FutureTask`
 - `ExecutorCompletionService`
- **Queues**
 - `BlockingQueue`
 - `ConcurrentLinkedQueue`
 - `LinkedBlockingQueue`
 - `ArrayBlockingQueue`
 - `SynchronousQueue`
 - `PriorityBlockingQueue`
 - `DelayQueue`
- **Concurrent Collections**
 - `ConcurrentMap`
 - `ConcurrentHashMap`
 - `CopyOnWriteArray{List,Set}`
- **Synchronizers**
 - `CountDownLatch`
 - `Semaphore`
 - `Exchanger`
 - `CyclicBarrier`
- **Locks: `java.util.concurrent.locks`**
 - `Lock`
 - `Condition`
 - `ReadWriteLock`
 - `AbstractQueuedSynchronizer`
 - `LockSupport`
 - `ReentrantLock`
 - `ReentrantReadWriteLock`
- **Atomics: `java.util.concurrent.atomic`**
 - `Atomic{Type}`
 - `Atomic{Type}Array`
 - `Atomic{Type}FieldUpdater`
 - `Atomic{Markable,Stampable}Reference`

Threads in Java

Thread: definizione

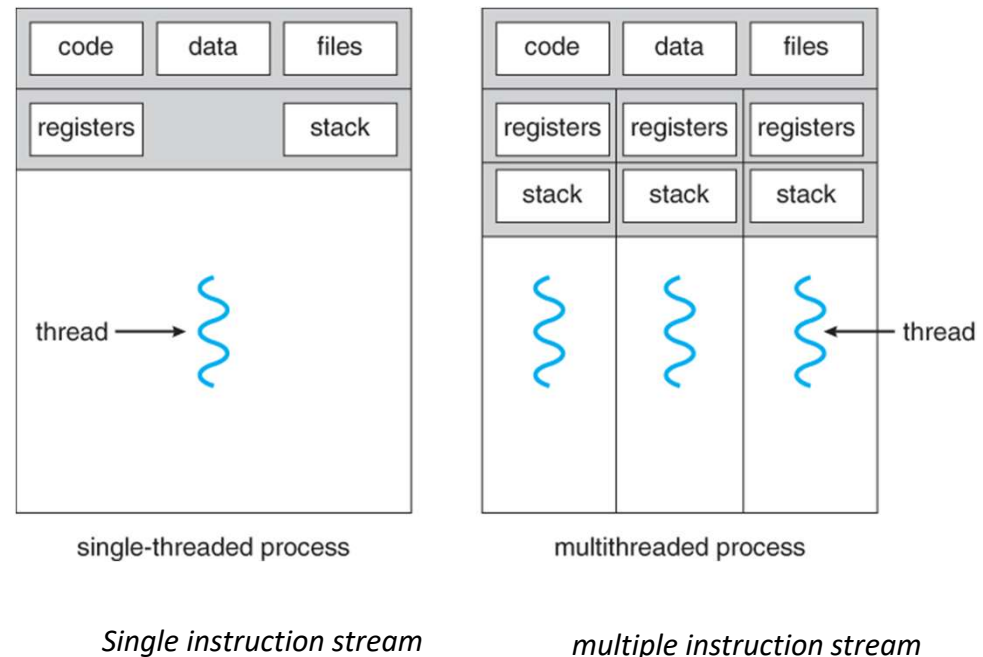
- Processo: istanza di un programma in esecuzione
 - esempio: se mando in esecuzione due diverse applicazioni, ad esempio MS Word, MS Access, vengono creati due processi
- Thread (light weight process): un **flusso di esecuzione** all'interno di un processo. Ogni processo ha almeno un thread. I thread condividono le risorse di un processo.
- multitasking, si può riferire a thread o processi
 - a livello di processo è controllato esclusivamente dal sistema operativo
 - a livello di thread è controllato, almeno in parte, dal programmatore
- Multi-threaded execution in java: il main thread di un programma può attivare altri thread
- esecuzione dei thread:
 - single core: es. interleaving (meccanismi di time sharing,...)
 - multicore: più flussi in esecuzione eseguiti in parallelo, simultaneità di esecuzione

PROCESSI E THREADS

Contesto di un processo: insieme delle informazioni necessarie per ristabilire esattamente lo stato in cui si trova il sistema al momento in cui se ne interrompe l'esecuzione per passare ad un altro (stato dei registri del processore, memoria del processo, etc.).

Thread verso process multitasking:

- sono meno costosi
 - il cambiamento di contesto tra thread
 - la comunicazione tra thread

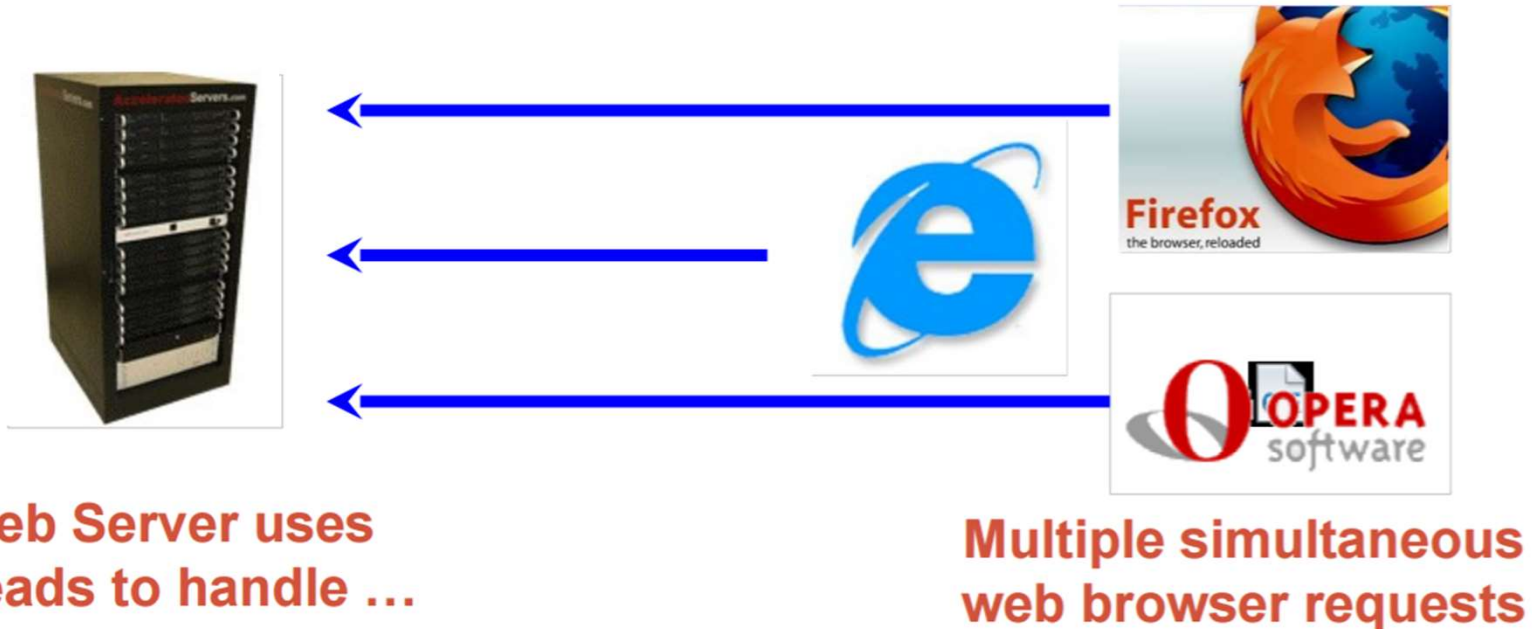


Multithreading: perché?

- Struttura di un browser:
 - visualizza immagini sullo schermo
 - controlla input dalla keyboard e da mouse
 - invia e ricevi dati dalla rete
 - legge e scrive dati su un file
 - esegue computazione (editor, browser, game)
- come gestire tutte queste funzionalità simultaneamente? una decomposizione del programma in threads implica:
 - modularizzazione della struttura dell'applicazione
 - aumento della responsiveness
- altre applicazioni complesse che richiedono la gestione contemporanea di più attività ad esempio: applicazioni interattive distribuite come giochi multiplayer
 - Ad es. interazione con l'utente + messaggi da altri giocatori (dalla rete...)

Multithreading: perché?

- cattura la struttura della applicazione
 - molte componenti interagenti
 - ogni componente gestita da un thread separato
 - semplifica la programmazione della applicazione



MULTITHREADING: PERCHE'?

- **struttura di un server sequenziale**

```
while(server is active){  
    listen for request  
    process request  
}
```

- ogni client deve aspettare la terminazione del servizio della richiesta precedente
- responsiveness limitata

- **struttura di un server multithreaded**

```
while(server is active){  
    listen for request  
    hand request to worker thread  
}
```

- un insieme di worker thread per ogni client
- aumento responsiveness

Multithreading: perché?

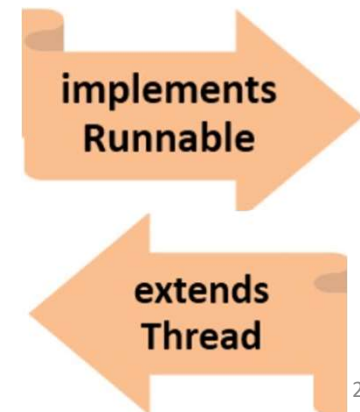
- migliore utilizzazione delle risorse
 - quando un thread è sospeso, altri thread vengono mandati in esecuzione
 - riduzione del tempo complessivo di esecuzione
- migliore performance per computationally intensive application
 - dividere l'applicazione in task ed eseguirli in parallelo
- tanti vantaggi, ma anche alcuni problemi:
 - più difficile il debugging e la manutenzione del software rispetto ad un programma single threaded
 - race conditions, sincronizzazioni
 - deadlock, livelock, starvation,...

Creazione ed attivazione di thread

- Java è un linguaggio concorrente: costrutti linguistici per la gestione di thread, concorrenza ecc.
- quando si manda in esecuzione un programma JAVA, la JVM crea un thread in corrispondenza dell'esecuzione del metodo main del programma
 - almeno un thread per ogni programma
 - altri thread sono attivati automaticamente da JAVA (gestore eventi, interfaccia, garbage collector,...).
 - ogni thread durante la sua esecuzione può creare ed attivare altri threads.
- un thread è un *oggetto* che della classe Thread. Come creare ed attivare un thread? Due metodi:

Metodo 1: definire un task, ovvero definire una classe C che implementi l'interfaccia Runnable, creare un'istanza di C, quindi creare un thread passandogli l'istanza del task creato

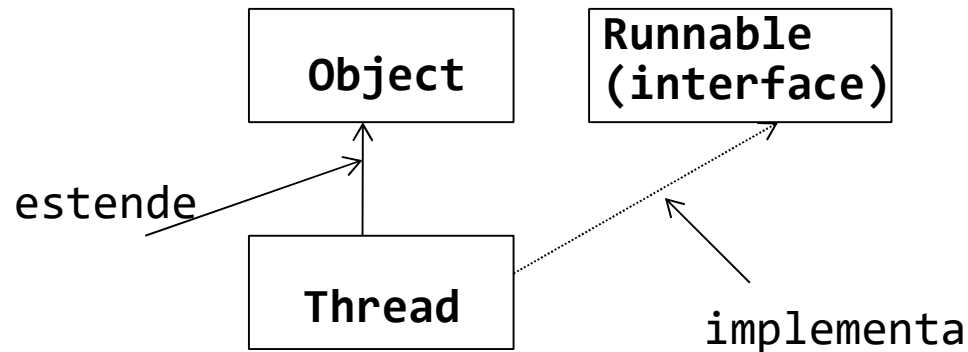
Metodo 2: estendere la classe java.lang.Thread



Metodo 1: Runnable Interface

- appartiene al package **java.language**
- contiene solo la segnatura del metodo **void run()**
- ogni classe C che implementa l'interfaccia Runnable deve fornire un'implementazione del metodo **run()**
- un oggetto istanza di C è un task: un **frammento di codice** che può essere eseguito in un thread
 - la creazione del task non implica la creazione di un thread che lo esegua.
 - lo stesso task può essere eseguito da più threads: un solo codice, più esecutori
- Per eseguire il task si istanzia un oggetto di tipo Thread e si passa un riferimento al task che il thread deve eseguire
- Si invoca il metodo **start()** dell'oggetto di tipo Thread

La gerarchia delle classi: runnable



- runnable: una variabile locale della classe Thread, memorizza un oggetto Runnable passato come parametro

- il metodo run() della classe Thread:

```
public void run()  
{ if (runnable != null)  
    runnable.run(); }
```

- l'invocazione del metodo start() provoca la esecuzione del metodo run() che, a sua volta, provoca l'esecuzione del metodi run() della Runnable

Attivazione di threads: un esempio

- Scrivere un programma che stampa le tabelline moltiplicative dall' 1 al 10
 - si attivino 10 threads
 - ogni numero n , $1 \leq n \leq 10$, viene passato ad un thread diverso
 - il task assegnato ad ogni thread consiste nello stampare la tabellina corrispondente al numero che gli è stato passato come parametro

Task Calculator

```
public class Calculator implements Runnable {  
    private int number;  
    public Calculator(int number) {  
        this.number=number; }  
    public void run() {  
        for (int i=1; i<=10; i++){  
            System.out.printf("%s: %d * %d = %d\n",  
                Thread.currentThread().getName(),number,i,i*number);  
        }  
    }  
}
```

- più thread potranno eseguire il codice di Calculator
- qual è il thread che sta eseguendo attualmente questo codice?

CurrentThread() restituisce un riferimento al thread che sta eseguendo il segmento di codice all'interno del quale si trova la sua invocazione

Main program

```
public class CalculatorMain {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; i++){  
            Calculator calculator=new Calculator(i);  
            Thread thread=new Thread(calculator);  
            thread.start();  
            System.out.println("Avviato Calcolo Tabelline"); } } }
```

L'output Generato dipende dalla schedulazione effettuata, ad es:

Thread-0: 1 * 1 = 1

Thread-9: 10 * 1 = 10

Thread-5: 6 * 1 = 6

Thread-8: 9 * 1 = 9

Thread-7: 8 * 1 = 8

Thread-6: 7 * 1 = 7

Avviato Calcolo Tabelline

Thread-4: 5 * 1 = 5

Alcune osservazioni

- Output generato (dipendere comunque dallo schedulatore):

Thread-0: $1 * 1 = 1$

Thread-9: $10 * 1 = 10$

Thread-5: $6 * 1 = 6$

Thread-8: $9 * 1 = 9$

Thread-7: $8 * 1 = 8$

Thread-6: $7 * 1 = 7$

Avviato Calcolo Tabelline

Thread-4: $5 * 1 = 5$

Thread-2: $3 * 1 = 3$

- da notare: il messaggio **Avviato Calcolo Tabelline** è stato visualizzato prima che tutti i threads completino la loro esecuzione. Perché?
 - il controllo ripassa al programma principale, dopo la attivazione dei threads e prima della loro terminazione.

Metodo start() verso run()

```
public class CalculatorMain {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; i++){  
            Calculator calculator=new Calculator(i);  
            Thread thread=new Thread(calculator);  
            thread.run();  
        }  
        System.out.println("Avviato Calcolo Tabelline")  
    }  
}
```

Output generato

```
main: 1 * 1 = 1  
main: 1 * 2 = 2  
main: 1 * 3 = 3  
.....  
main: 2 * 1 = 2  
.....
```

Avviato Calcolo Tabelline

start e run

- cosa accade se sostituisco l'invocazione del metodo run alla start?
 - non viene attivato alcun thread
 - il metodo run() viene eseguito all'interno del flusso del thread attivato per l'esecuzione del programma principale
 - flusso di esecuzione sequenziale
- il messaggio **“Avviato Calcolo Tabelline”** viene visualizzato dopo l'esecuzione di tutti i metodi metodo run() quando il controllo torna al programma principale
- solo il metodo start() comporta la creazione di un nuovo thread()!

Attivazione di threads metodo 1: riepilogo

Per definire tasks ed attivare threads che li eseguano

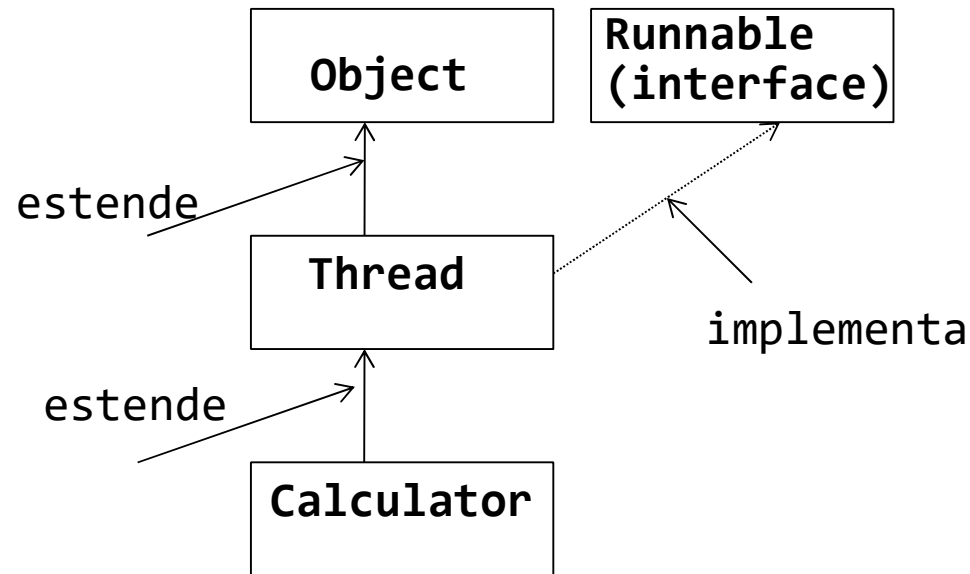
1. definire una classe R che implementi l'interfaccia **Runnable**, cioè implementare il metodo **run**. In questo modo si definisce un oggetto runnable, cioè un **task** che può essere eseguito
2. creare un'istanza T di R
3. Per costruire il thread, utilizzare il costruttore
public Thread (Runnable target)
passando il task T come parametro
4. attivare il thread con una **start()**

Metodo 2 di creazione
thread e Classe Thread

Creazione/attivazione threads: metodo 2

- utilizzare funzionalità dei linguaggi ad oggetti: subclassing, overriding
- creare una classe C che estenda la classe **Thread** ed effettuare l'**overriding** del metodo **run()** definito di quella classe
 - reminder: l'overriding
 - definire un metodo in una sottoclasse con lo stesso nome e segnatura del metodo della superclasse
 - a run-time si decide quale metodo viene invocato a seconda dell'oggetto su cui il metodo viene invocato
- istanziare un oggetto di quella classe: questo oggetto è un thread il cui comportamento è quello definito nel metodo run in cui è stato fatto l'overriding
- invocare il metodo **start()** sull'oggetto istanziato.

La gerarchia delle classi: overriding



- overriding del metodo `run()` all'interno della classe che estende `Thread()`
 - overriding di `run()` all'interno della classe `Calculator`
 - comportamento del thread definito dal metodo `run()` di `Calculator`

Creazione/attivazione threads: metodo 2

```
public class Calculator extends Thread {  
    .....  
    public void run() {  
        for (int i=1; i<=10; i++)  
            {System.out.printf("%s: %d * %d = %d\n",  
  
Thread.currentThread().getName(),number,i,i*number);}}}  
  
public class Main {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; i++){  
            Calculator calculator=new Calculator(i);  
            calculator.start();  
            System.out.println("Avviato Calcolo Tabelline"); } }
```

Quale alternativa utilizzare?

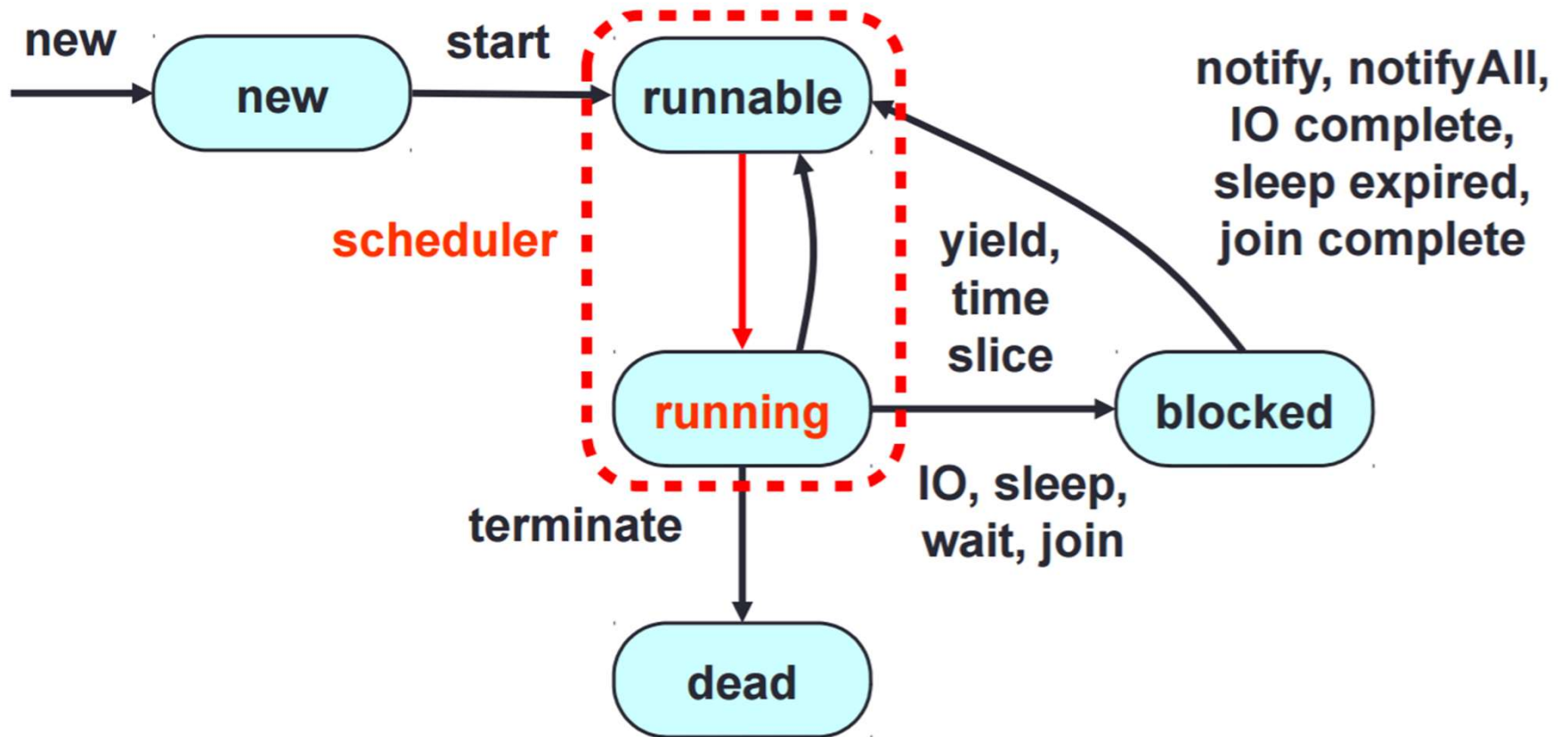
- in JAVA una classe può estendere una sola altra classe (**eredità singola**)
 - se si estende la classe Thread, la classe i cui oggetti devono essere eseguiti come thread non può estendere altre classi.
- questo può risultare svantaggioso in diverse situazioni, ad esempio:
 - gestione di eventi dell'interfaccia (movimento mouse, tastiera...)
 - la classe che gestisce un evento deve estendere una classe predefinita C di JAVA.
 - se il gestore deve essere eseguito in un thread separato, occorrerebbe definire una classe che estenda sia C che Thread, ma questo non è permesso in JAVA, occorrerebbe l'ereditarietà multipla
- si definisce allora una classe che :
 - estenda C (non può estendere contemporaneamente Thread)
 - implementi l'interfaccia Runnable

Attivazione di threads

Il metodo **start()**

- segnala allo schedulatore (tramite la JVM) che il thread può essere attivato (invoca un metodo nativo). L'ambiente del thread viene inizializzato
- restituisce immediatamente il controllo al chiamante, senza attendere che il thread attivato inizi la sua esecuzione.
- la stampa del messaggio “Avviato Calcolo Tabelline” può precedere quelle effettuate dai threads. Questo significa che il controllo è stato restituito al thread chiamante (il thread associato al main) prima che sia iniziata l'esecuzione dei threads attivati

Stati di un thread



Stati di un thread

Created/New (New Thread)

subito dopo l'istruzione **new** le variabili sono state allocate e inizializzate; il thread è in attesa di passare allo stato di eseguibile

Runnable/Running

thread è in esecuzione, o in coda d'attesa per ottenere l'utilizzo della CPU. (PS le specifiche java non separano i due stati, in realtà, un thread running è nello stato Runnable..)

Not Runnable (Blocked/Waiting)

il thread non può essere messo in esecuzione dallo scheduler. Entra in questo stato quando è in **attesa** di un'operazione di I/O, o dopo l'invocazione di metodi quali ad es. **wait()**, **sleep()**

Dead

al termine "naturale" della sua esecuzione o dopo l'invocazione del suo metodo **stop()** da parte di un altro thread (N.B. metodo deprecato)

Classe Thread

La classe **java.lang.Thread** contiene metodi per:

- **costruire** un thread interagendo con il sistema operativo ospite
- **attivare, sospendere, interrompere** i threads
- **non contiene i metodi per la sincronizzazione** tra i thread. Questi metodi sono definiti in `java.lang.Object`, perché la sincronizzazione opera su oggetti

Costruttori:

- diversi costruttori che differiscono per i parametri utilizzati
- Ad es. nome del thread, gruppo a cui appartiene il thread,...(vedere le API ad es. <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>)

Metodi

- possono essere utilizzati per interrompere, sospendere un thread, attendere la terminazione di un thread + un insieme di metodi set e get per impostare e reperire le caratteristiche di un thread
- esempio: assegnare nomi e priorità ai thread

Classe Thread

La classe `java.lang.Thread` contiene metodi per

- porre un thread allo stato blocked:

`public static native void sleep (long M)` sospende l'esecuzione del thread, per M millisecondi.

- durante l'intervallo di tempo relativo alla sleep, il thread può essere interrotto
- metodo statico: non può essere invocato su una istanza di un thread

sleep(long time) blocca per il tempo specificato in time l'esecuzione di un thread. Nessun lock in possesso del thread viene rilasciato.

Alcuni metodi per il controllo di thread

start() fa partire l'esecuzione di un thread. La macchina virtuale Java invoca il metodo `run()` del thread appena creato

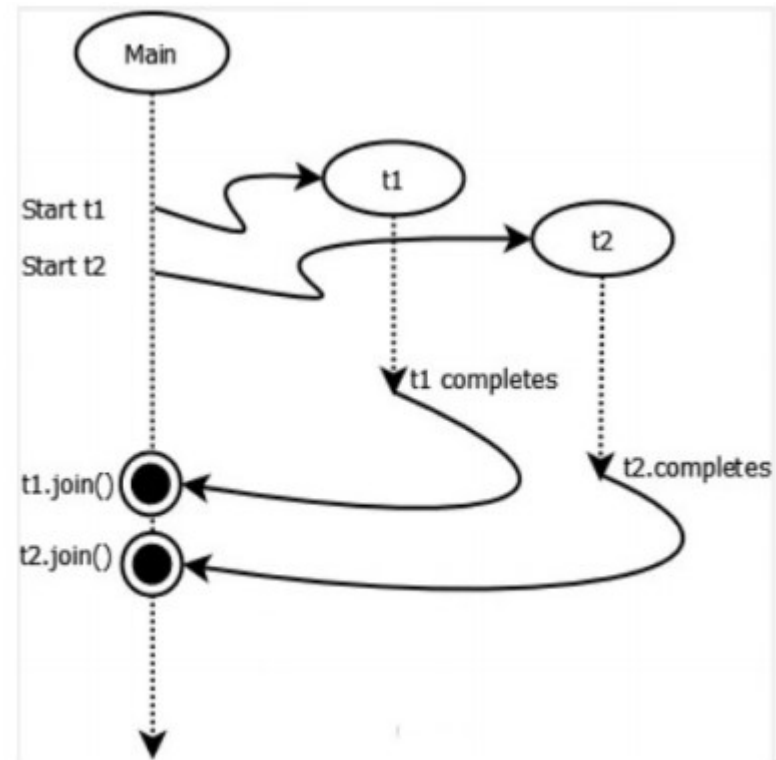
yield(): può mettere in pausa l'esecuzione del thread invocante per lasciare l'uso della CPU agli altri thread, con la stessa priorità, in coda d'attesa. «A hint to the scheduler that the current thread is willing to yield its current use of a processor.»

`stop()`, `suspend()`, `resume()` **DEPRECATED**

join() blocca il thread chiamante in attesa della terminazione del thread su cui si invoca il metodo. Anche con timeout – vedi slide successiva

Il metodo join

- il metodo **join()** della classe Thread():
 - il thread che invoca join() su una istanza di Thread **t** si sospende e attende la terminazione dell'istanza di thread su cui ha invocato il metodo join()
- possibile specificare il **timeout di attesa**
 - Se entro l'intervallo specificato il thread non termina, il metodo ritorna comunque
- join() può lanciare una InterruptedException, se il thread sospeso sulla join() riceve una interruzione.



Join - esempio

```
public class Exercise1_join {  
    public static void main(String[] args) {  
        double x = 90;  
        double y = 0;  
        double z = 180;  
        double result;  
  
        MathOperation synOp = new MathOperation("syn", x);  
        Thread synThread = new Thread(synOp);  
        synThread.start();  
  
        MathOperation cosOp = new MathOperation("cos", y);  
        Thread cosThread = new Thread(cosOp);  
        cosThread.start();  
  
        ...  
    }  
}
```

Join esempio

...

```
try {  
    synThread.join();  
    cosThread.join();  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
  
result = synOp.getResult()+cosOp.getResult();  
System.out.println("Final result is " + result);  
}  
}
```

- completare il programma precedente specificando il codice del task MathOperation
- provare ad eseguire il programma e verificare il corretto funzionamento

Gestire le interruzioni

- JAVA mette a disposizione un meccanismo per interrompere un thread e diversi meccanismi per intercettare l'interruzione
 - dipendenti dallo stato in cui si trova un thread (runnable, blocked)
- Metodo *interrupt()*: non forza la terminazione del thread, permette di inviare una richiesta di terminazione del thread
- un thread può essere interrotto e può intercettare l'interruzione in modi diversi, a seconda dello stato in cui si trova
 - se è **blocked** (es. sleep) l'interruzione causa il sollevamento di una **InterruptedException**
 - se è **in esecuzione**, si può testare un flag che segnala se è stata inviata una interruzione.

Interrompere un Thread

- implementazione del metodo **interrupt()**: imposta a true una variabile **booleana (flag)** nel descrittore del thread
- il flag vale true se esistono interruzioni pendenti
- è possibile testare il valore del flag mediante:
 - **public static boolean** interrupted(): metodo statico, si invoca con il metodo Thread.interrupted(), restituisce un valore booleano che segnala se il thread ha ricevuto un'interruzione **e riporta il valore del flag a false**
 - **public boolean** isInterrupted()
 - deve essere invocato su un'istanza di un oggetto di tipo Thread
 - restituisce un valore booleano che segnala se il thread ha ricevuto un'interruzione
 - non cambia il valore del flag a false

Gestione di interruzioni

Intercettare una interruzione quando il thread si sospende

```
public class SleepInterrupt implements Runnable {  
    public void run ( ) {  
        try{  
            System.out.println("dormo per 20 secondi");  
            Thread.sleep(20000);  
            System.out.println ("svegliato");}  
        catch (InterruptedException x ){  
            System.out.println("interrotto");  
            return;  
        }  
        System.out.println("esco normalmente");  
    }  
}
```

Se in un istante compreso tra l'inizio e la fine della sleep (inizio e fine inclusi), il flag "interrupted" ha valore vero, allora l'eccezione viene lanciata

Gestione delle interruzioni

```
public class SleepMain {  
    public static void main (String args [ ])    {  
        SleepInterrupt si = new SleepInterrupt();  
        Thread t = new Thread (si);  
        t.start ( );  
        try  
            {Thread.sleep(2000);}  
        catch (InterruptedException x) {    };  
        System.out.println("Interrompo l'altro thread");  
        t.interrupt( );  
        System.out.println ("sto terminando..."); } }
```

In pratica...

- Affinché il meccanismo di interruzioni funzioni correttamente, il thread interrotto deve supportare la sua interruzione!
- Se la richiesta di interruzione arriva quando il thread è nello stato Blocked (es. sleep), viene lanciata una InterruptedException e il flag settato a false!
- Se il thread non è nello stato Blocked, è **necessario controllare lo stato del flag**

```
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]);  
}
```

```
public class ThreadInterrupt implements Runnable {  
    public void run ()  
    {  
        while (!Thread.interrupted())  
  
            System.out.println(Thread.currentThread().get  
            Name() + " in esecuzione");  
    }  
}
```


Thread demoni

- i thread demoni hanno il compito di fornire un servizio in background fino a che il programma è in esecuzione
 - non sono intesi come parte fondamentale del programma
 - bassa priorità
- quando tutti i thread non-demoni sono completati, il programma termina (anche se ci sono thread demoni in esecuzione). Quando gli user threads sono completati (incluso il main), la JVM termina il programma e i thread demoni associati
- se ci sono thread non-demoni ancora in esecuzione, il programma non termina
- un esempio di thread non-demone è il `main()`

Thread demoni

```
public class SimpleDaemons implements Runnable {  
    public SimpleDaemons() { }  
    public void run() {  
        while(true) {  
            try { Thread.sleep(100); }  
                catch (InterruptedException e) {...}  
            System.out.println("termino");  
        }  
    }  
    public static void main(String[] args) {  
        for(int i = 0; i < 10; i++){  
            SimpleDaemons sd=new SimpleDaemons();  
            Thread t= new Thread (sd);  
            t.setDaemon(true);  
            t.start();}  
        }  
    }  
}
```

Perchè non viene
stampato alcun
messaggio "termino"?

Terminazione di programmi concorrenti

Per determinare la terminazione di un programma JAVA:

- un programma JAVA termina quando terminano tutti i threads non demoni che lo compongono
- se il thread iniziale, cioè quello che esegue il metodo `main()` termina, i restanti thread ancora attivi continuano la loro esecuzione, fino alla loro terminazione.
- se uno dei thread usa l'istruzione `System.exit()` per terminare l'esecuzione, allora tutti i threads terminano la loro esecuzione **DA EVITARE**

Analizzare le proprietà di un thread

- La classe Thread salva alcune informazioni che aiutano l'identificazione dei thread
 - **ID**: identificatore del thread
 - **nome**: nome del thread
 - **priorità**: valore da 1 a 10 (1 priorità più bassa).
 - **nome gruppo**: gruppo a cui appartiene il thread
 - **stato**: uno dei possibili stati: **new**, **runnable**, **blocked**, **waiting**, **time waiting** o **terminated**.

- metodi setter e getter per reperire il valore di ogni proprietà.

```
public final void setName(String newName),  
public final String getName( )
```

consentono, rispettivamente, di associare un nome ad un thread e di reperirlo

Analizzare le proprietà di un thread

```
public class CurrentThread {  
    public static void main(String args[]) {  
        Thread current = Thread.currentThread();  
        System.out.println("ID: " + current.getId());  
        System.out.println("NOME: " + current.getName());  
        System.out.println("PRIORITA: " +  
            current.getPriority());  
    }  
}
```

ID: 1

NOME: main

PRIORITA': 5

`Thread.currentThread()` restituisce un riferimento al thread che sta eseguendo il frammento di codice (nell'esempio, il thread è quello associato al main)

Analizzare le proprietà di un thread

```
public static void main(String[] args) throws Exception
{
    Thread threads[]=new Thread[10];
    Thread.State status[]=new Thread.State[10];
    for (int i=0; i<10; i++){
        threads[i]=new Thread(new Calculator(i));
        if ((i%2)==0){
            threads[i].setPriority(Thread.MAX_PRIORITY);
        } else {
            threads[i].setPriority(Thread.MIN_PRIORITY);
        }
        threads[i].setName("Thread "+i);
    }
    ...continua...
```

Analizzare le proprietà di un thread

```
FileWriter file = new FileWriter("log.txt");
PrintWriter pw  = new PrintWriter(file);
pw.printf("*****\n");
for (int i=0; i<10; i++){
    pw.println("Status of

    Thread"+i+": "+threads[i].getState());
    status[i]=threads[i].getState();
}
for (int i=0; i<10; i++){
    threads[i].start();
}

...continua...
```

Analizzare le proprietà di un thread

```
boolean finish=false;
while (!finish) {
    for (int i=0; i<10; i++){
        if (threads[i].getState()!=status[i]) {
            pw.printf("Id %d - s\n",threads[i].getId(),threads[i].getName())
            pw.printf("Priority: %d\n",threads[i].getPriority());
            pw.printf("Old State: %s\n",status[i]);
            pw.printf("New State: %s\n",threads[i].getState());
            pw.printf("*****\n");
            pw.flush();
            status[i]=threads[i].getState();}}
        boolean completed = true;
        for (int i=0; i<10; i++)
            completed= completed &&(threads[i].getState()==
                Thread.State.TERMINATED);
        finish=completed;    }}}
```


Un esempio di esecuzione

```
Status of Thread 0 : NEW
Status of Thread 1 : NEW
Status of Thread 2 : NEW
Status of Thread 3 : NEW
Status of Thread 4 : NEW
Status of Thread 5 : NEW
Status of Thread 6 : NEW
Status of Thread 7 : NEW
Status of Thread 8 : NEW
Status of Thread 9 : NEW
```

```
*****
```

```
Id 10 - Thread 0
```

```
Priority: 10
```

```
Old State: NEW
```

```
New State: RUNNABLE
```

```
*****
```

```
Id 11 - Thread 1
```

```
Priority: 1
```

```
Old State: NEW
```

```
New State: RUNNABLE
```

```
*****
```

```
.....
```

- Dal diagramma si possono analizzare i cambiamenti di stato del thread
- I thread di priorità maggiore dovrebbero terminare prima degli altri

N.B. non ci sono garanzie, solo un'indicazione per lo scheduler, implementazione system-dependent