Laboratorio di Reti – A (matricole pari)

Autunno 2021,
instructor: Laura Ricci

laura.ricci@unipi.it

Lezione 1

JAVA multithreading: creazione, attivazione, terminazione, interruzione di threads.

14/9/2021

INFORMAZIONI UTILI

- **Docente**
  - *Laura Ricci* (laura.ricci@unipi.it),
- **Supporto alla Didattica**
  - *Matteo Loporchio*
- lezioni in aula e online sull'aula virtuale Teams (link sulla pagina Moodle)
- **Orario**

  martedì 11.00 -13.00 - presentazione concetti

  martedì 14.00 -16.00 - sperimentazione, quiz, correzione esercizi
- **Materiale del corso su Moodle:**

  https://elearning.di.unipi.it/course/view.php?id=196
  - slides
  - forum, chats...
  - quiz
  - assignments, progetto finale

INFORMAZIONI UTILI

- **Laboratorio**
  - verifica esercizi assegnati nelle lezioni teoriche
  - consegna degli esercizi entro 15 giorni dalla data di assegnazione.
    - se si consegna l'80% degli esercizi, sarà possibile discuterli all'esame e, se la discussione è positiva, ottenere un bonus di 2 punti.
  - quiz anonimi a risposta chiusa per l'autoverifica

MODALITA' DI ESAME

- l'esame di Reti e Laboratorio si svolge in due prove:
  - prova di Reti (Teoria)
  - prova di Laboratorio
- non ci sono vincoli di precedenza tra la prova di Reti e quella di Laboratorio.
- il voto di ciascuna prova ha validità per l'AA 2021/22 (entro l'appello straordinario di novembre 2022 compreso per chi ha i requisiti per partecipare all'appello).
- Voto finale:
  - media dei voti ottenuti nelle due prove (arrotondamento per eccesso).
  - nel calcolo della media gli esami con lode vengono valutati 32/30.

MODALITA' DI ESAME

- Tutte le prove d'esame prevedono obbligatoriamente l'iscrizione sul SISTEMA DI ISCRIZIONE DI ATENEO
  - chi non si iscrive entro i termini non può partecipare alla prova di esame
  - attenzione alle scadenze!!!
- Prova di Laboratorio
  - lo studente deve consegnare un progetto, da svolgere secondo le specifiche consegnate durante il corso (entro la prima metà di dicembre).
  - le specifiche del progetto sono valide fino all'appello straordinario di novembre 2022 (a questo appello può accedere solo chi ha i requisiti).
  - la prova consiste in un colloquio orale che include la discussione del progetto e verifica dell'apprendimento dei concetti e contenuti presentati a lezione.
  - il progetto deve essere svolto individualmente

INFORMAZIONI UTILI: PREREQUISITI

- corso di Programmazione 2, conoscenza del linguaggio JAVA. In particolare:
  - packages
  - gestione delle eccezioni
  - collezioni
  - generics
- dal modulo teorico di reti: conoscenza protocolli TCP/IP
- linguaggio di programmazione di riferimento: anche se l'ultima release è la 16, facciamo riferimento a JAVA 8
  - concorrenza: costrutti base, JAVA.UTIL.CONCURRENT
  - JAVA.NIO
  - collezioni
  - rete: JAVA.NET, JAVA.RMI
- ambiente di sviluppo di riferimento: Eclipse

INFORMAZIONI UTILI

- Materiale Didattico:
  - lucidi delle lezioni
  - testi consigliati (non obbligatori) per la parte relativa ai threads
    - *Bruce Eckel*, Thinking in JAVA - Volume 3 - Concorrenza e Interfacce Grafiche
    - *B. Goetz*, JAVA Concurrency in Practice, 2006
  - Testi consigliati (non obbligatori) per la parte relativa alla programmazione di rete
    - *Dario Maggiorini*, Introduzione alla Programmazione Client Server, Pearson
    - *Esmond Pitt*, Fundamental Networking in JAVA
- Materiale di Consultazione:
  - *Harold*, JAVA Network Programming 3nd 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

PROGRAMMA PRELIMINARE DEL CORSO

Threads

- creazione ed attivazione di threads,
  - meccanismi di gestione di pools di threads, Callable: threads che restituiscono risultati, interruzioni
- 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

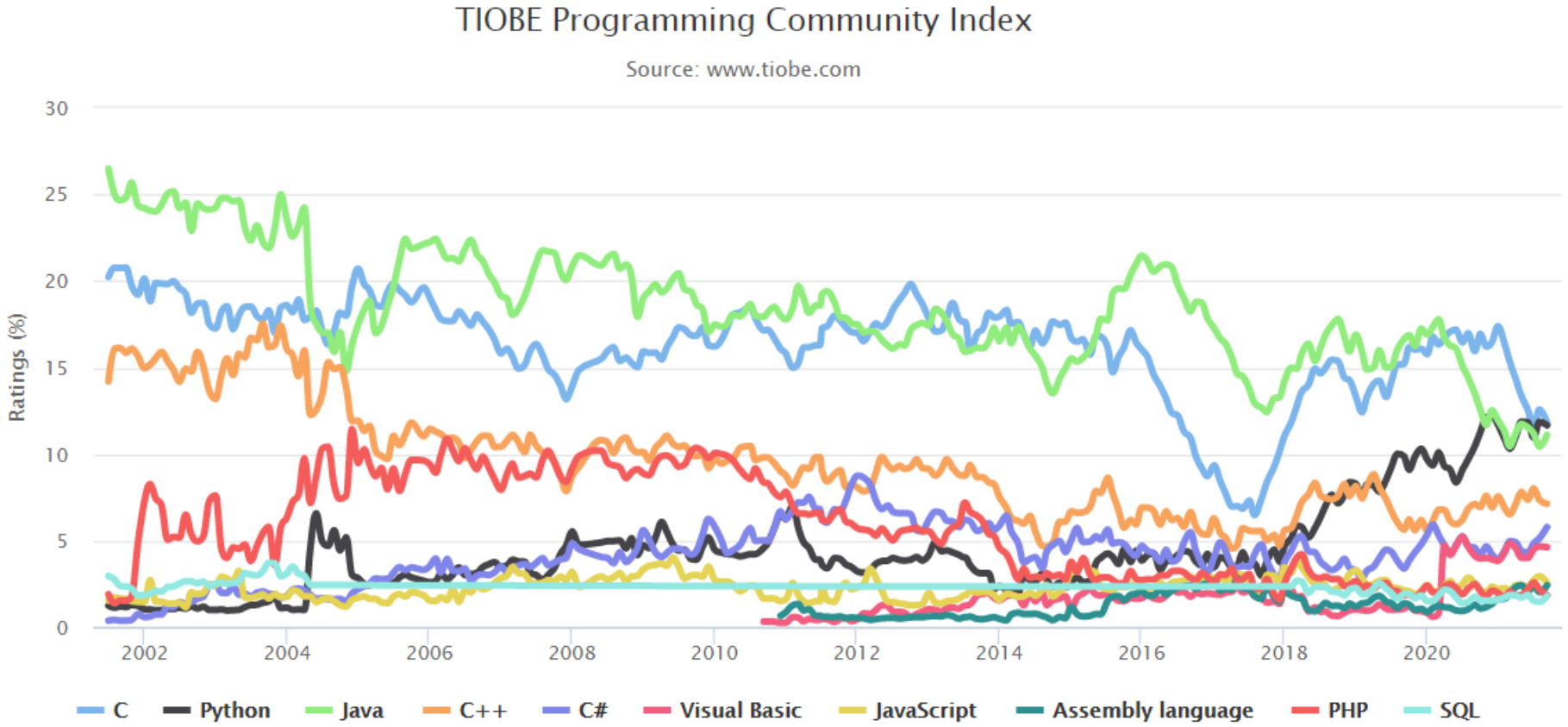
- streams: tipi di streams, composizione di streams
- meccanismi di serializzazione
  - serializzazione standard di JAVA: problemi
  - JSON

continua....

PROGRAMMA PRELIMINARE DEL CORSO

- NewIO
  - Channels, buffers, memory mapped IO
- Programmazione di rete a basso livello
  - connection oriented Sockets
  - connectionless sockets: UDP, multicast
- NewIO e sockets
  - Selector: channel multiplexing
- Oggetti Distribuiti
  - definizione di oggetti remoti
  - il meccanismo di Remote Method Invocation (RMI)
  - dynamic code loading
  - problemi di sicurezza
  - il meccanismo delle callbacks

L'INDICE TIOBE DEI LINGUAGGI DI PROGRAMMAZIONE



- 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!

L'EVOLUZIONE DI JAVA

1995 - JDK BETA

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

23 JAN 1996 - JAVA 1

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

19 FEB 1997 - JAVA 1.1

Inner Classes, Java Beans, JDBC, RMI

8 DEC 1998 - JAVA 1.2

Swing, JIT Compiler, Collections

8 MAY 2000 - JAVA 1.3

HotSpot JVM, JNDI, JPDA

6 FEB 2002 - JAVA 1.4

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

30 SEP 2004 - JAVA 5

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

L'EVOLUZIONE DI JAVA

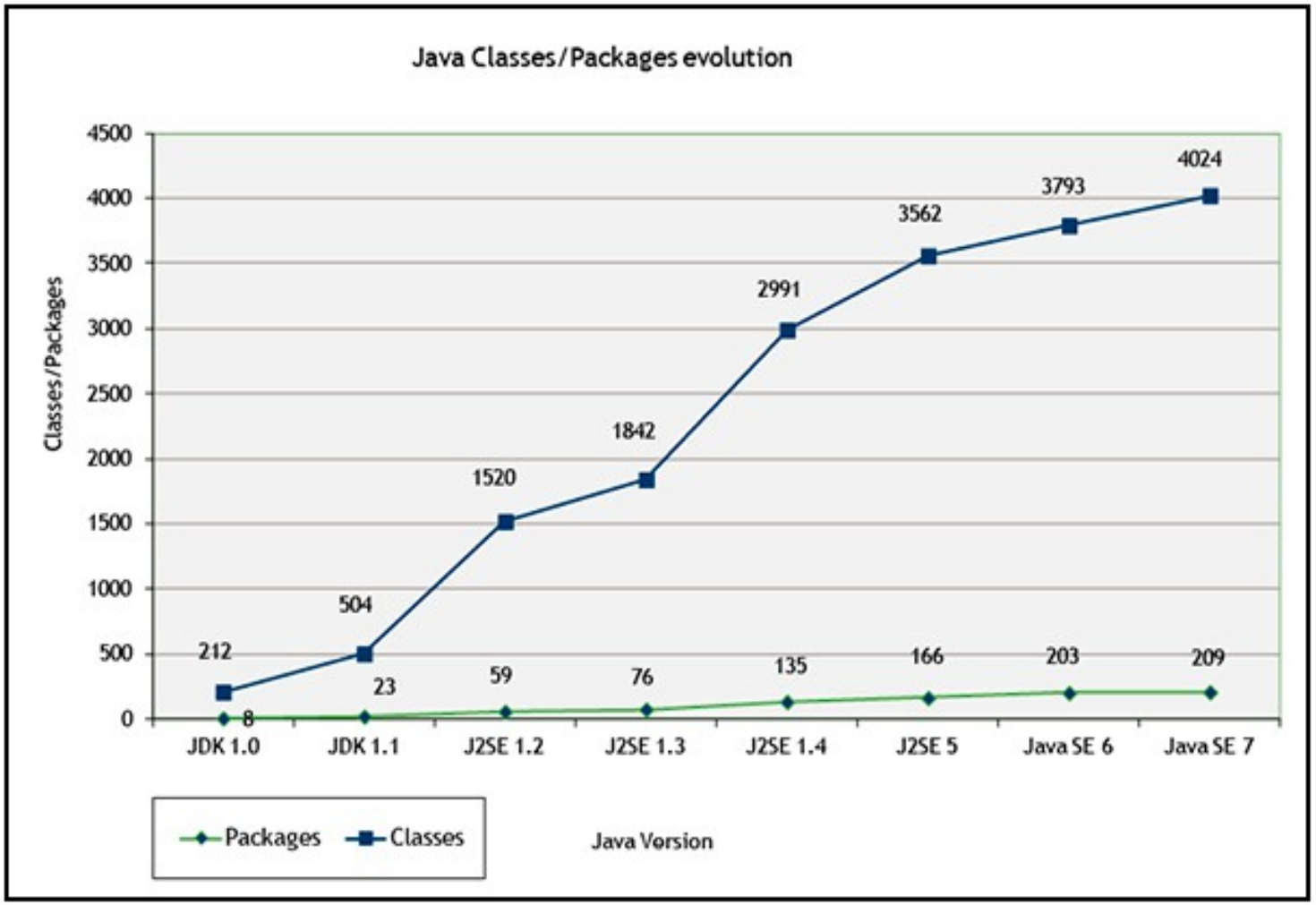
Ultima versione JAVA 16

In questo corso faremo riferimento a JAVA8

EVOLUZIONE: CLASSI BLU IN QUESTO CORSO

- 1.0.2 prima versione stabile, rilasciata il 23 gennaio del 1996
  - AWT Abstract Window Tollkit, 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
- 5: una vera rivoluzione generics, concorrenza,....
- 7: acquisizione da parte di Oracle: framework fork and join
- 8: Lambda Expressions

AUMENTO NUMERO DELLE CLASSI



MULTITHREADING: PERCHE'?

gli utenti (sia che usino un computer, un tablet, un mobile) possono interagire simultaneamente con diverse applicazioni

- scrivere un documento in word
- ascoltare musica
- postare su un social network

- un processo attivato per ogni applicazione

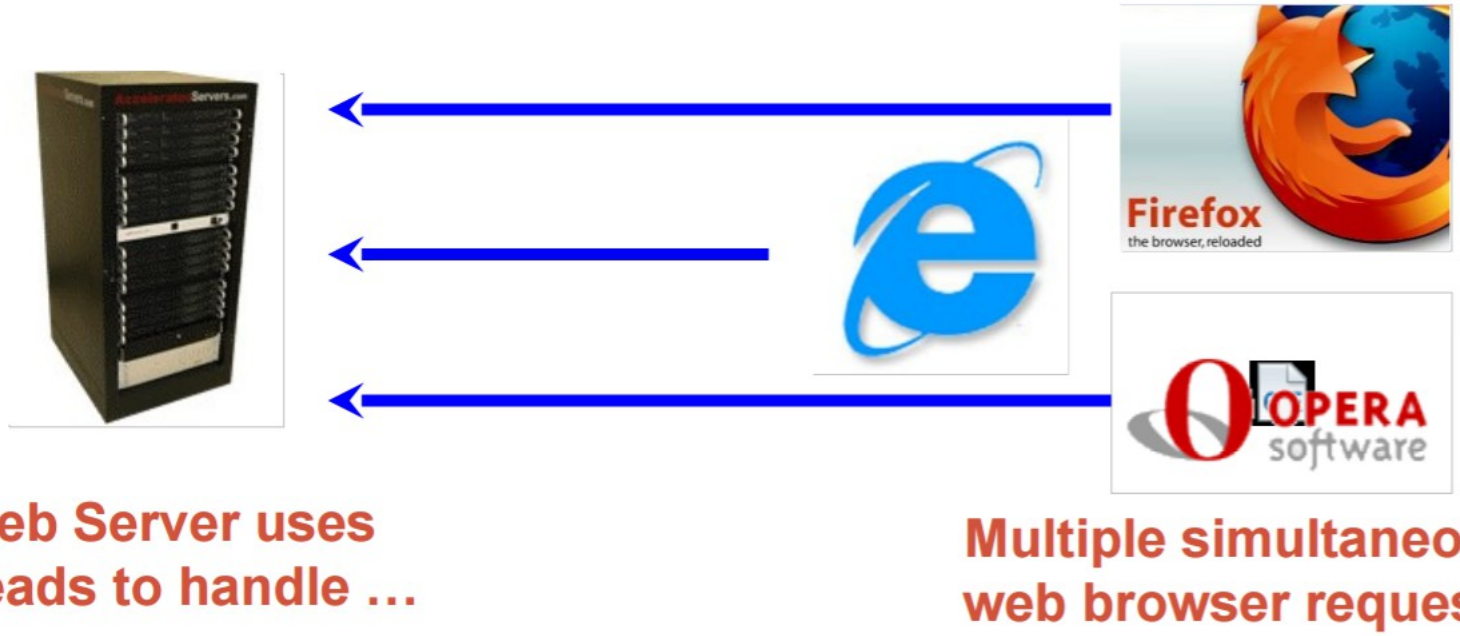
ma anche una stessa applicazione può eseguire diversi task simultaneamente

- nel word processor
  - salvare un documento mentre si evidenzia un testo in neretto
- nel browser
  - caricare dati dalla rete, mentre si salvano dati su un file, e viene eseguita la computazione per animare una gif.

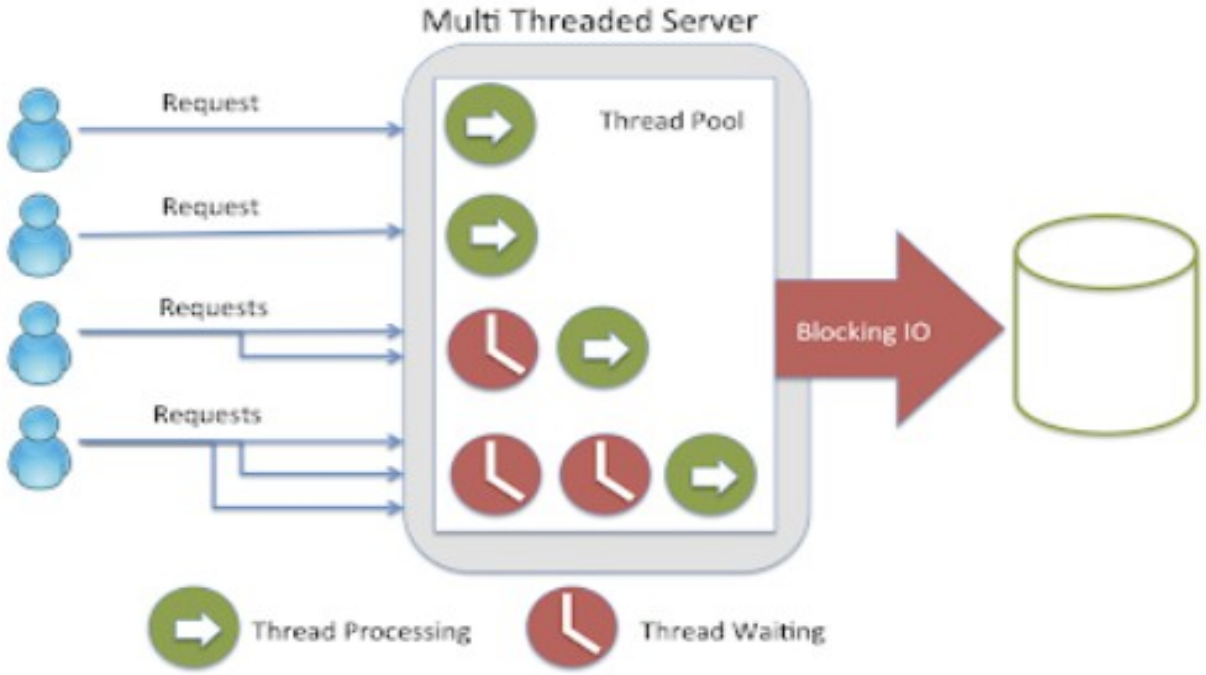
- un thread attivato per ogni task

SERVIRE PIU' CLIENT CONTEMPORANEAMENTE

- applicazioni client server
  - più client serviti simultaneamente
  - un client non deve aspettare che il server termini di elaborare la richiesta del client precedente



SERVIRE PIU' CLIENT CONTEMPORANEAMENTE



- il throughput dell'applicazione può essere incrementato se client diversi sono serviti da thread diversi, ma solo fino ad un certo limite
- oltre quel limite, i thread iniziano a competere per la CPU e il costo del cambio di contesto supera il beneficio del multithreading
- sfrutteremo il meccanismo del threadpooling per limitare questo fenomeno

MULTITHREADING: PERCHE'?

- migliore utilizzazione delle risorse
  - quando un thread è sospeso, altri thread vengono mandati in esecuzione
  - riduzione del tempo complessivo di esecuzione
- migliore performance per applicazioni computationally intensive
  - 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,...

JAVA UTIL.CONCURRENT FRAMEWORK

- JAVA < 5 built in for concurrency: lock implicite, wait, notify e poco più.
- `JAVA.util.concurrency`: lo scopo è lo stesso del framework java.util.Collections: 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.

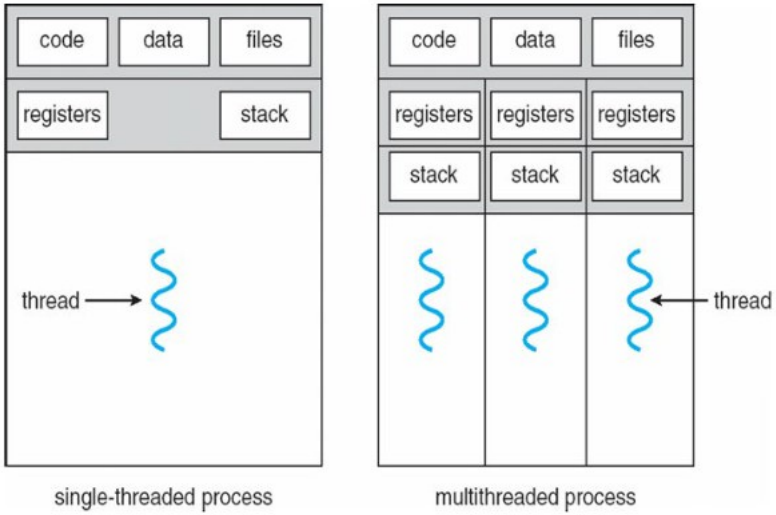
- tra i package principali:
  - java.util.concurrent
    - executor, concurrent collections, semaphores,...
  - java.util.concurrent.atomic
    - AtomicBoolean, AtomicInteger,...
  - java.util.concurrent.locks
    - Condition
    - Lock
    - ReadWriteLock

JAVA 5 CONCURRENCY 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

THREAD: DEFINIZIONE

- processo: programma in esecuzione
  - due diverse applicazioni, ad esempio MS Word, MS Access, sono eseguite da processi diversi.
- thread (light weight process): un flusso di esecuzione all'interno 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

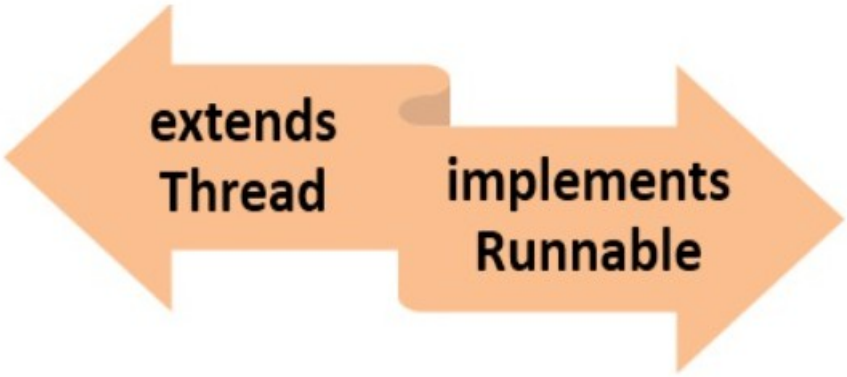
PROCESSI E THREADS

- thread multitasking verso process multitasking:
  - i thread condividono lo stesso spazio degli indirizzi
  - meno costosi
    - il cambiamento di contesto tra thread
    - la comunicazione tra thread

- esecuzione dei thread:
  - single core: multiplexing, interleaving (meccanismi di time sharing,...)
  - multicore: più flussi in esecuzione eseguiti in parallelo, simultaneità di esecuzione

JAVA: CREAZIONE ED ATTIVAZIONE DI THREAD

- quando si manda in esecuzione un programma JAVA
  - la JVM crea un thread che invoca il metodo main del programma
  - esiste sempre almeno un thread per ogni programma, il main
- in seguito...
  - altri thread sono attivati automaticamente da JAVA (gestore eventi, interfaccia, garbage collector,...).
  - ogni thread durante la sua esecuzione può creare ed attivare altri threads.
- come creare ed attivare esplicitamente un thread? Due modalità



CREAZIONE ED ATTIVAZIONE DI THREAD

primo metodo:

- definire un task
- creare un oggetto thread e passargli il task definito, da eseguire
- attivare il thread con una `start()`

per definire un task

- definire una classe che implementi l'interfaccia `Runnable`
- creare un'istanza R di questa classe, Questo è il task da passare al thread

DEFINIRE ED ESEGUIRE TASK

```
public class ThreadRunnable {
    public static class MyRunnable implements Runnable {
    public void run() {
        System.out.println("MyRunnable running");
        System.out.println("MyRunnable finished");
    }
  }

public static void main(String [] args) {
   Thread thread = new Thread (new MyRunnable());
   thread.start();
   }
 }
```

Stampa:

MyRunnable running

MyRunnable finished

L' INTERFACCIA RUNNABLE

- appartiene al package `java.language`
- contiene solo la segnatura del metodo `void run()`, che deve essere implementato
- un'istanza della classe che implementa `Runnable` è un task
  - un fragmento di codice che può essere eseguito in un thread
  - la creazione del task non implica la creazione di un thread per lo esegua.
  - lo stesso task può essere eseguito da più threads: un solo codice, più esecutori
  - il task viene passato al Thread che deve eseguirlo

TASK DEFINITO CON CLASSE ANONIMA

```
public class RunnableAnonymous {
  public static void main (String[] args) {
        Runnable runnable = new Runnable () {
          public void run() {
           System.out.println("Runnable running");
           System.out.println("Runnable finished");
           }
       };

       Thread thread = new Thread (runnable);
       thread.start();
   }}
  }
```

Stampa:

Runnable running

Runnable finished

SOLUZIONE 2: ESTENDERE THREAD

- creare una classe C che estenda `Thread`
- effettuare l'overriding del metodo `run()`
- istanziare un oggetto di quella classe
  - questo oggetto è un thread il cui comportamento è quello definito nel metodo run ridefinito
- invocare il metodo `start()` sull'oggetto istanziato.

Overriding:

- metodo in una sottoclasse con lo stesso nome e segnatura del metodo della superclasse
- decidere a run-time quale metodo viene invocare in base all'istanza su cui si invoca il metodo

SOLUZIONE 2: ESTENDERE THREAD

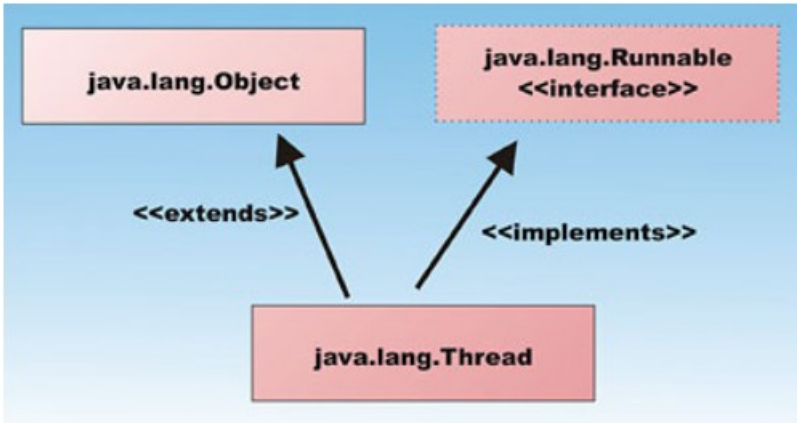
```
public class ExtendingThread {

    public static class MyThread extends Thread {
        public void run() {
            System.out.println("MyThread running");
            System.out.println("MyThread finished");
            }
         }
   public static void main (String [] args) {
        MyThread myThread = new MyThread();
        myThread.start();
     }
   }
```

Stampa

MyThread running

MyThread finished

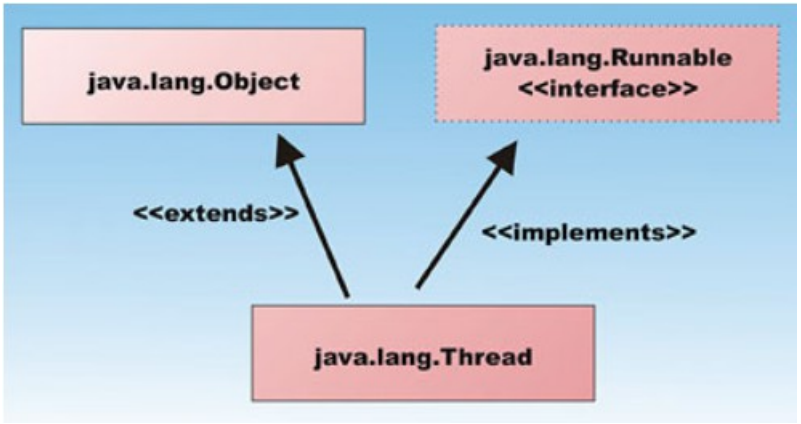
LA CLASSE THREAD



- memorizza un riferimento all'oggetto Runnable, eventualmente passato come parametro, nella variabile `runnable`
- definisce il metodo `run( )` come segue

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

LA CLASSE THREAD



- quando viene invocata la `start()`

  se il metodo `run()` è stato ridefinito mediante overriding (soluzione 2)

  si invoca il metrdo run() più specifico, che è quello definito dal programmatore
- altrimenti, si esegue il metodo `run()` predefinito nella classe Thread, (soluzione 1)
  - se la variable runnable è diversa da nil, questo metodo, a sua volta, invoca il metodo `run()` dell'oggetto `Runnable` passato
  - si esegue il metodo definito dal programmatore

ATTIVARE UN INSIEME DI THREAD

- scrivere un programma che stampi 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

IL 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);
          }}}
```

- NOTA: `public static native Thread currentThread ( )`:
  - 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 fragmento di codice

IL MAIN PROGRAM

```
public class Main {
    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 effettuta, un esempio è il seguente:

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

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. Perchè?
  - il controllo ripassa al programma principale, dopo la attivazione dei threads e prima della loro terminazione.

START( ) E RUN( )

```
public class Main {
    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(); // questa versione del programma è errata
              System.out.println("Avviato Calcolo Tabelline"} }
```

Output generato

```
main: 1 * 1 = 1
main: 1 * 2 = 2
main: 1 * 3 = 3
 ……..
main: 2 * 1 = 2
main: 2 * 2 = 4
……..
Avviato Calcolo Tabelline
```

START E RUN

cosa accade se sostituisco l'invocazione del metodo run alla start?

- non viene attivato alcun thread
- ogni 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()!

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" precede 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

TASK CALCULATOR CON 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 solo 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 C predefinita 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 la interfaccia Runnable

TERMINAZIONE DI PROGRAMMI MULTITHREADED

```
public class DemonExample {
   public static void main (String [] args) {
         Runnable runnable = new Runnable () {
            public void run()
               {while (true) {
                  sleep(1000);
                  System.out.println("Running");
                 } }   };
         Thread thread = new Thread(runnable);
         thread.start();
          }
        sleep(3100); }
public static void sleep(long millis) {
      try {
           Thread.sleep(millis);
         } catch (InterruptedException e) {e.printStackTrace(); }}}
```

Stampa

Running, Running, Running,...
all'infinito

TERMINAZIONE DI PROGRAMMI MULTITHREADED

```
public class DemonExample {
   public static void main (String [] args) {
       Runnable runnable = new Runnable () {
       public void run()
         {while (true) {
              sleep(1000);
              System.out.println("Running");
                } }   };
        Thread thread = new Thread(runnable); thread.setDaemon(true);
        thread.start();
         }
        sleep(3100); }
public static void sleep(long millis) {
       try {
            Thread.sleep(millis);
           } catch (InterruptedException e) {e.printStackTrace(); }}}
```

Stampa

Running, Running, Running.
poi termina

THREAD DEMONI

- threads a bassa priorità
  - adatti per jobs non-critici da eseguire in background
  - servizi di background utili fino a che il programma è in esecuzione, generalmente creati dalla JVM, ad esempio per garbage collection
  - ma anche l'utente può dichiarare che un thread è un demone: ad esempio un thread che offre un servizio di timing
- non appena tutti i thread non demoni del programma sono terminati
  - la JVM termina il programma
  - forza la terminazione dei thread demoni
- il `main()` è un thread non demone!

TERMINAZIONE DI PROGRAMMI CONCORRENTI

- 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 e non demoni continuano la loro esecuzione, il programma termina quando anche questi terminano.
  - il "quadratino" rosso di Eclipse rimane "rosso" anche se il main è terminato
- se uno dei thread usa l'istruzione `System.exit()`per terminare l'esecuzione, allora tutti i threads terminano la loro esecuzione

GESTIRE LE INTERRUZIONI

JAVA mette a disposizione

- un meccanismo per interrompere un thread
- diversi meccanismi per intercettare l'interruzione
  - dipendenti dallo stato in cui si trova un thread, running, blocked
  - se il thread è sospeso l'interruzione solleva una `InterruptedException`
  - se è in esecuzione, può testare un flag che segnala se è stata inviata una interruzione.
- il thread decide comunque autonomamente come rispondere alla interruzione

GESTIONE DI INTERRUZIONI IN STATO DI SOSPESO

```
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");
            }

          }
```

- in un istante compreso tra l'inizio e la fine della sleep (inizio e fine inclusi), al thread arriva una interruzione
- 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..."); } }
```

INTERROMPERE UN THREAD

- il metodo interrupt( )
  - imposta a true un valore booleano nel descrittore del thread.
  - il flag vale true, se esistono interrupts pendenti

- per testare il valore del flag:
  - `public static boolean Interrupted ( )`

    metodo statico, si invoca con il nome della classe Thread.Interrupted( ))
  - `public boolean isInterrupted ( )`

    deve essere invocato su un'istanza di un oggetto di tipo thread
  - entrambi i metodi
    - restituiscono un valore booleano che segnala se il thread ha ricevuto un'interruzione
    - `interrupted()` rimette la flag a false, mentre `isInterrupted()` non cambia il valore

ATTENDERE LA TERMINAZIONE DI UN THREAD

```
public class JoinExample {
    public static void main (String [] args) {
        Runnable runnable = new Runnable () {
            public void run()
               {for (int i=0; i<5; i++){
                sleep(1000);
                System.out.println("Running");
               }}};
            Thread thread = new Thread(runnable);
            thread.setDaemon(true);
            thread.start();
    }
    public static void sleep(long millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) { Esecuzione
            e.printStackTrace(); }}}        Il programma termina immediatamante
```

ATTENDERE LA TERMINAZIONE DI UN THREAD

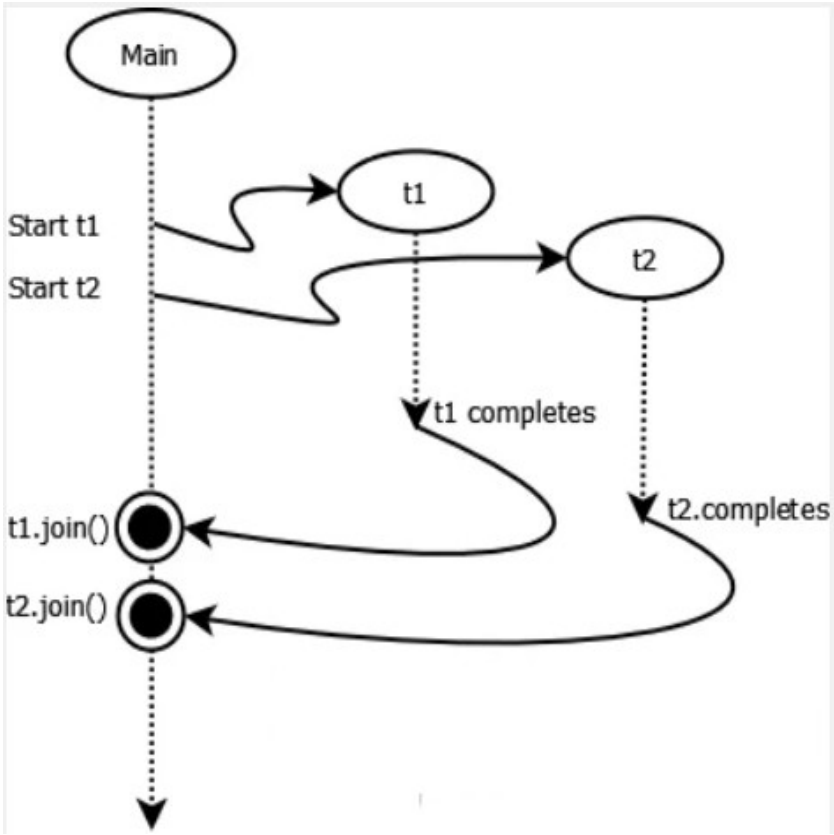
```
public class JoinExample {
    public static void main (String [] args) {
        Runnable runnable = new Runnable () {
            public void run()
               {for (int i=0; i<5; i++){
                sleep(1000);
                System.out.println("Running");
               }}};
            Thread thread = new Thread(runnable);
            thread.setDaemon(true);
            thread.start();
            try {
                thread.join();
            } catch (InterruptedException e) {e.printStackTrace();} }
    public static void sleep(long millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace(); }}}
```

Ora il programma stamapa 5 volte Running e poi termina

ASPETTARE LA TERMINAZIONE DI UN THREAD

`join()`

- `metodo` della classe `Thread()`, invocato sull'istanza t di un thread.
- il thread che lo esegue si sospende in attesa della terminazione di t.
- possibile specifica di un tempo massimo di attesa (timeout di attesa)
- se il thread sospeso sulla `join()` riceve un'interruzione
  - viene sollevata una eccezione
  - buona pratica mettere la join in una `try catch`



ASPETTARE LA TERMINAZIONE DI UN THREAD

```
import java.util.Arrays;import java.util.Collections;import java.util.List;

public class ThreadJoinExample {
    public static void main(String[] args) {
        Integer[] values = new Integer[] { 3, 1, 14, 3, 4, 5, 6, 7, 8, 9, 11,
                                           3, 2, 1 };
        Average avg = new Average(values);
        // Average is a task that implements Runnable
        Median median = new Median(values);
        // Median is a task that implements Runnable
        Thread t1 = new Thread(avg, "t1");
        Thread t2 = new Thread(median, "t2");
        System.out.println("Start the thread t1 to calculate average");
        t1.start();
        System.out.println("Start the thread t2 to calculate median");
        t2.start();
```

ASPETTARE LA TERMINAZIONE DI UN THREAD

```
try { System.out.println("Join on t1");
      t1.join();
      System.out.println("t1 has done with its job of calculating average");
      } catch (InterruptedException e) {
                          System.out.println(t1.getName() + " interrupted"); }
try { System.out.println("Join on t2");
      t2.join();
      System.out.println("t2 has done with its job of calculating median");
        } catch (InterruptedException e) {
                              System.out.println(t2.getName() + " interrupted");
        }
        System.out.println("Average: " + avg.getMean() + ", Median: "
                                        + median.getMedian());
    }
```

ASPETTARE LA TERMINAZIONE DI UN THREAD

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

LA 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.
  - definiti in `java.lang.object`, perchè la sincronizzazione opera su oggetti

Costruttori: diversi costruttori che differiscono per i parametri utilizzati

- nome del thread, gruppo a cui appartine il thread,...(vedere le JAVA API)

LA CLASSE THREAD

Metodi

- interruzione, sospensione di un thread, attendere la terminazione di un thread
- porre un thread nello stato di 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
- metodi set e get per impostare e reperire le caratteristiche di un thread
  - esempio: assegnare nomi e priorità ai thread

ANALIZZARE LE PROPRIETA' DI UN THREAD

- La classe Thread salva alcune informazioni che aiutano ad identificare un 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 therad
  - 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

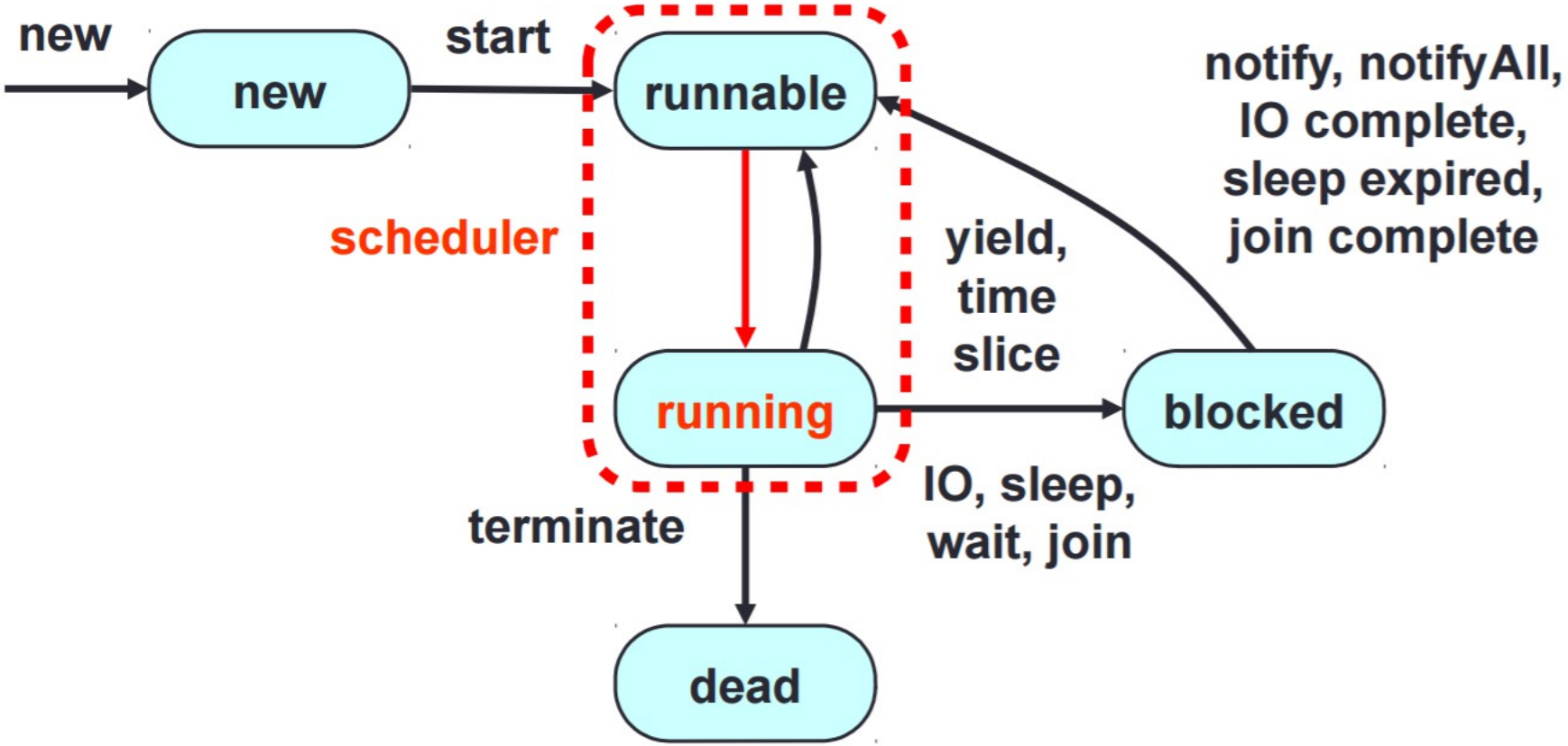
ANALIZZARE LE PROPRIETA' DI UN THREAD

```
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());
        System.out.println("NOMEGRUPPO"+
                  current.getThreadGroup().getName());
        }}
```

```
ID: 1
NOME: main
PRIORITA': 5
NOME GRUPPO: main
```

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

THREAD STATES NELLA JVM



THREAD STATE MONITORING

```
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)); (classe descritta
          if ((i%2)==0){                            in precedenza)
              threads[i].setPriority(Thread.MAX_PRIORITY);
              }
          else {
              threads[i].setPriority(Thread.MIN_PRIORITY);
              }
         threads[i].setName("Thread "+i);
      }
```

THREAD STATE MONITORING

```
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....
```

THREAD STATE MONITORING

```
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();}}
  finish=true;
  for (int i=0; i<10; i++){
     finish=finish &&(threads[i].getState()== Thread.State.TERMINATED);
   }}}}
```

THREAD STATE MONITORING

```
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

ASSIGNMENT 1: CALCOLO DI π

Scrivere un programma che attiva un thread T che effettua il calcolo approssimato di π. Il programma principale riceve in input da linea di comando un parametro che indica il grado di accuratezza (accuracy) per il calcolo di π ed il tempo massimo di attesa dopo cui il programma principale interomp thread T.

Il thread T effettua un ciclo infinito per il calcolo di π usando la serie di Gregory-Leibniz (π = 4/1 – 4/3 + 4/5 - 4/7 + 4/9 - 4/11 ...).

Il thread esce dal ciclo quando una delle due condizioni seguenti risulta verificata:

1) il thread è stato interrotto
2) la differenza tra il valore stimato di π ed il valore Math.PI (della libreria JAVA) è minore di accuracy