Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Programmazione parallela con i Java Thread – Thread e Task – Pool di Thread, Future e Callable

Thread e Task - I

Task (Runnable): unità logica di lavoro. Si tratta della sequenza di istruzioni che va eseguita, possibilmente in parallelo rispetto a altri task.

Thread: è il meccanismo di esecuzione di un task.

```
Class MyTask implements Runnable {
    ...
    public void run() {…}
}

// Creo un nuovo Thread e gli passo il task da eseguire
Thread t = new Thread(new MyTask());
```

Thread e Task - II

Sebbene i thread siano processi "light", occupano una discreta quantità di RAM e la loro creazione e distruzione genera lavoro (in termini di interazione con il sistema operativo) che, in presenza di molti thread, aumenta l'overhead nella JVM →

- La creazione di thread nuovi può andare bene per piccoli programmi, ma non è scalabile
- Limitare la creazione di nuovi thread e usare i Runnable per definire i task, da far eseguire ai thread in modo controllato

Thread e Task - III

Per organizzare un'applicazione in modo modulare e facilitare il parallelismo conviene

- suddividere la sua logica applicativa in Task - i compiti da fare (in modo indipendente) e
- specificare nella gestione dei Thread le politiche di esecuzione dei task (permettendo parallelismo, o imponendo sequenzialità, a seconda delle relazioni di dipendenza esistenti tra i task).

Thread pool - I

Un programma che crea numerosi thread, ciascuno dei quali fa attività brevi, è inefficiente

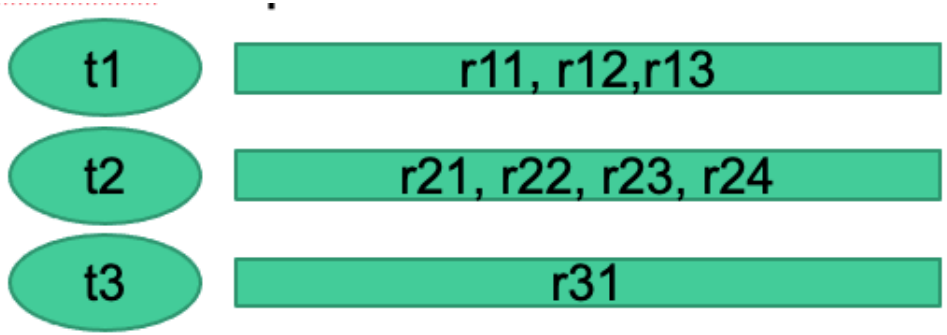
→**Thread pool:** il programma può creare un certo numero di thread in un colpo solo e poi distribuire su questi i task da eseguire

→La creazione di pool di thread e la loro esecuzione viene gestita dagli **Executors**, a cui si chiede di creare i pool e di mandarli in esecuzione

Thread pool - II

Insieme di thread omogenei e pronti all'esecuzione dei task che vengono loro assegnati.

Ogni thread del pool ha una coda di task da eseguire:



Quando un thread ha finito un task prende un task dalla sua coda. Se la coda è vuota, il thread ruba il task ad un altro thread. Se non ci sono task da fare il thread si sospende in attesa di nuovi task o di terminazione.

Thread pool: Executors - I

Package java.util.concurrent:

Executor interface: specifica la policy di esecuzione dei task di un'applicazione

- **void execute(Runnable command)**: esegue command ad un certo punto, nel futuro, secondo la policy specificata (ci sono tipi diversi di executor, vd. prossimo lucido)

Thread pool: Executors - II

Executors: factory degli oggetti di tipo Executor:

- **ExecutorService newFixedThreadPool(int n): crea un pool di thread con un numero fisso di elementi**
- **ScheduledExecutorService newScheduledThreadPool(int n)**: crea un pool thread con scheduling delle attività (serve per creare pool di thread che eseguiranno i task con un ritardo specificato in input, oppure ciclicamente)
- **ExecutorService newSingleThreadExecutor()** crea un Executor che usa un solo thread e usa la ThreadFactory per crearne di nuovi
- ...

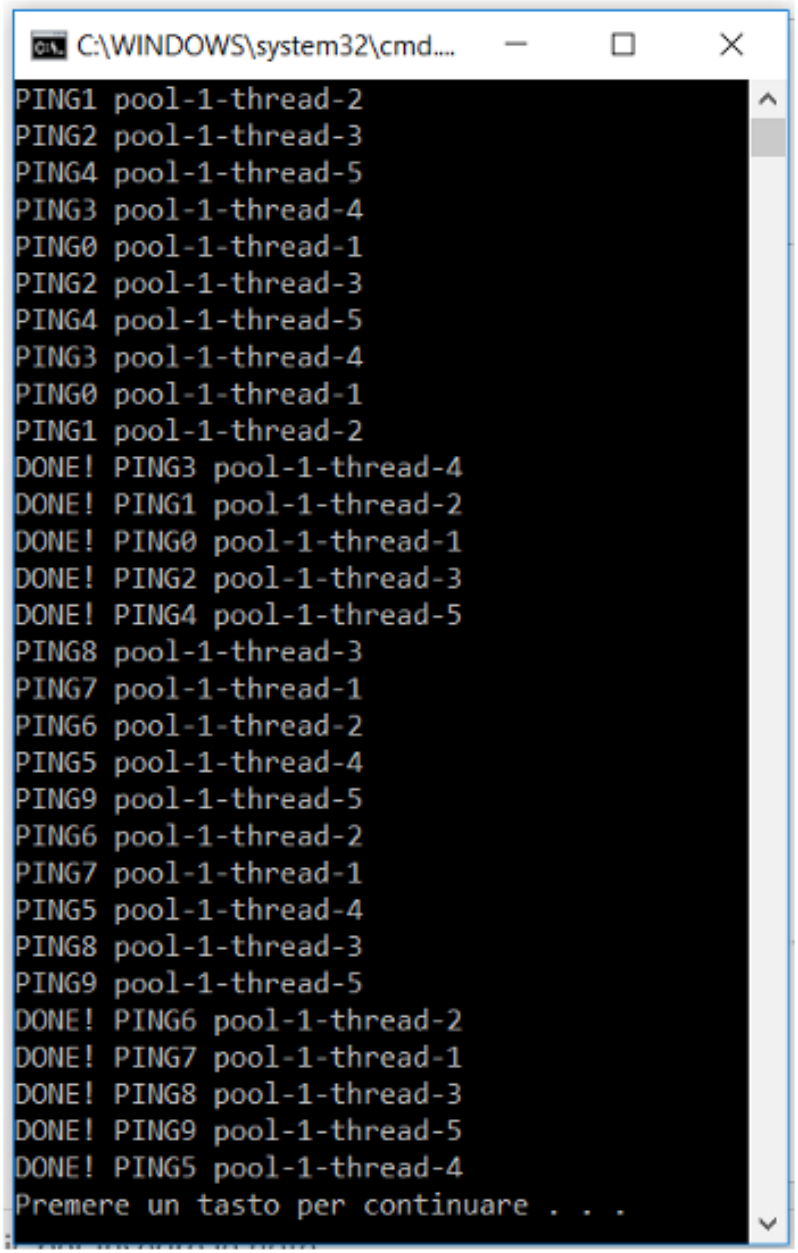
Thread pool fisso – Esempio - I

```
class PingPong implements Runnable {
    private int delay; private String name;

    public PingPong(String str, int d) {
          delay = d; name = str;  }

    public void run() {
       for (int i = 1; i < 3; i++) {
          System.out.println(name + " " +
                              Thread.currentThread().getName()););
          try {
             Thread.sleep(delay);
          } catch (InterruptedException e) {return;}
       }
       System.out.println("DONE! "+name+"
                              "+Thread.currentThread().getName());
    }
}
```

Thread pool fisso – Esempio - II

```
public class PingPongThreadPool {
  private static final int NUM_THREAD = 5;

  public static void main (String[] args) {
    ExecutorService exec = Executors.newFixedThreadPool(NUM_THREAD);
      for (int i=0; i<10; i++) {        // creo 10 task da eseguire
          Runnable task = new PingPong("PING"+i, 200);
          exec.execute(task); }
    exec.shutdown(); // initiates an orderly shutdown in which previously
          // submitted tasks are executed, but no new tasks will be accepted
    try {
          exec.awaitTermination(5, TimeUnit.SECONDS);
          //Blocks until all tasks have completed execution after a shutdown
    request,
          // or the timeout occurs, or the current thread is interrupted,
          // whichever happens first (lo uso se voglio terminare entro una
    scadenza)
      } catch (InterruptedException e) {
          System.out.println(e.getMessage());
      }
}}
```

Thread pool fisso – Esecuzione



i 5 thread del pool eseguono 2 task per ciascuno

Thread pool fissi e "graceful degrading" delle applicazioni

I thread pool fissi sono essenziali per permettere alle applicazioni di degradare le proprie prestazioni, quando sotto carico eccessivo, senza smettere di funzionare.

Es. web server: ogni richiesta HTTP viene servita da un thread. Se il web server creasse un thread per ogni richiesta, in presenza di molte richieste ne genererebbe troppi → smetterebbe di funzionare. Invece con un pool fisso di thread servirà le richieste in coda man mano che termina l'esecuzione delle precedenti → ritardo, ma senza crash dell'applicazione

Interface Callable

- I task che devono restituire un risultato al chiamante possono essere implementati come Callable

- **Callable<T>:**
  - Offre metodo **T call()** (analogo a run()) per l'esecuzione del task
  - La sua esecuzione restituisce un valore di tipo T

Interface Future<T>

Rappresenta il risultato di una computazione asincrona. Offre i metodi per verificare se la computazione è completata e per estrarre il risultato. Il tipo T è quello del risultato della computazione

- **FutureTask(Callable<T> task):** crea un FutureTask che, quando parte, esegue il task passato come parametro.
- **boolean isDone():** per verificare se il task è completato.
- **T get():** se necessario attende che la computazione asincrona termini; poi restituisce il suo risultato.
  - NB: se il task non termina, chi invoca la get() resta bloccato in attesa – ma esistono metodi con time-out, in caso si possa ignorare qualche risultato

Future-Callable – Esempio - I

```
class Computazione implements Callable<Integer> {
    private int num;
        // Computazione è un task che restituisce un risultato
    public Computazione(int n) {num = n;}

    public Integer call() {
        System.out.println("INIZIO computazione " + num);
        try {
                Thread.sleep((int) Math.round(Math.random() * 4000));
                } catch (InterruptedException e)
                        {System.out.println("Interruzione thread");}
        System.out.println("FINE computazione " + num +
                                        ": risultato = " + 2*num);
        return new Integer(2*num);
    }
}
```

Future-Callable – Esempio - II

```
public static void main (String[] args) {
                    // estratto di codice da esempio FutureCallable
                    // preparo il pool di esecutori dei task callable
    ExecutorService exec =
                    Executors.newFixedThreadPool(NUM_THREADS);
          ... // creo il task callable
    FutureTask<Integer> ft = new FutureTask<>(new Computazione(i));
    exec.execute(ft);
        ...
    try {
        int result = (ft.get()).intValue();
    } catch (Exception e) {System.out.println(e.getMessage());}
}
```

Future-Callable – Esempio - esecuzione

```
inizio il main
INIZIO computazione 0
INIZIO computazione 1
FINE computazione 1: risultato = 2
INIZIO computazione 2
FINE computazione 0: risultato = 0
INIZIO computazione 3
FINE computazione 3: risultato = 6
INIZIO computazione 4
FINE computazione 4: risultato = 8
INIZIO computazione 5
FINE computazione 2: risultato = 4
INIZIO computazione 6
FINE computazione 5: risultato = 10
INIZIO computazione 7
FINE computazione 7: risultato = 14
INIZIO computazione 8
FINE computazione 6: risultato = 12
INIZIO computazione 9
FINE computazione 9: risultato = 18
FINE computazione 8: risultato = 16

Il totale è: 90
```

Thread pool: Executors - III

Executors: factory degli oggetti di tipo Executor:

- **ExecutorService newFixedThreadPool(int n)**: crea un pool di thread con un numero fisso di elementi
- **ScheduledExecutorService newScheduledThreadPool(int n): crea un pool thread con scheduling delle attività** (serve per creare pool di thread che eseguiranno i task con un ritardo specificato in input, oppure ciclicamente)
- **ExecutorService newSingleThreadExecutor()** crea un Executor che usa un solo thread e usa la ThreadFactory per crearne di nuovi
- ...

SheduledExecutorService - I

schedule (Runnable task, long delay, TimeUnit timeunit)

```
public static void main (String[] args) {
    ScheduledExecutorService exec =
        Executors.newScheduledThreadPool(3); // pool di 3 thread

        ScheduledFuture<Integer> task =
                        exec.schedule(new Computazione1(2),
                                      5,       // tempo di attesa
                                      TimeUnit.SECONDS);
        try {
                System.out.println("result = " + task.get());
        } catch (Exception e){System.out.println(e.getMessage());}
        exec.shutdown();
}
```

// TimeUnit temporizza esecuzione di task con ritardo

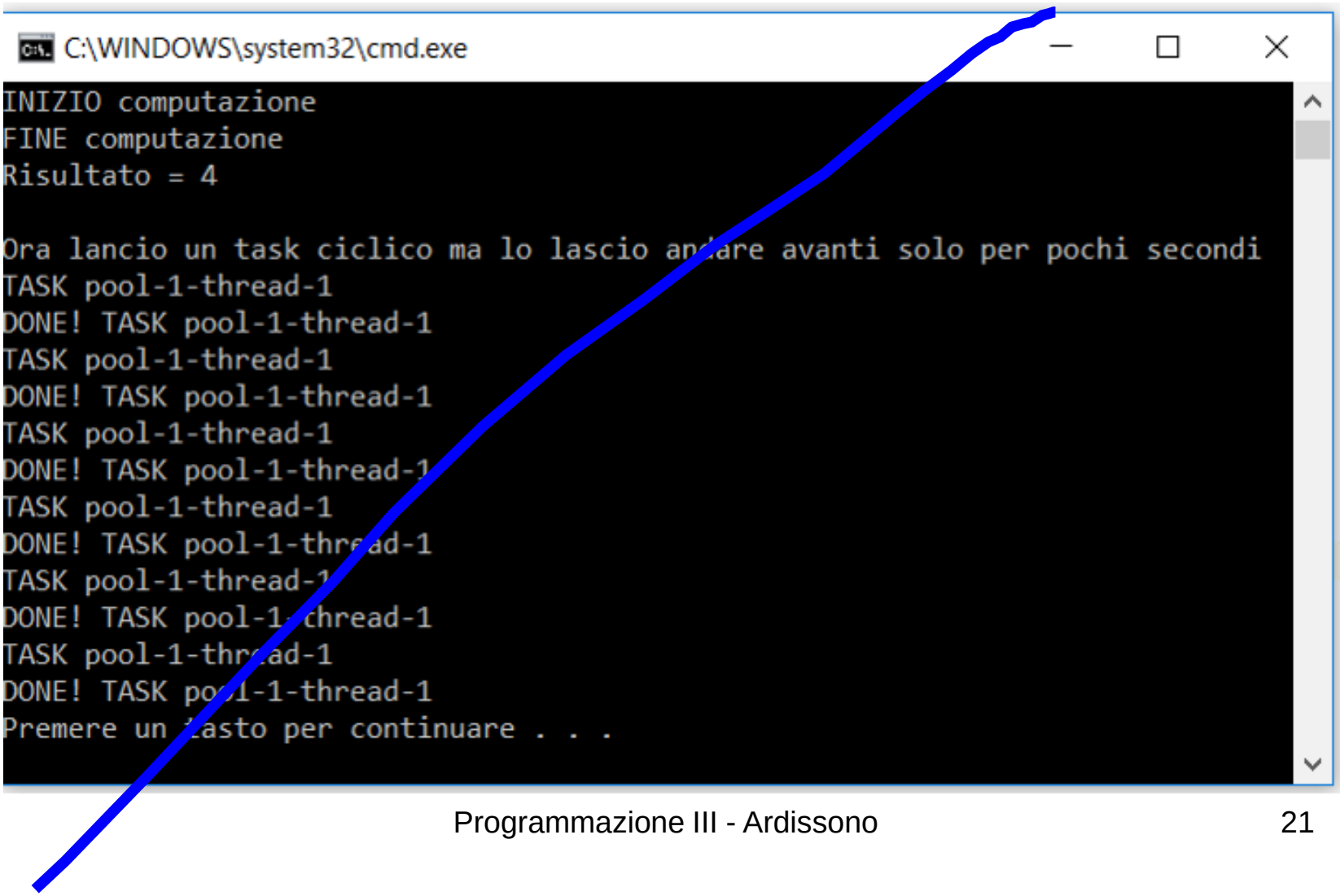
SheduledExecutorService - II

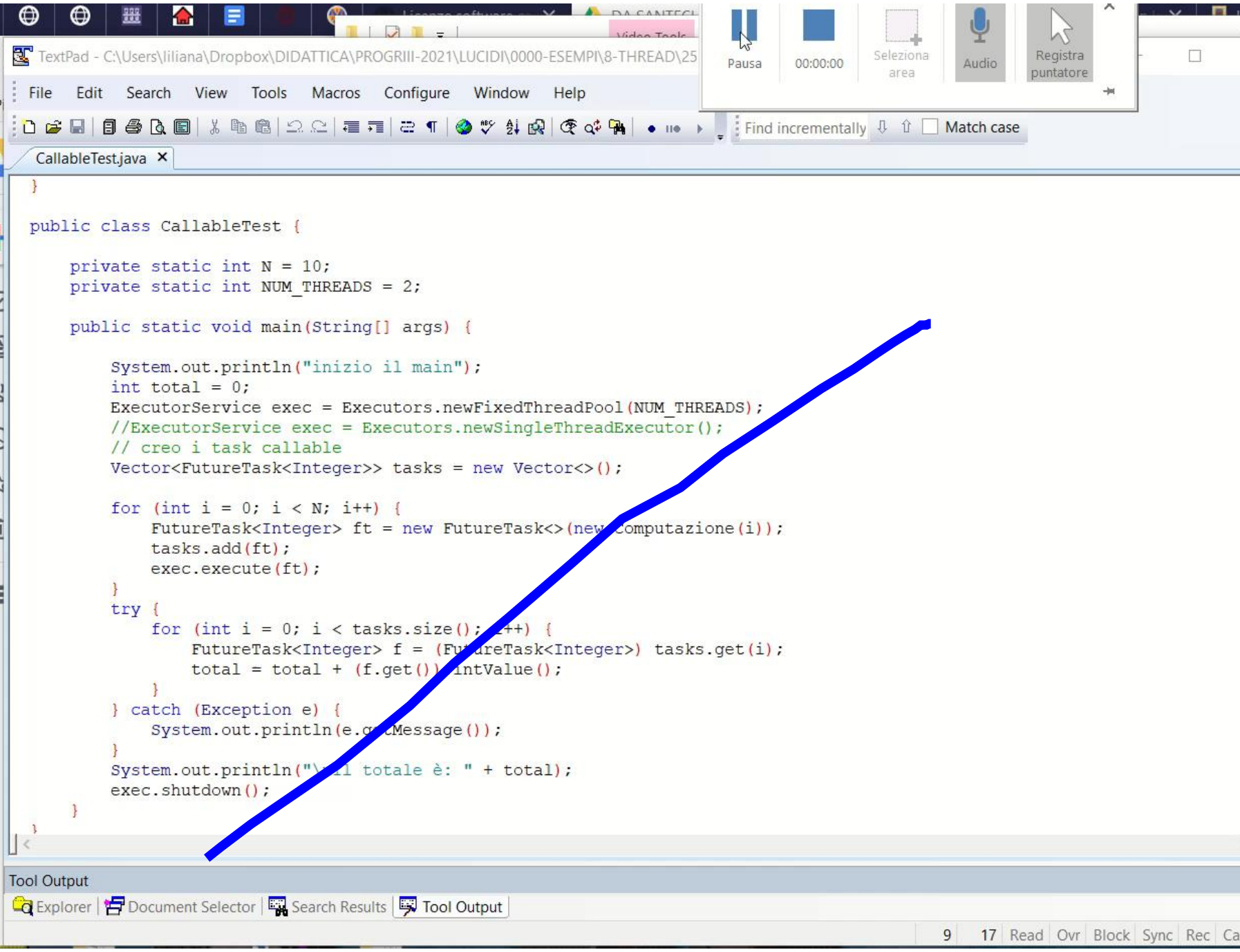
Esecuzione ciclica di task con ritardo iniziale (initDelay)

scheduleAtFixedRate (Runnable, long initDelay, long period, TimeUnit timeunit)

```
public static void main (String[] args) {
        ScheduledExecutorService exec =
                Executors.newScheduledThreadPool(3);

        exec.scheduleAtFixedRate (new MyTask("TASK"),
                                  2,
                                  3,
                                  TimeUnit.SECONDS );

        // esecuzione ciclica infinita. Se voglio fermarla devo inserire lo
        // shutdown ma dopo un certo tempo se no il thread termina subito
        // e non esegue mai il task. Vd. ScheduledExecutorTest
}
```

ScheduledExecutorTest – Esecuzione




TextPad - C:\Users\liliana\Dropbox\DIDATTICA\PROGRIII-2021\LUCIDI\0000-ESEMPI\8-THREAD\25
File Edit Search View Tools Macros Configure Window Help
Find incrementally
Match case
CallableTest.java
}
public class CallableTest {
private static int N = 10;
private static int NUM_THREADS = 2;
public static void main(String[] args) {
System.out.println("inizio il main");
int total = 0;
ExecutorService exec = Executors.newFixedThreadPool(NUM_THREADS);
//ExecutorService exec = Executors.newSingleThreadExecutor();
// creo i task callable
Vector<FutureTask<Integer>> tasks = new Vector<>();
for (int i = 0; i < N; i++) {
FutureTask<Integer> ft = new FutureTask<>(new Computazione(i));
tasks.add(ft);
exec.execute(ft);
}
try {
for (int i = 0; i < tasks.size(); i++) {
FutureTask<Integer> f = (FutureTask<Integer>) tasks.get(i);
total = total + (f.get()).intValue();
}
} catch (Exception e) {
System.out.println(e.getMessage());
}
System.out.println("\nil totale è: " + total);
exec.shutdown();
}
}
Pausa
00:00:00
Seleziona area
Audio
Registra puntatore
Tool Output
Explorer
Document Selector
Search Results
Tool Output
9
17
Read
Ovr
Block
Sync
Rec
Caps

Conclusioni

I pool di thread non aggiungono nulla ai concetti che ho descritto (parallelismo etc.), ma servono per sviluppare applicazioni scalabili.

Anche nel caso di pool di thread potrebbe essere necessario gestire accessi in mutua esclusione a risorse condivise → si usano i lock, Semaphore, synchronized, etc., per gestire le sezioni critiche, wait e notify per attendere condizioni, etc.