

Reti e Laboratorio

Modulo Laboratorio III

AA. 2025-2026

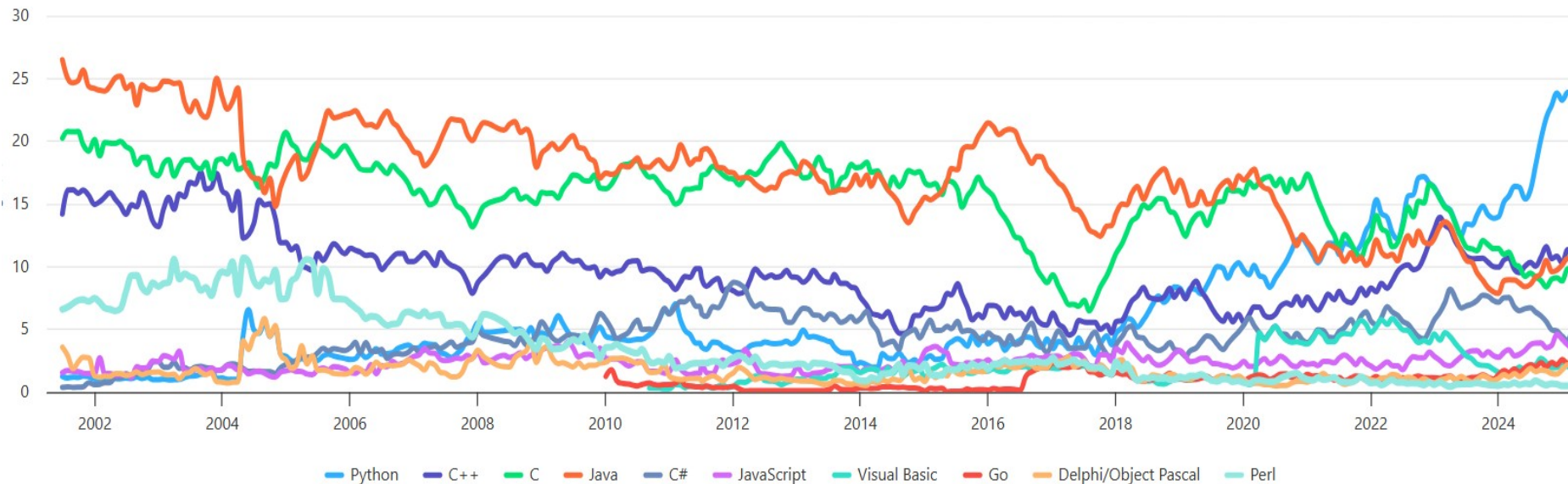
docente: Laura Ricci

laura.ricci@unipi.it

Lezione 1

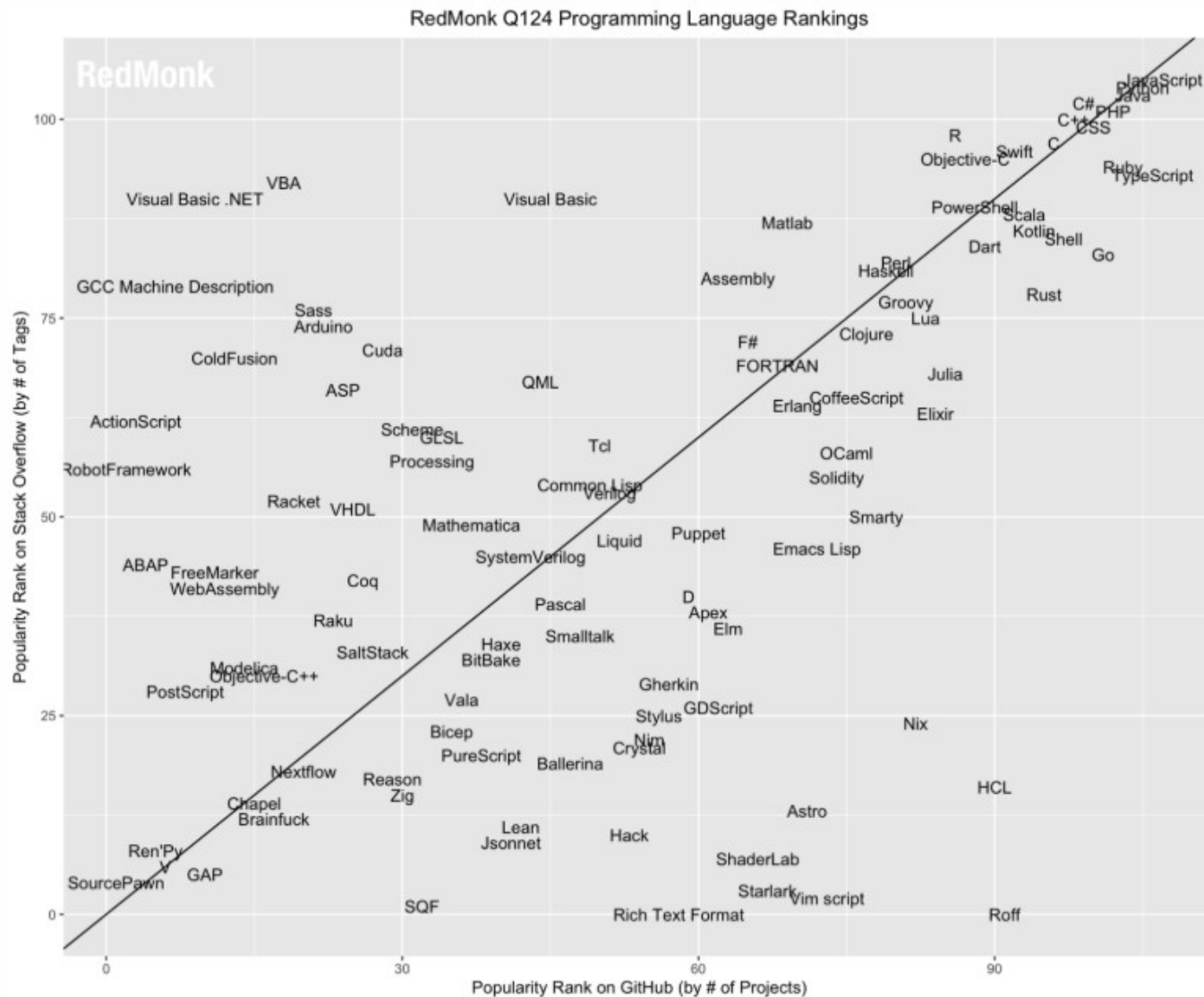
**JAVA multithreading:
threads e thread pooling
19/09/2025**

PERCHE' JAVA?: L'INDICE TIOBE DEI LINGUAGGI: WI



- misura la popolarità dei linguaggi di programmazione in funzione del numero di ricerche contenenti il nome del linguaggio
- fonti: motori di ricerca, Wikipedia, Youtube
- JAVA uno dei top-4 linguaggi

PERCHE' JAVA?: REDMONK RANKING



JAVA: CARATTERISTICHE

- **object-oriented**
 - un modello chiaro e semplice per gli oggetti
- **robusto**
 - controlli a tempo di compilazione (strong typing) e a run-time (exception handling), gestione automatica della memoria
- **multithreaded**
- **interpretato**
 - compilato in bytecode (rappresentazione intermedia), poi interpretato
- **portabile**
 - JVM: ambiente di esecuzione indipendente dalla piattaforma
- **distribuito**
 - gestione del protocollo TCP/IP
- **sicuro**
 - ambiente di esecuzione ben definito e confinato

CHE TIPO DI APPLICAZIONI DI RETE?

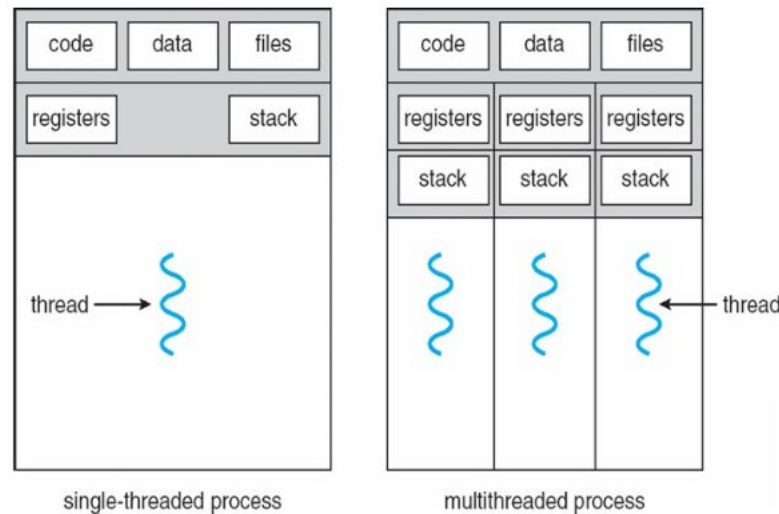
- applicazioni client server
 - web browsers
 - email
 - social networks
 - teleconferences (skype, Zoom,...)
 - home banking
 - collaborative work: Overleaf
 - multiplayer games: War of Warcraft
- applicazioni peer-to-peer
 - P2P File sharing: Bittorrent
 - blockchain: cryptocurrencies (Bitcoin), NFT,...
- scopo del corso: fornire gli strumenti per sviluppare una semplice applicazione di rete, utilizzando costrutti ad alto livello offerti da JAVA per multithreading e networking e I socket JAVA

NETWORK APPLICATIONS

- due o più **processi** (non thread!) in esecuzione su **hosts diversi**, distribuiti geograficamente sulla rete **comunicano** e **cooperano** per realizzare una funzionalità globale
- ogni processo può essere strutturati utilizzando
 - multithreading
 - multiplexing dei canali
- i **processi comunicano sulla rete**: per comunicare si utilizzano protocolli, ovvero insieme di regole che i partners devono seguire per comunicare correttamente.
- in questo corso utilizzeremo i protocolli di livello trasporto:
 - **connection-oriented**: TCP, Transmission Control Protocol
 - **connectionless**: UDP, User Datagram Protocol

PROCESSI E THREADS: RICHIAMI

- 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



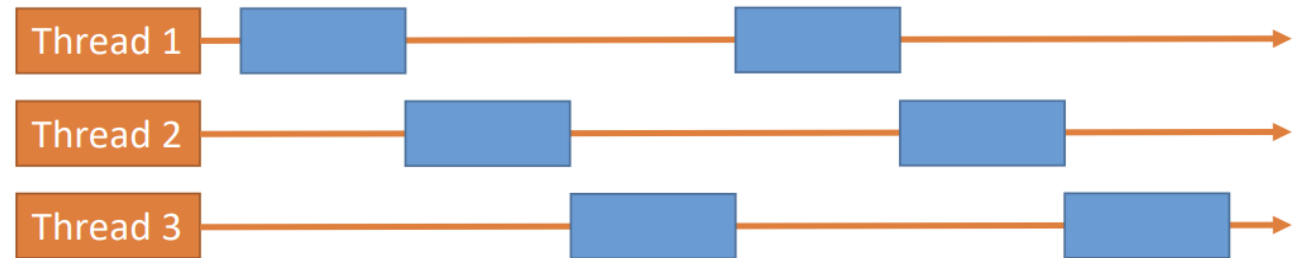
- multitasking, si può riferire a thread o processi
 - a livello di processo è controllato esclusivamente dal sistema operativo
 - a livello di thread è controllato, almeno in parte, dal programmatore

PROCESSI E THREADS: RICHIAMI

- thread multitasking contro 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

MULTITHREADING: UNA VS. MOLTE CPU

Multiple threads sharing a single CPU



Multiple threads on multiple CPUs



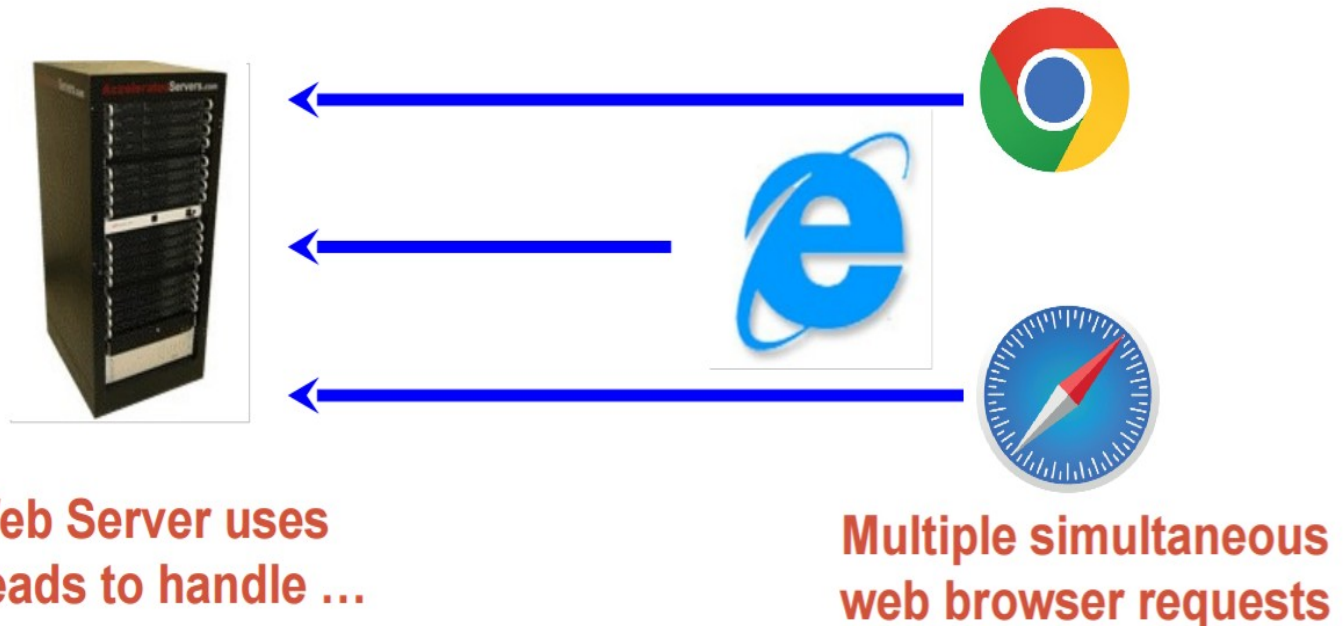
MULTITHREADING: PERCHE'?

- migliore utilizzazione delle risorse
 - quando un thread è sospeso, altri thread vengono mandati in esecuzione
 - riduzione del tempo complessivo di esecuzione (in generale)
- migliori 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,...

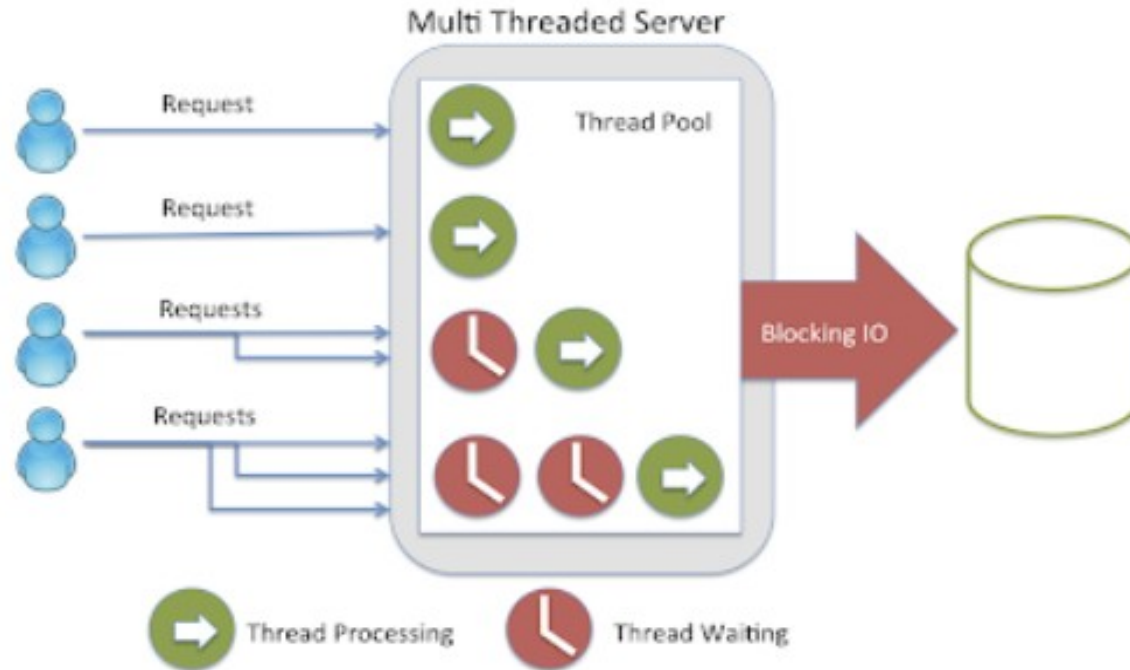
- Ok, ma abbiamo già visto la programmazione concorrente in altri corsi...
- trove le differenze
 - si adotterà un approccio basato sull'impiego di costrutti ad alto livello per la gestione della concorrenza e della sincronizzazione,
 - motivazioni:
 - maggiore leggibilità, manutenibilità e robustezza del codice,
 - riduzione della complessità e dei potenziali rischi derivanti dall'uso diretto di meccanismi di sincronizzazione a basso livello, come le lock esplicite
- costrutti utilizzati: Thread pooling, monitors, concurrent collections,

THREAD E PROGRAMMAZIONE DI RETE

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



THREAD E PROGRAMMAZIONE DI RETE



- 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
- limitare questo fenomeno con il meccanismo del [threadpooling](#)

JAVA UTIL.CONCURRENT FRAMEWORK

- JAVA < 5 built in for concurrency: lock implicite, wait, notify e poco più.
 - `JAVA.util.concurrent`
 - lo stesso scopo 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

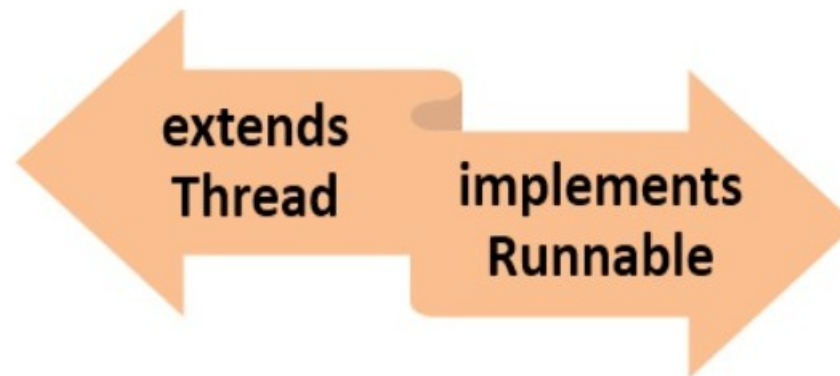
- sviluppato in parte da Doug Lea, disponibile, come insieme di librerie JAVA non standard prima della integrazione in JAVA 5.0.
- tra i package principali:
 - `java.util.concurrent`
 - executor, concurrent collections, semaphores,...
 - `java.util.concurrent.atomic`
 - AtomicBoolean, AtomicInteger,...
 - `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`

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
 - quindi 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.
- due modalità per creare ed attivare esplicitamente un thread





- definire un task
- creare un oggetto thread e passargli il task definito che contiene il codice 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

CREAZIONE-ATTIVAZIONE DI THREAD: SOLUZIONE 1

```
public class ThreadRunnable {  
    public 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.lang`
- 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

CREAZIONE-ATTIVAZIONE DI THREAD: SOLUZIONE 2

- creare una classe che estenda la classe predefinita `Thread`
- effettuare l'`overriding` del metodo `run()`
- istanziare un oggetto di quella classe
 - questo oggetto è un thread
 - il suo 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
- decisione a run-time su quale metodo viene invocare in base all'istanza su cui si invoca il metodo

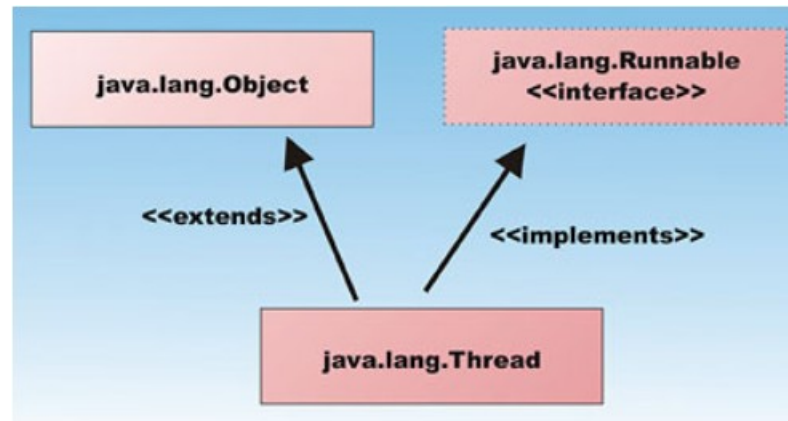
```
public class ExtendingThread {  
  
    public static class MyThread extends Thread {  
        public void run() {  
            System.out.println("MyThread running");  
            System.out.println("MyThread finished");  
        }  
    }  
  
    public static void main (String [] args) {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

Stampa

MyThread running

MyThread finished

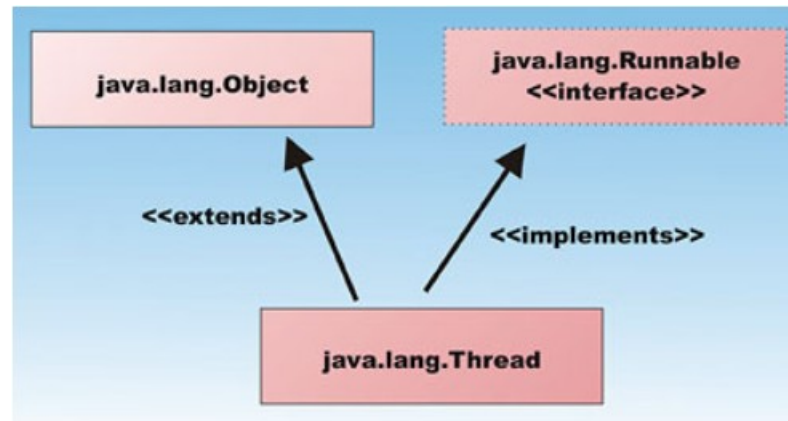
LA CLASSE THREAD



- 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 metodo `run()` più specifico, che è quello definito dal programmatore
 - altrimenti, si esegue il metodo `run()` predefinito nella classe `Thread`, (soluzione 1)
 - se la variabile `runnable` è diversa da `null`, 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 effettuata, 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 schedatore):

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 l'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

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?

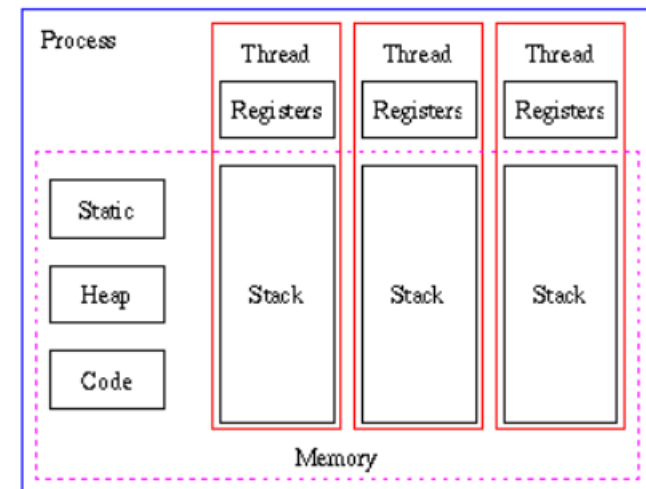
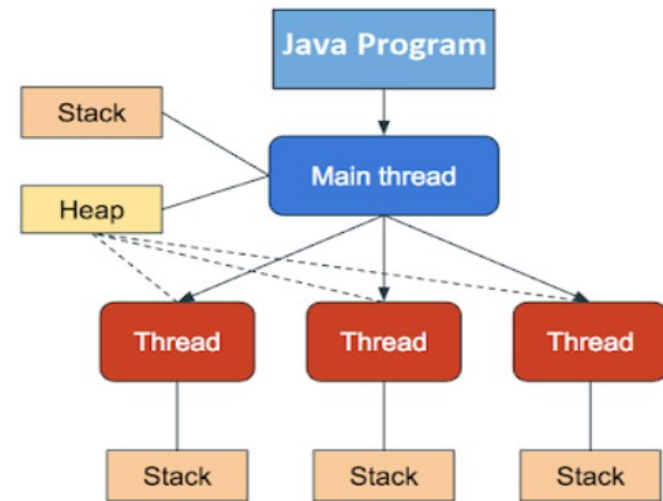
- in JAVA una classe può estendere una solo altra classe (**eredità singola**)
 - se si estende la classe Thread, la classe i cui oggetti devono essere eseguiti come thread non può estendere altre classi.
- questo può risultare svantaggioso in diverse situazioni, ad esempio:
 - gestione di eventi dell'interfaccia (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 l'interfaccia Runnable

TERMINAZIONE DI PROGRAMMI CONCORRENTI

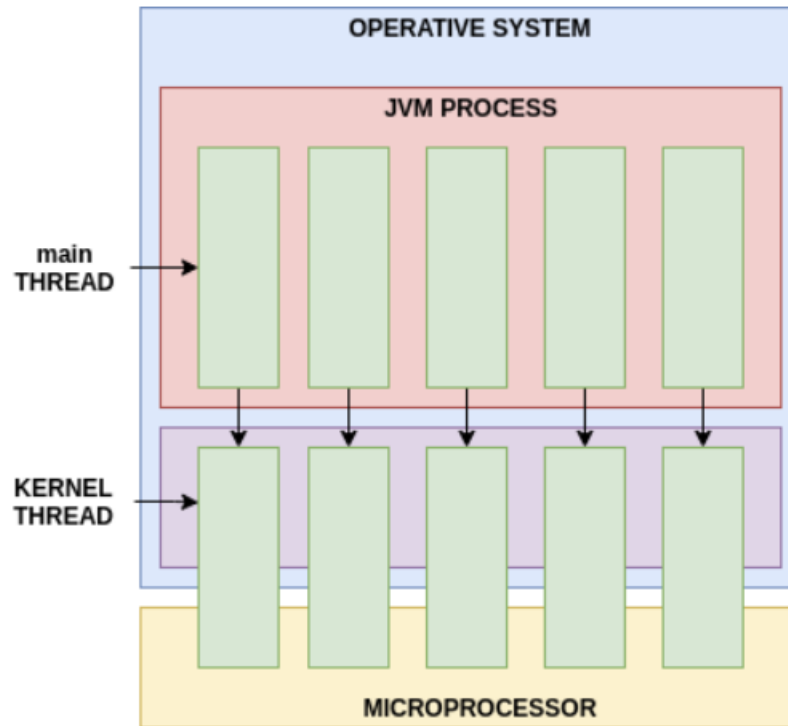
- un programma JAVA termina quando terminano tutti i threads **non demoni** che lo compongono
- se il thread iniziale, cioè quello che esegue il metodo `main()` termina, i restanti thread ancora attivi e non demoni continuano la loro esecuzione, il programma termina quando anche questi terminano.
 - il “quadrato” 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

THREAD OVERHEAD

- **attivazione/eliminazione** di thread
 - richiede interazione tra JVM e SO
 - impatto sulle prestazioni variabile a seconda del SO
 - mai trascurabile, specie per richieste di servizio frequenti e 'lightweight'
- **resource consumption**
 - alloca uno stack per ogni thread
 - garbage collector stress
 - alcuni SO limitano per questo max numero di thread per programma



LIMITAZIONI DEL JAVA THREAD MODEL



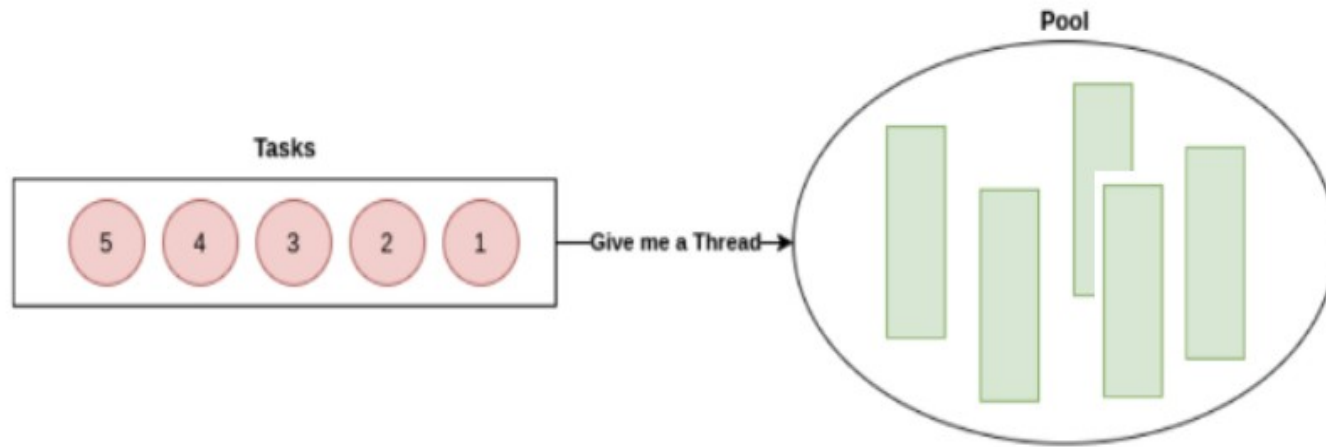
Java Thread model

- numero di thread limitato dal livello di capacità di kernel thread
- “JAVA break” se si usano più thread di quelli supportati dal SO

THREAD POOL: MOTIVAZIONI

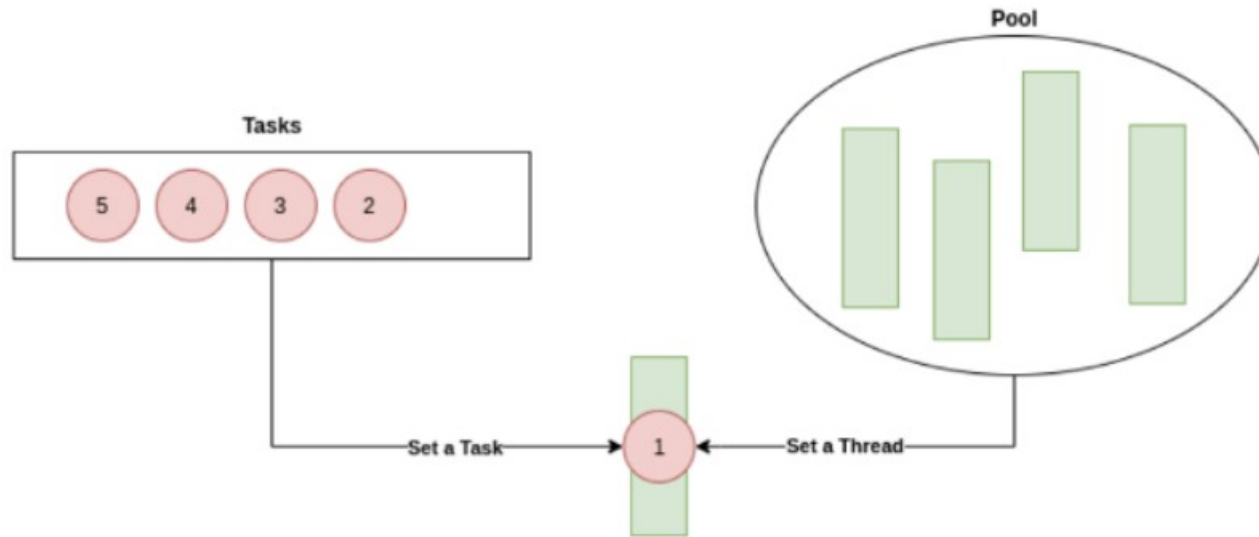
- scenario di riferimento: si deve eseguire un gran numero di task
 - esempio: un task nel server per ogni client
- un thread per ogni task: può diventare non sostenibile, specialmente nel caso di lightweight tasks molto frequenti
- alternativa
 - creare un **pool di thread**
 - ogni thread può essere usato per l'esecuzione di più task
- obiettivo:
 - **riusare lo stesso thread** per l'esecuzione di più tasks
 - diminuire il costo per l'attivazione/terminazione dei threads
 - controllare il numero massimo di thread che possono essere eseguiti concorrentemente

THREAD POOLING: CONCETTI DI BASE



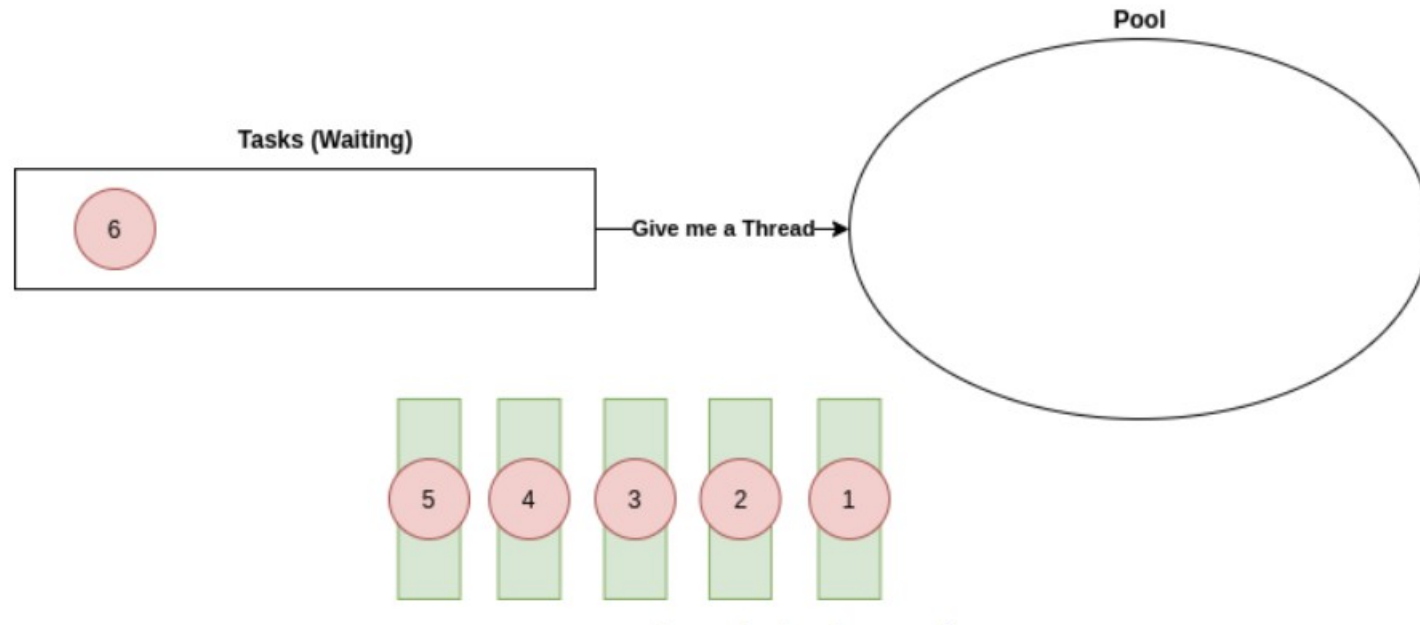
- una coda di task che aspettano l'esecuzione
 - politica FIFO per l'estrazione dei task dalla coda
- un pool di thread disponibili (rettangoli verdi) per l'esecuzione di un task
- il sistema di gestione del threadpool chiede se esiste un thread libero per l'esecuzione del primo task della coda

THREAD POOLING: CONCETTI DI BASE



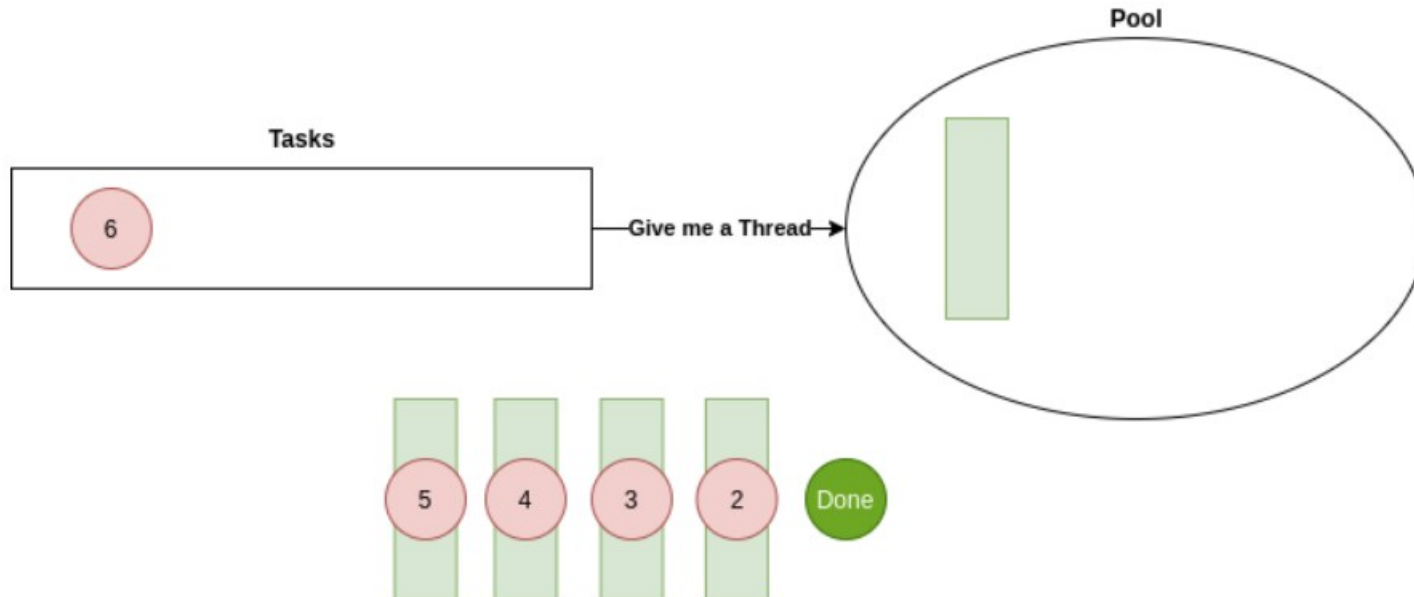
- il task viene assegnato ad un thread libero
- il thread viene tolto dal pool dei thread disponibili

THREAD POOLING: CONCETTI DI BASE



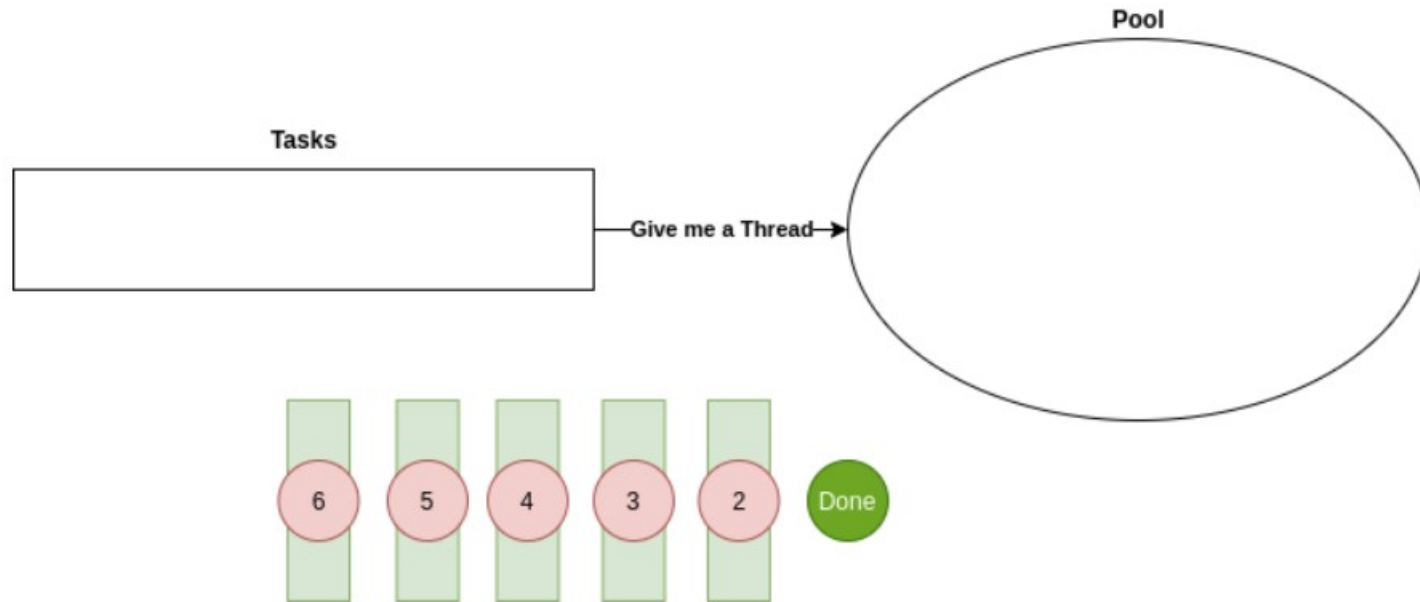
- tutti i thread sono occupati nell'esecuzione di task
- il pool è vuoto
- due alternative
 - il task successivo viene inserito nella coda, in attesa che si renda disponibile un thread
 - si crea un nuovo thread all'interno del threadpool

THREAD POOLING



- supponiamo che il task 6 attenda nella coda
- quando un thread finisce l'esecuzione del task assegnato, il thread ritorna nel pool e si rende disponibile per l'esecuzione di un altro task

THREAD POOLING: CONCETTI DI BASE



- il task in attesa viene associato al thread che si è reso disponibile
- il pool di thread ritorna ad essere vuoto
- il comportamento descritto è quello del `FixedThreadPool` di JAVA
 - in JAVA disponibili altre politiche di gestione dei threads

UN PO' DI TERMINOLOGIA

- l'utente struttura l'applicazione mediante un insieme di tasks.
- task segmento di codice che può essere eseguito da un esecutore
 - in JAVA corrisponde ad un oggetto di tipo Runnable
- Thread
 - un esecutore di tasks.
- Thread Pool
 - struttura dati la cui dimensione massima può essere prefissata, che contiene riferimenti ad un insieme di threads
 - i thread del pool possono essere riutilizzati per l'esecuzione di più tasks
 - la sottomissione di un task al pool viene disaccoppiata dall'esecuzione del thread.
 - l'esecuzione del task può essere ritardata se non vi sono risorse disponibili

THREAD POOL: CONCETTI GENERALI

- il progettista
 - crea il **pool** e stabilisce una politica per la gestione dei thread del pool
 - quando i thread **vengono attivati**: al momento della creazione del pool, on demand, all'arrivo di un nuovo task,....
 - se e quando è opportuno **terminare l'esecuzione di un thread**
 - se non c'è un numero sufficiente di tasks da eseguire
 - sottomette i tasks per l'esecuzione al thread pool
- il supporto, al momento della sottomissione del task, può
 - **utilizzare un thread attivato in precedenza**, inattivo in quel momento
 - **creare un nuovo thread**
 - **memorizzare il task** in una **struttura dati (coda)**, in attesa dell'esecuzione
 - **respingere** la richiesta di esecuzione del task
- il numero di threads attivi nel pool può **variare dinamicamente**

JAVA THREADPOOL: IMPLEMENTAZIONE

- fino a JAVA 4 la programmazione del threadpool è a carico del programmatore
- JAVA 5.0 definisce la libreria `java.util.concurrent` che contiene metodi per
 - creare un thread pool ed il gestore associato
 - definire specifiche politiche per la gestione del pool
 - tipo di coda
 - elasticità del threadpool
- il meccanismo introdotto permette una **migliore strutturazione del codice** poichè tutta la gestione dei threads può essere delegata al supporto

JAVA THREADPOOL: IMPLEMENTAZIONE

- alcune interfacce definiscono servizi generici di esecuzione

```
public interface Executor {  
    public void execute (Runnable task) }  
public interface ExecutorService extends Executor  
    {.. }
```

- diversi servizi implementano il generico ExecutorService (ThreadPoolExecutor, ScheduledThreadPoolExecutor,...)
- la classe `Executors` opera come una Factory in grado di generare oggetti di tipo ExecutorService con comportamenti predefiniti.
- i tasks devono essere incapsulati in oggetti di tipo Runnable e passati a questi esecutori, mediante invocazione del metodo `execute()`

I TASK DA SOTTOMETTERE AL POOL

```
import java.util.*;

public class Task implements Runnable {
    private int name;

    public Task(int name) {this.name=name;}

    public void run() {
        try{
            Long duration=(long)(Math.random()*10);
            System.out.printf("%s: Task %s: Starting a task during %d seconds\n",
                               Thread.currentThread().getName(),name,duration);
            Thread.sleep(duration);
        }
        catch (InterruptedException e) {e.printStackTrace();}
        System.out.printf("%s: Task Finished %s \n",
                           Thread.currentThread().getName(),name);}}}

```


FIXEDTHREADPOOL

- un tipo di threadpool con comportamento predefinito
- viene creato un numero fisso di thread: n thread, n fissato al momento dell'inizializzazione del pool, riutilizzati per l'esecuzione di più tasks
- quando viene sottomesso un task T
 - se tutti i threads sono occupati nell'esecuzione di altri tasks, T viene inserito in una coda, gestita automaticamente dall' `ExecutorService`
 - se almeno un thread è inattivo, viene utilizzato quel thread
- utilizza una `LinkedBlockingQueue`
- coda illimitata

FIXEDTHREADPOOL

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
public class ExampleFixed{
    public static void main(String[] args) {
        // create the pool
        ExecutorService service = Executors.newFixedThreadPool(10);
        //submit the task for execution
        for (int i=0; i<100; i++) {
            service.execute(new Task(i)) }
        System.out.println("Thread Name:"+
            Thread.currentThread().getName());
    } }
```

L'OUTPUT DEL PROGRAMMA

Thread Name:main

```
pool-1-thread-7: Task 6: Starting during 6 seconds
pool-1-thread-9: Task 8: Starting during 9 seconds
pool-1-thread-8: Task 7: Starting during 7 seconds
pool-1-thread-10: Task 9: Starting during 9 seconds
pool-1-thread-2: Task 1: Starting during 9 seconds
pool-1-thread-4: Task 3: Starting during 9 seconds
pool-1-thread-1: Task 0: Starting during 1 seconds
pool-1-thread-6: Task 5: Starting during 0 seconds
pool-1-thread-3: Task 2: Starting during 9 seconds
pool-1-thread-5: Task 4: Starting during 3 seconds
pool-1-thread-6: Task Finished 5
pool-1-thread-6: Task 10: Starting during 9 seconds
pool-1-thread-1: Task Finished 0
pool-1-thread-1: Task 11: Starting during 3 seconds
pool-1-thread-5: Task Finished 4
pool-1-thread-5: Task 12: Starting during 5 seconds
pool-1-thread-1: Task Finished 11
pool-1-thread-7: Task Finished 6
pool-1-thread-1: Task 13: Starting during 1 seconds
pool-1-thread-7: Task 14: Starting during 2 seconds
```

.....

importante:

- lo stesso thread riutilizzato per più tasks

DISTANZIARE I TASK

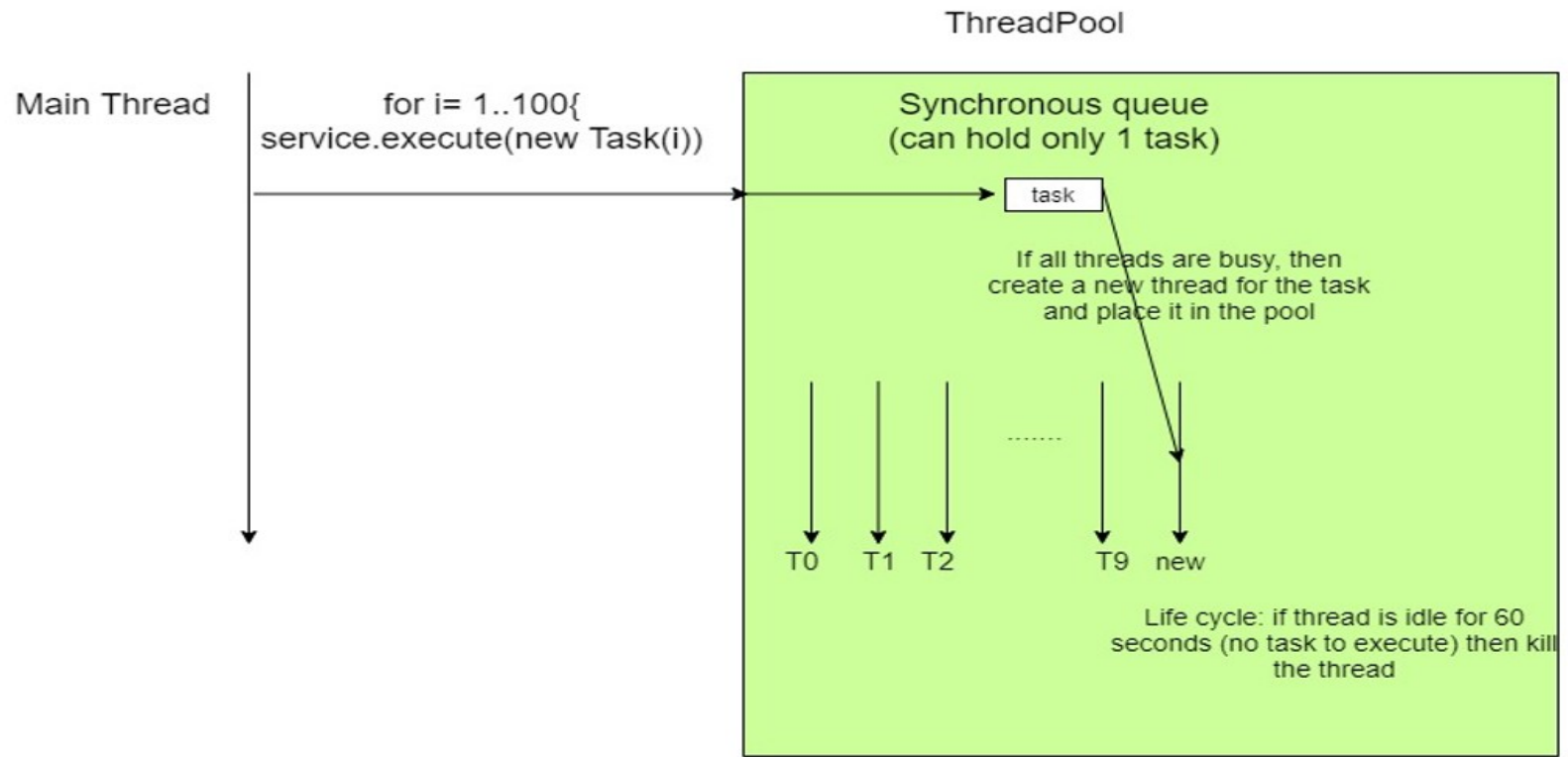
```
pool-1-thread-1: Task 0: Starting during 4 seconds
pool-1-thread-1: Task Finished 0
pool-1-thread-2: Task 1: Starting during 7 seconds
pool-1-thread-2: Task Finished 1
pool-1-thread-3: Task 2: Starting during 0 seconds
pool-1-thread-3: Task Finished 2
pool-1-thread-4: Task 3: Starting during 0 seconds
pool-1-thread-4: Task Finished 3
pool-1-thread-5: Task 4: Starting during 4 seconds
pool-1-thread-5: Task Finished 4
pool-1-thread-6: Task 5: Starting during 2 seconds
pool-1-thread-6: Task Finished 5
pool-1-thread-7: Task 6: Starting during 9 seconds
pool-1-thread-7: Task Finished 6
pool-1-thread-8: Task 7: Starting during 5 seconds
pool-1-thread-8: Task Finished 7
pool-1-thread-9: Task 8: Starting during 5 seconds
pool-1-thread-9: Task Finished 8
pool-1-thread-10: Task 9: Starting during 6 seconds
pool-1-thread-10: Task Finished 9
pool-1-thread-1: Task 10: Starting during 3 seconds
pool-1-thread-1: Task Finished 10
```

- cosa accade se si distanzia la sottomissione dei task ai thread, ad esempio inserendo una sleep, nel for dopo la execute?
- i thread sono tutti attivi e vengono utilizzati in modalità round-robin

CACHEDTHREADPOOL

- un altro tipo di threadpool con comportamento predefinito
- attivato con

ExecutorService service = Executors.newCachedThreadPool();



CACHEDTHREADPOOL

- se tutti i thread del pool sono occupati nell'esecuzione di altri task e c'è un nuovo task da eseguire, viene creato un nuovo thread.

nessun limite alla dimensione del pool

- se disponibile, viene **riutilizzato** un thread che ha terminato l'esecuzione di un task precedente.
- se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina
- **elasticità**: *“un pool che può espandersi all'infinito, ma si contrae quando la domanda di esecuzione di task diminuisce”*

CACHEDTHREADPOOL

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class ExampleCached{

    public static void main(String[] args) {

        ExecutorService service = Executors.newCachedThreadPool();

        for (int i =0; i<100; i++) {
            service.execute(new Task(i));  sleep(1000); }

        System.out.println("ThreadName:"+Thread.currentThread().getName());
    }

    private static void sleep(long timeMillis) {
        try {
            Thread.sleep(timeMillis);
        } catch (InterruptedException e) {}}}
```

OUTPUT DEL PROGRAMMA

```
pool-1-thread-11: Task 10: Starting during 5 seconds
pool-1-thread-100: Task 99: Starting during 5 seconds
Thread Name:main
pool-1-thread-99: Task 98: Starting during 7 seconds
pool-1-thread-98: Task 97: Starting during 7 seconds
pool-1-thread-97: Task 96: Starting during 6 seconds
pool-1-thread-96: Task 95: Starting during 6 seconds
pool-1-thread-95: Task 94: Starting during 9 seconds
pool-1-thread-94: Task 93: Starting during 2 seconds
pool-1-thread-93: Task 92: Starting during 3 seconds
pool-1-thread-92: Task 91: Starting during 0 seconds
pool-1-thread-92: Task Finished 91
pool-1-thread-91: Task 90: Starting during 8 seconds
pool-1-thread-90: Task 89: Starting during 6 seconds
pool-1-thread-89: Task 88: Starting during 6 seconds
pool-1-thread-88: Task 87: Starting during 1 seconds
pool-1-thread-87: Task 86: Starting during 3 seconds
pool-1-thread-86: Task 85: Starting during 7 seconds
pool-1-thread-85: Task 84: Starting during 7 seconds
pool-1-thread-84: Task 83: Starting during 7 seconds
pool-1-thread-83: Task 82: Starting during 4 seconds
pool-1-thread-82: Task 81: Starting during 8 seconds
```

attivato un nuovo thread per ogni nuovo
task

DISTANZIARE I TASK

```
pool-1-thread-1: Task 0: Starting during 3 seconds
pool-1-thread-1: Task Finished 0
pool-1-thread-1: Task 1: Starting during 7 seconds
pool-1-thread-1: Task Finished 1
pool-1-thread-1: Task 2: Starting during 0 seconds
pool-1-thread-1: Task Finished 2
pool-1-thread-1: Task 3: Starting during 3 seconds
pool-1-thread-1: Task Finished 3
pool-1-thread-1: Task 4: Starting during 5 seconds
pool-1-thread-1: Task Finished 4
pool-1-thread-1: Task 5: Starting during 5 seconds
pool-1-thread-1: Task Finished 5
pool-1-thread-1: Task 6: Starting during 9 seconds
pool-1-thread-1: Task Finished 6
pool-1-thread-1: Task 7: Starting during 6 seconds
pool-1-thread-1: Task Finished 7
pool-1-thread-1: Task 8: Starting during 1 seconds
pool-1-thread-1: Task Finished 8
pool-1-thread-1: Task 9: Starting during 1 seconds
pool-1-thread-1: Task Finished 9
pool-1-thread-1: Task 10: Starting during 0 seconds
```

.....

- cosa accade se distanzio la
sottomissione dei task ai thread, ad
esempio inserendo una sleep, nel for,
dopo la execute?
- ora viene utilizzato sempre il thread-1
per tutti i task

ASSIGNMENT 1

- *non è tutto oro quello che luccica... ovvero, non sempre il multithreading è conveniente....*
- scrivere un'applicazione JAVA che
 - crea e attiva n thread.
 - ogni thread esegue esattamente lo stesso task, ovvero conta **il numero di interi minori di 10,000,000 che sono primi**
 - il numero di thread che devono essere attivati e mandati in esecuzione viene richiesto all'utente, che lo inserisce tramite la CLI (Command Line Interface)
- analizzare come varia il tempo di esecuzione dei thread attivati a seconda del loro numero
- sviluppare quindi un programma in cui si creano n task, tutti eseguono la computazione descritta in precedenza e vengono sottomessi a un threadpool di dimensione prefissata

ASSIGNMENT 1

- `System.currentTimeMillis ()`: per misurare le prestazioni dei programmi
- restituisce il numero di millisecondi trascorsi dalla mezzanotte del 1 Gennaio, 1970

```
long tStart = System.currentTimeMillis ();  
/*  
CODE TO BE PROFILED HERE  
*/  
long tEnd= System.currentTimeMillis ();  
System.out.println ("Elapsed time (ms) " + (tEnd - tStart));
```

ESERCIZI DI RIPASSO: AZIENDA

- considerare un'azienda nella cui organizzazione sono coinvolte diverse persone, con i seguenti diversi ruoli
 - impiegati livello 1: hanno uno stipendio mensile base
 - impiegati livello 2: ottengono un bonus da aggiungere allo stipendio mensile base degli impiegati di primo livello
- lavoratori a ore: vengono pagati una cifra standard per ogni ora lavorata
- volontari: non percepiscono alcuna paga
- tutte le persone coinvolte nella organizzazione sono caratterizzate dal nome, l'indirizzo ed il numero di telefono.
- per tutti gli impiegati di livello 1 e di livello 2 livello e per i lavoratori a ore viene registrato anche il codice fiscale e un campo che riporta lo stipendio base per gli impiegati di livello 1 e di livello 2 e la paga oraria per i lavoratori a ore

ESERCIZIO DI RIPASSO: AZIENDA

- si scriva un programma JAVA che
 - crei uno staff contenente un numero prefissato (dato in input) di lavoratori per ognuno dei tipi precedenti, impostando, in fase di creazione, lo stipendio mensile standard per gli impiegati di livello 1 e 2, il bonus per quelli di livello 1 e la cifra oraria per i lavoratori a ore
 - quindi calcoli e stampi la paga per tutte le persone coinvolte nell'azienda, stampando per i volontari la stringa “Grazie!”, invece dello stipendio
 - si deve considerare il valore dello stipendio mensile, calcolato in base ai valori registrati in fase di creazione, e alle ore lavorate per i lavoratori a ore
- il programma
 - deve essere strutturato in un insieme di classi organizzate gerarchicamente
 - deve utilizzare il polimorfismo per la definizione del metodo che calcola lo stipendio di ogni membro dell'organizzazione

ESERCIZI DI RIPASSO: CENTRO DI CALCOLO

- in un centro di calcolo le singole **unità di calcolo** che eseguono tasks sono inserite in slot, che vengono identificati univocamente in base a tre numeri
 - la fila
 - posizione relativa della colonna di unità nella fila
 - posizione relativa dell'unità all'interno di tale colonna (e.g. terza fila della stanza, quarta colonna della fila, dodicesima unità nella colonna).
- il coordinatore del centro deve mantenere una struttura dati efficiente per l'assegnazione e rimozione di task di ogni unità, con gestione FIFO in cui il task più vecchio viene completato e rimosso per primo.

ESERCIZIO DI RIPASSO: CENTRO DI CALCOLO

- si definisca una classe generica per la gestione di tuple ordinabili di lunghezza arbitraria e contenenti tipi qualsiasi ma che supportano un ordinamento.
- oltre a supportare l'ordinamento la classe offre getters (getters di tipo non primitivo) e setters per accedere o modificare l'elemento i-esimo.
- si utilizzi la classe delle tuple per individuare univocamente le unità.
- si rappresenti ogni task attraverso una stringa di lunghezza fissata di 16 caratteri.
- si scelga poi un'opportuna combinazione di collections per l'assegnamento e rimozione efficiente di task ad ogni unità.
- L'utente invia una richiesta (aggiunta, rimozione o visione del task corrente) specificando l'unità attraverso la propria tupla e il task stesso come stringa (solo per l'inserimento).
- si definisca un nuovo tipo di eccezione per gestire il caso in cui si tenti di creare una nuova unità con la stessa tripla identificativa di un'unità già esistente.