Continuo esercitazione esercizi 4.x

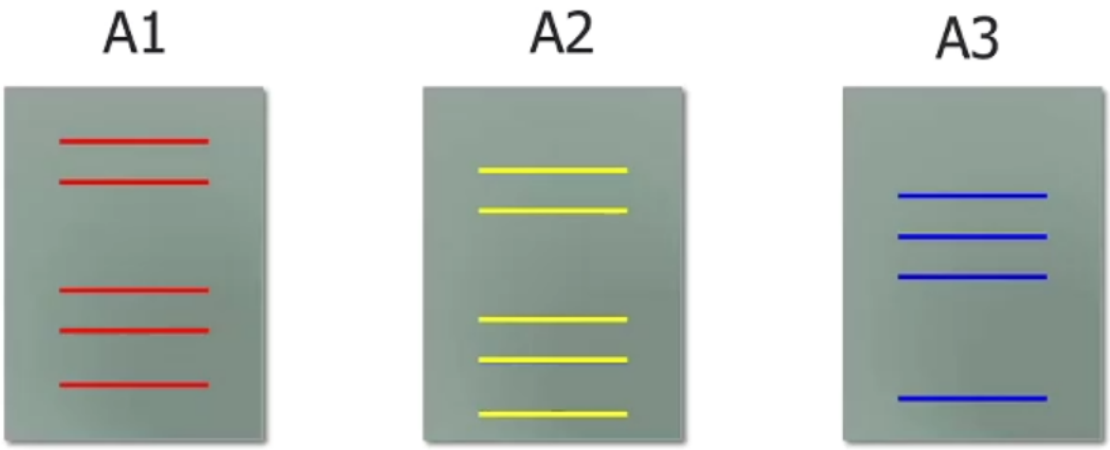
Esercizi 4.3, 4.4.

Programmazione Concorrente

Cosa è la programmazione concorrente?

Quando parliamo di programma concorrente stiamo parlando di un programma composto da molte **attività**. Ogni attività che compone un programma, esegue le sue istruzioni in maniera **sequenziale**; le istruzioni dell'intero programma, però, sono seguite secondo un ordine non noto.

Questo ci consente di dire che stiamo parlando di un **programma concorrente**.



L'ordine globale dell'esecuzione di queste istruzioni **non è noto.** Siccome diamo la possibilità a ciascuna attività di eseguire in modo indipendente, ogni attività può eseguire in modo arbitrario ogni sua istruzione.

In poche parole, stiamo dicendo che **localmente** ogni attività esegue le sue istruzioni in modo sequenziale, quindi l'istruzione 2 verrà eseguita dopo l'istruzione 1. **Globalmente**, invece, l'istruzione da eseguire viene "pescata" da una delle attività, quindi non è detto che l'istruzione A1.2 venga eseguita subito dopo A1.2, ma nel mezzo potrebbe essere eseguita (ad esempio) A2.1.

Astrazioni per la concorrenza

Computer

Una prima astrazione che possiamo introdurre è un computer: possiamo usarlo per eseguire un programma sequenziale, un altro computer per eseguire un altro programma (ecc.), avendo così un'esecuzione concorrente; i due programmi si scambiano dei messaggi, ed in questo caso abbiamo il massimo livello di autonomia, questo perchè ogni programma ha a disposizione il massimo livello di risorse del proprio computer.

Processo

Un processo è un'unità logica di esecuzione definito dal sistema operativo. Le attività sono implementate dalle unità logiche in esecuzione che comunicano attraverso dei meccanismi di **IPC** (visti nella repository di architettura dei calcolatori); tra questi meccanismi aggiungiamo anche le **socket**, che consentono di far comunicare i processi sia sulla stessa macchina che su macchine diverse.

Grazie al sistema operativo i processi sono in grado di condividere le risorse della macchina ospitante, come la memoria.

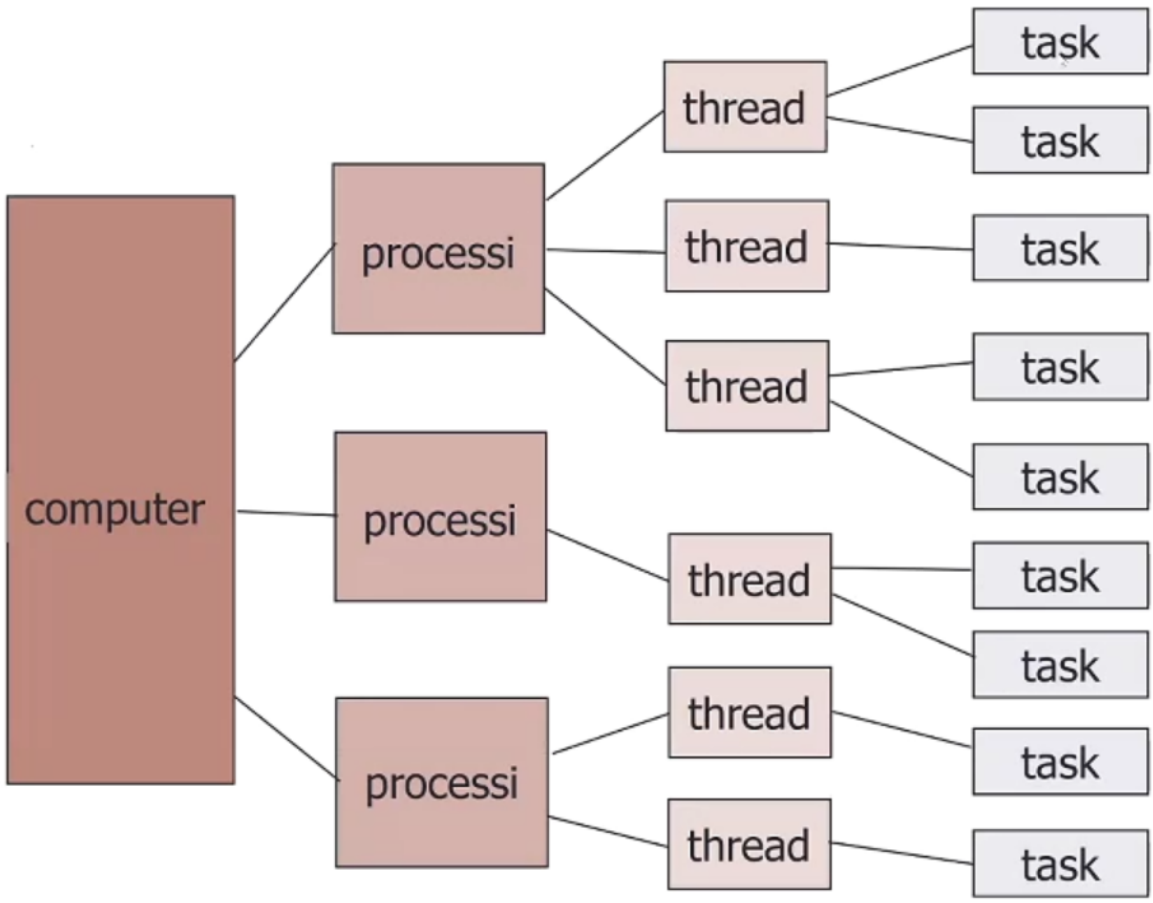
Thread

I thread comunicano sfruttando la memoria condivisa all'interno di un processo; è anche possibile prevedere thread in diversi processi, ed in questo caso i thread comunicano grazie agli stessi meccanismi che i processi usano, ovvero i vari meccanismi di IPC.

Task

E' un'unità logica di esecuzione che **"viva all'interno di un thread"**: un thread potrebbe ospitare più task. Anche in quest caso possiamo pensare ai task che comunicano grazie alla memoria condivisa del processo che ospita il thread, oppure usare meccanismi di IPC per far comunicare task che sono in processi diversi.

Vista grafica



Quindi?

Il concetto di **concorrenza** va completamente disaccoppiato dal concetto di processo e thread; questo perchè processi e thread sono **possibili implementazioni** del concetto di attività.

Il concetto di **parallelismo** e **concorrenza** sono due concetti diversi: la concorrenza abilita il parallelismo, ma non è detto che implementando un'applicazione usando la programmazione concorrente si abbia la possibilità di eseguire in parallelo.

Per trasformare la concorrenza in parallelismo abbiamo bisogno di **risorse hardware**.

Parallelismo

Se abbiamo a disposizione, all'interno della nostra macchina, più **unità di elaborazione**, ovvero di **processing units (CPU)**, allora le attività che avevamo in precedenza, possono avere delle unità fisiche di elaborazione dedicate. In questo caso, se adesempio abbiamo 4 cores diversi, possiamo dividere le attività in 4 gruppi diversi e mandare in esecuzione un gruppo per ogni core, in modo da avere (teoricamente) l'esecuzione simultanea di 4 attività allo stesso momento.

Se creiamo più attività di quanti core fisici disponiamo, ovviamente queste attività non verranno eseguite **tutte** allo stesso momento; il caso limite si verifica quando abbiamo un processore con **un unico core**, e di conseguenza non possiamo avere il parallelismo.

Preemption

Questo concetto è stato visto nella repo di architettura dei calcolatori.

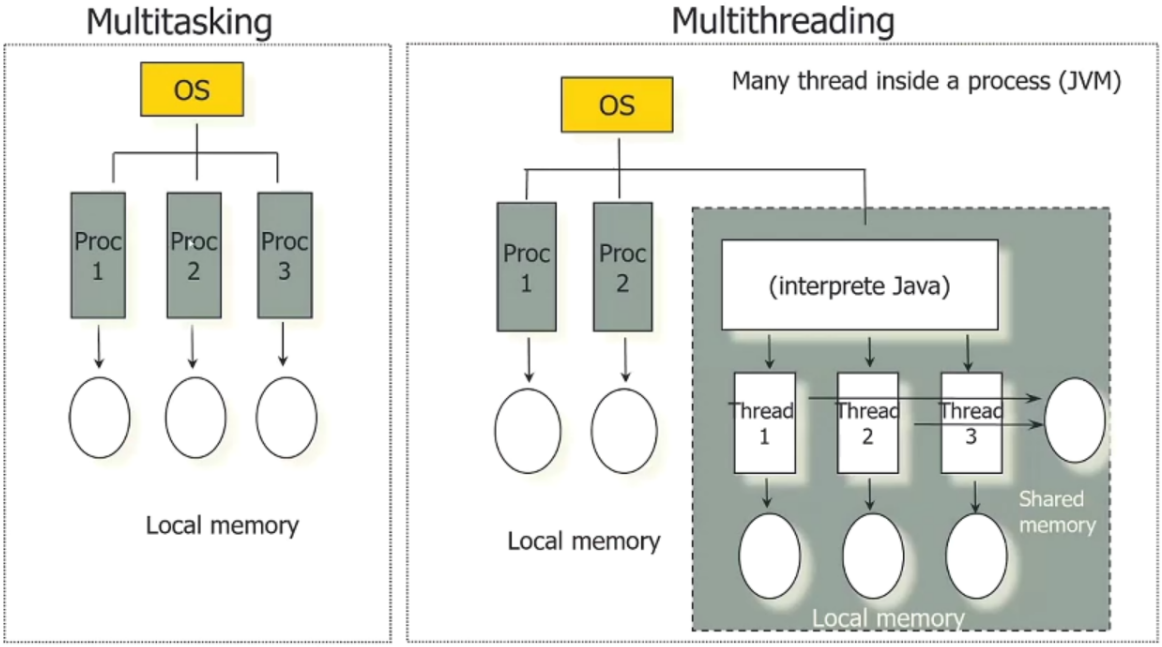
Sistema Preemptive

Quando il time slice di un processo termina, l'attività in esecuzione viene sospesa, e la CPU viene assegnata ad un altro processo.

Sistema non Preemptive

Un'attività in esecuzione viene sospesa solo volontariamente (termina) o perchè essa è in attesa di un evento I/O. Non è possibile sottrarre la CPU al processo.

Programmazione concorrente in Java



A sinistra vediamo la rappresentazione del **multitasking**, dove eseguiamo più processi allo stesso momento; il modello di concorrenza è basato sull'impiego dei processi come attività concorrente. Ogni processo ha la sua memoria e comunicano tra loro tramite IPC.

A destra vediamo l'espansione di uno di questi processi, che esegue la macchina virtuale java (JVM); all'interno del processo troviamo delle attività (**thread**) che "vivono" all'interno del processo, e comunicano tra loro grazie alla memoria condivisa del processo stesso.
Questo tipo di rappresentazione logica corrisponde alla rappresentazione logica di una macchina multi-processore: se sostituiamo ogni thread con un processore, otteniamo esattamente lo schema di una macchina multi-processore. Quindi ogni processore ha la sua memoria locale (cache) che usa per effettuare operazioni più veloci, e comunica con una memoria condivisa (RAM), utilizzata anche dagli altri processori (in questo caso da altri thread del processo).

Thread dal punto di vista della programmazione

In java un thread è un'istanza di una classe (package java.lang.Thread); questa classe mette a disposizione i metodi che consentono di gestire il ciclo di vita di un thread:

- **start()**: usato per creare un contesto di esecuzione, per trasformare un oggetto passivo in un oggetto "attivo", ovvero dotato di un proprio contesto di esecuzione. Solo all'atto dell'invocazione di questo metodo, l'oggetto istanza della classe **Thread** "prende vita". All'atto pratico stiamo creando il contesto di esecuzione dell'oggetto, in modo che operi come thread e non più come oggetto (statico).
- **run()**: in questo metodo specifichiamo le istruzioni che vogliamo siano eseguite nel contesto di esecuzione di quel thread specifico.

Ad esempio: se vediamo l'immagine vista prima, per creare i tre thread dobbiamo creare **tre istanze** della classe **Thread**, e su queste tre istanze dobbiamo invocare il metodo **start()**. In questo modo i thread iniziano ad eseguire le istruzioni **previste con il metodo run()**.

Classe thread

```
public class Thread implements Runnable{
    private Runnable target;

    public synchronized void start(){
```

```
        ... start() ...
    }

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

    // le funzioni thread native sono usate per creare un nuovo contesto di
esecuzione
    private native void start(){
        pthread_create(&tid, NULL, run(), NULL);
    }

    public Thread(Runnable t){
        target = t;
    }
}
```

La classe Thread implementa **Runnable**; all'interno troviamo una variabile di tipo Runnable chiamata target. Questa classe mette a disposizione il metodo start(); solo nel momento in cui invochiamo questo metodo l'oggetto diventa un **thread vero e proprio**, perchè gli stiamo creando il contesto di esecuzione. Quando invochiamo start(), diciamo anche che vogliamo eseguire le istruzioni previste con il metodo run().

```
public interface Runnable{
    void run();
}
```

L'interfaccia Runnable prevede un unico metodo: run().

🏁 1:15

Quando invochiamo start(), stiamo effettivamente invocando un cosiddetto **metodo nativo**, che ci dice che il metodo ha l'implementazione nativa, in linguaggio c. Stiamo delegando una libreria esterna alla gestione dei thread!

Attributi Thread

Tutte le macchine virtuali java implementano i propri thread sfruttando i thread nativi; questo perchè c'è una differenza tra **thread nativi e thread di spazio utente**, già vista nella repo di Architettura dei Calcolatori. Il vantaggio di usare dei thread nativi è quello di poter sfruttare il parallelismo perchè il SO è in grado di *vedere* i thread, e quindi sfruttare l'hardware.

Ogni thread è composto da:

- **Corpo**
- **Stato**
- **Priorità**
- **Gruppo**

Corpo

Il corpo di un thread è l'insieme delle istruzioni che vogliamo che il thread esegua; sono le istruzioni che dobbiamo prevedere all'interno del metodo run(); abbiamo due diversi approcci per la definizione del corpo di un thread:

1. Specializziamo la classe Thread andando ad estenderla sovrascrivendo il metodo run()
2. Implementiamo l'interfaccia Runnable.

1) Thread a sottoclasse

```
class SimpleThread extends Thread{
    public SimpleThread(string str){
        super(str); // la stringa viene usata per dare un nome alla classe di thread.
    }

    @override
    public void run(){  // il metodo run sovrascrive il metodo run della classe Thread
        for(int i = 0; i < 10; i++){    // è eseguito un ciclo; stampa su stdout l'indice ed il nome della classe
            Sout.println(i + " " + getName());
        }
        Sout.println("DONE!" + getName()); // alla fine del ciclo stampiamo done ed il nome del thread
    }
}
```

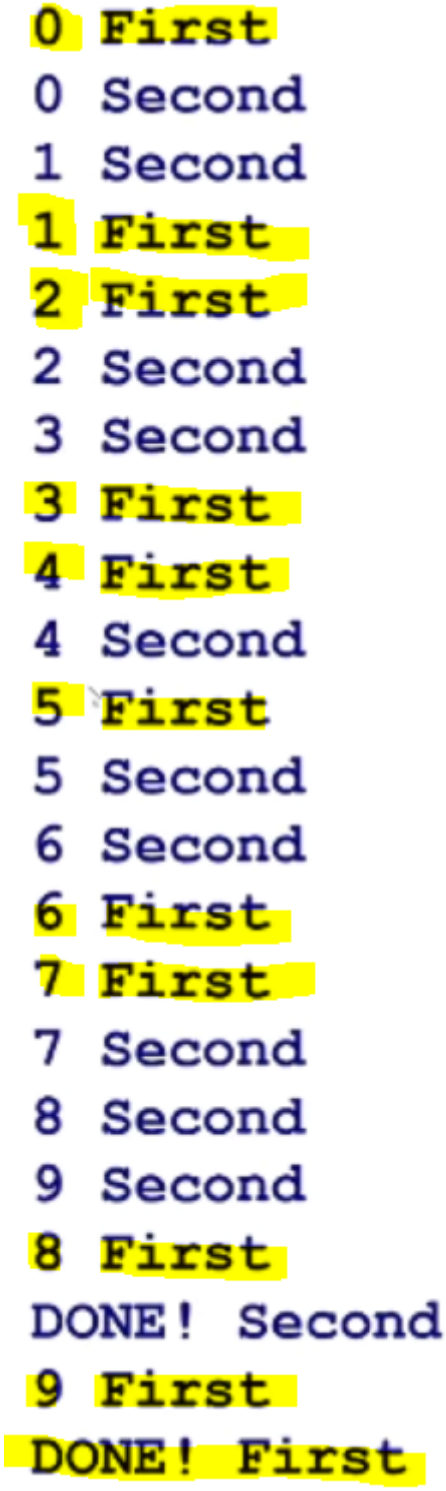
Vediamo quindi che il metodo run() ad eseguire le istruzioni che vogliamo vengano eseguite nel thread. Per far si che questo oggetto diventi un vero e proprio Thread, dobbiamo invocare su di esso il metodo run():

```
class TwoThreadsTest{
    public static void main(){
        // creiamo due thread
        SimpleThread ts1 = new SimpleThread("primo");
        SimpleThread ts2 = new SimpleThread("secondo");

        // per avviare i thread invochiamo il metodo start()
        ts1.start();
        ts2.start();
    }
}
```

Quando invochiamo start() su ts1, il thread 1 diventa un vero e proprio thread, perchè gli stiamo creando il contesto di esecuzione.

E l'output? Non possiamo sapere in che ordine verranno eseguite le istruzioni, perchè l'ordine è deciso dallo scheduler. Se però filtriamo l'output per il nome del thread, vediamo che le istruzioni **di quel thread** vengono eseguite in ordine:



2) Implementare Runnable

L'altra possibilità è quella di **implementare l'interfaccia Runnable**: prevediamo la possibilità di scrivere il codice che vogliamo sia eseguito attraverso l'implementazione del metodo run().

```
class PrimeRun extends Math implements Runnable{
    private long minPrime;

    public PrimeRun(long minPrime){
        this.minPrime = minPrime;
    }

    public void run(){
        //operazioni
    }
}
```

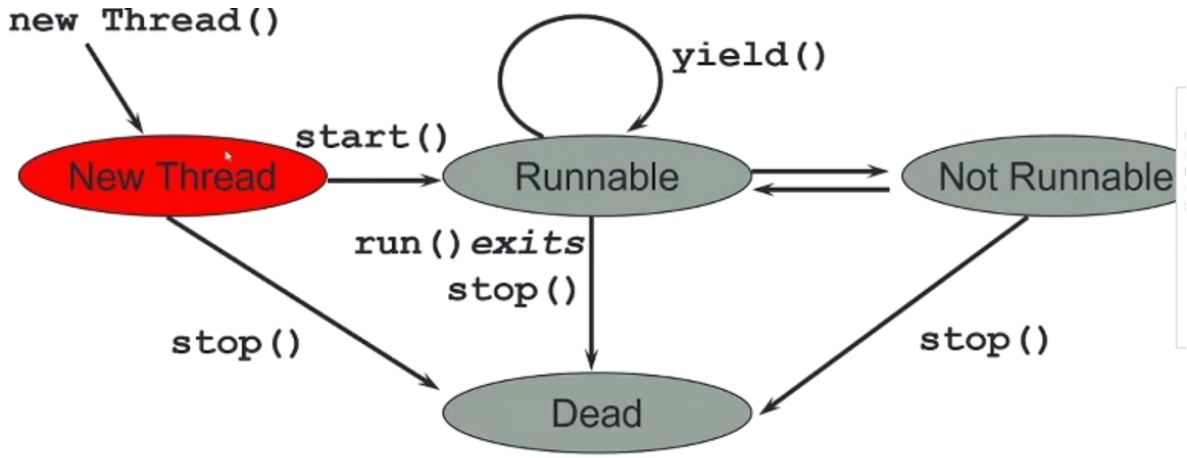
Una volta che abbiamo scritto una classe di tipo Runnable, per dar vita al thread dobbiamo necessariamente creare un'istanza della classe:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Creiamo quindi un Thread passandogli la nostra istanza di classe che contiene il metodo run (referenziata da p); successivamente invochiamo sul Thread il metodo start().

Quindi Siccome in Java non possiamo ereditare più di una classe, se abbiamo bisogno di ereditare del codice per semplificare l'implementazione del metodo run, torniamo al secondo approccio ed estendiamo Runnable.

Ciclo di vita di un thread



New Thread -> Runnable

Quando invochiamo start() sull'oggetto passivo Thread, abbiamo la transizione allo stato **runnable.**

A questo punto il thread ha un suo contesto di esecuzione ed è in grado di eseguire le istruzioni previste dal metodo run() ed è schedulabile per l'esecuzione. **Perchè chiamiamo questo stato runnable e non running?** Semplicemente perche il thread è **pronto** per l'esecuzione, ma non lo è ancora. Affinchè il thread entri nello stato running, deve attendere che lo scheduler lo scheduli per l'esecuzione.

Runnable -> not Runnable

Quali sono gli eventi che fanno passare il thread da Runnable a not Runnable? Nello stato not runnable il thread è bloccato, e non può eseguire. Ad esempio, l'invocazione esplicita di **suspend()** sul thread lo fa passare allo stato **not runnable**; questo metodo sospende il thread finchè non ci sarà un'operazione duale effettuata sullo stesso thread che consente di risvegliarlo. Un'altra possibilità è che sul thread venga invocato il metodo **sleep(*time in ms*)**; questo metodo sospende il thread per un intervallo di tempo passato come parametro.
Possiamo inoltre invocare **wait()** sul thread.

Not Runnable -> Runnable

Dallo stato not runnable accade spesso che i thread si spostino tra i due stati runnable e not runnable. il thread torna runnable quando su di esso viene invocato **resume()** (dopo aver invocato suspend()). Potrebbe inoltre essere scaduto l'invervallo di tempo che avevamo specificato con **sleep()**.

Potrebbe essere invocata l'operazione duale di **wait()** sul thread, ovvero le operazioni **notify()** e **notifyAll()**.

Stato Dead

Questo stato è lo stato in cui il thread non è più in grado di eseguire; il thread che entra in questo stato rimane comunque un oggetto utilizzabile (come tutti gli altri oggetti), l'unica differenza è che non è più schedulabile come thread.

Il thread entra in uno stato Dead nel momento in cui viene invocato il metodo **stop()**; questo metodo è sconsigliato, perchè l'invocazione potrebbe provocare la "morte" del thread in uno stato inconsistente degli oggetti su cui stava lavorando (viene quindi ucciso mentre stava lavorando su un oggetto).
Utilizzo di stop: `myThread.start() - myThread.stop()`

L'altra modalità per far terminare un thread è quella di far terminare il **metodo run()**: quando il metodo run (ovvero le istruzioni che abbiamo previsto all'interno del metodo run) termina, il thread termina.

Fine lezione 12