

CORSO JAVA EE

Indice generale

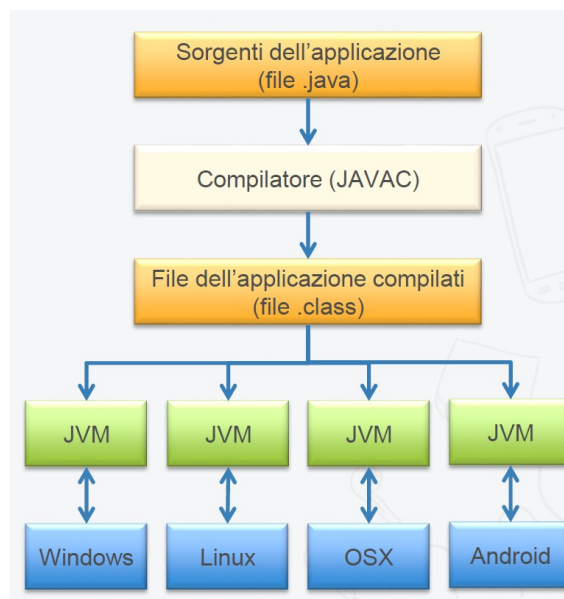
INTRODUZIONE A JAVA.....	3
JAVA PLATFORM.....	4
VANTAGGI E SVANTAGGI JAVA.....	4
JRE, JDK E AMBIENTI DI SVILUPPO.....	5
JAVA RUNTIME ENVIRONMENT (JRE).....	5
JAVA DEVELOPMENT KIT (JDK).....	5
AMBIENTI DI SVILUPPO CON INTERFACCIA GRAFICA.....	5
TIPI DI SOFTWARE.....	6
STANDALONE.....	6
CLIENT / SERVER.....	6
APPLICAZIONE WEB.....	6
SVILUPPO SOFTWARE: WATERFALL VS AGILE.....	7
METODOLOGIA WATERFALL.....	7
METODOLOGIA AGILE.....	8
COMPILAZIONE ED ESECUZIONE DEL PROGRAMMA.....	9
FILE JAR.....	9
FILE MANIFEST.....	9
THREAD E CONCORRENZA.....	10
PROCESSO.....	10
THREAD.....	10
CONCORRENZA.....	12
CICLO DI VITA DI UN THREAD.....	13
THREAD PRIORITY.....	14
CREARE UN THREAD IN JAVA.....	15
MULTITHREADING.....	16
CONCORRENZA IN JAVA.....	19
CONCORRENZA CON L'UTILIZZO DEI THREAD.....	19
CONCORRENZA CON L'UTILIZZO DEGLI EXECUTOR.....	22
CONCORRENZA CON L'UTILIZZO DEL FRAMEWORK FORK/JOIN.....	24
SINCRONIZZAZIONE.....	26
ESEMPIO MULTITHREADING E CONCORRENZA : PRODUCER-CONSUMER.....	31
METODI WAIT(), NOTIFY(), NOTIFYALL().....	35
WAIT().....	35
NOTIFY() E NOTIFYALL().....	36
SINCRONIZZAZIONE AVANZATA CON LOCK E REENTRANTLOCK.....	36
LOCK.....	36
REENTRANTLOCK.....	37
UTILIZZARE IL BLOCCO TRYFINALLY CON I THREAD.....	38
THREAD POOL.....	40
CLASSI ARRAYBLOCKINGQUEUE E LINKEDBLOCKINGQUEUE.....	43
CLASSE ARRAYBLOCKINGQUEUE.....	44
CLASSE LINKEDBLOCKINGQUEUE.....	46
GESTIONE BANCONC SALUMI MEDIANTE THREAD POOL.....	47
DEADLOCK, STARVATION E LIVELOCK.....	51
DEADLOCK.....	51
STARVATION.....	51
LIVELOCK.....	52

INTRODUZIONE A JAVA

JAVA è un linguaggio di programmazione object oriented (orientato agli oggetti). La programmazione a oggetti è un paradigma ancora oggi molto utilizzato, perché ci consente di modellare al meglio delle situazioni reali.

La sintassi del linguaggio JAVA è molto simile ai linguaggi C e C++, da cui eredita parecchie caratteristiche. Per certi versi, però, JAVA è più semplice da utilizzare rispetto i due linguaggi citati precedentemente.

La caratteristica principale che ha reso JAVA così popolare è la portabilità, cioè l'essere indipendente dal sistema operativo su cui viene eseguito un software. Gli elementi fondamentali che rendono JAVA un linguaggio di programmazione portabile sono la Java Virtual Machine (Macchina virtuale o JVM) e la Java Platform.



Appunto, portabilità vuol dire scrivere il proprio codice sorgente una sola volta, compilarlo ed eseguirlo su qualsiasi dispositivo dotato di una macchina virtuale. Un programma JAVA è rappresentato da uno o più file.java, al cui interno sono presenti i nostri sorgenti, ovvero il codice che scriviamo. Attraverso il compilatore chiamato JAVAC, il nostro codice sorgente viene tradotto in un linguaggio intermedio, chiamato bytecode. Tutti questi file compilati avranno l'estensione .class e potranno essere interpretati dalla JVM, che è un processore virtuale che legge il/i nostro/i file.class e traduce il bytecode al loro interno in linguaggio macchina. Quindi, se su dispositivi differenti, con sistema operativo differente, abbiamo la stessa JVM, possiamo eseguire il nostro software senza necessità di ricompilarlo ogni volta.

Oggi JAVA è utilizzato per scrivere:

- applicazioni web: La maggior parte delle applicazioni enterprise (applicazioni aziendali) utilizzano JAVA, soprattutto nella parte backend;
- applicazioni per smartphone e tablet: Android, per esempio, è un sistema operativo scritto in JAVA, così come tutte le app scritte per questo sistema operativo;
- applicazioni per decoder digitali, elettrodomestici e così via.

JAVA PLATFORM

La Java platform è una piattaforma composta da due componenti:

- JVM, di cui abbiamo già parlato precedentemente;
- API (Application Programming Interface), cioè un set di librerie (componenti software) messi a disposizione degli sviluppatori per poter scrivere software JAVA.

La Java Platform è disponibile in 3 configurazioni:

- Java Standard Edition (JSE): mette a disposizione il set standard di API per poter scrivere applicazioni standalone (programma che può funzionare senza che siano richiesti altri componenti o addirittura senza sistema operativo), client e server, per accesso a database, per il calcolo scientifico e così via;
- Java Enterprise Edition (JEE): mette a disposizione, oltre alle API della Standard Edition, anche quelle per scrivere applicazioni distribuite (ad esempio applicazioni web);
- Java Micro Edition (JME): mette a disposizione le API per sviluppare applicazioni mobile.

VANTAGGI E SVANTAGGI JAVA

Vantaggi

- indipendenza del linguaggio bytecode: consente di eseguire lo stesso programma su più dispositivi dotati di JVM;
- velocità di sviluppo;
- grande disponibilità di librerie;
- alta integrazione con il web.

Svantaggi

- velocità di esecuzione: il programma viene eseguito ed elaborato dalla JVM, che a sua volta traduce le istruzioni in linguaggio macchina. Pertanto il tempo di esecuzione è leggermente più lento rispetto ad un programma scritto in C++;
- attraverso la decompilazione è possibile risalire al codice sorgente.

JRE, JDK E AMBIENTI DI SVILUPPO

JAVA RUNTIME ENVIRONMENT (JRE)

JRE è un'implementazione della JVM, ed è necessario per l'esecuzione dei programmi JAVA.

Il JRE contiene:

- JVM.
- API standard di JAVA.
- Un launcher necessario per avviare i programmi già compilati in bytecode. Tutti i programmi partono sempre da una classe dotata di un metodo main()

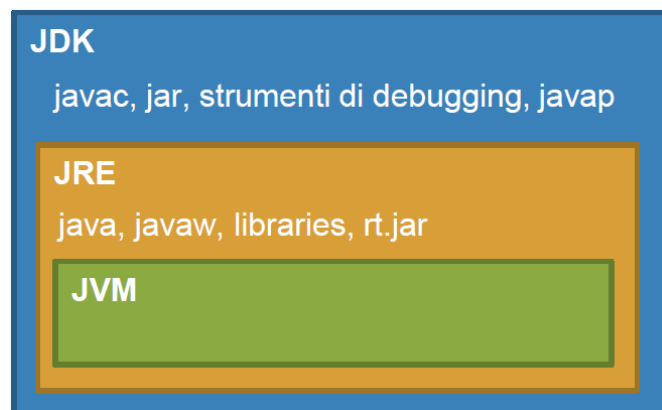
Il JRE deve essere installato su tutti i dispositivi che hanno necessità di eseguire software scritti in JAVA.

JAVA DEVELOPMENT KIT (JDK)

JDK è un insieme di librerie e software messe a disposizione da Oracle, che consentono di sviluppare software scritti in JAVA.

JDK è un ambiente di sviluppo a tutti gli effetti, poiché appunto contiene tutte queste librerie che ci consentono di sviluppare software, ma è un ambiente di sviluppo a console, ovvero non è dotato di un'interfaccia grafica, ma le istruzioni si eseguono mediante il prompt di comandi. Gli ambienti di sviluppo più famosi, dotati di interfaccia grafica, sono Eclipse, Netbeans e IntelliJ IDEA.

Il JDK contiene librerie importanti per lo sviluppo, il JRE e la JVM.



AMBIENTI DI SVILUPPO CON INTERFACCIA GRAFICA

Un ambiente di sviluppo integrato o IDE (Integrated Development Environment), rispetto al JDK, è un software dotato di interfaccia grafica che consente agli sviluppatori di creare software JAVA, semplificando la programmazione e la gestione dei file. È utile perché:

- consente di segnalare e visualizzare subito gli errori di sintassi all'interno del codice;
- consente di effettuare il debug in maniera semplice;
- offre una serie di strumenti e funzionalità di supporto allo sviluppatore.

TIPI DI SOFTWARE

STANDALONE

Il software standalone è un software che è installato all'interno di un sistema operativo ed è autonomo, perché non richiede l'uso di particolari componenti esterne con cui interagire (per esempio i server).

Esempi di software standalone sono:

- il pacchetto di software Microsoft (Word, Excel, ...);
- i software Adobe (Photoshop, InDesign, ...);
- alcune app per dispositivi mobile (ad es. l'app che visualizza le foto, l'app che gestisce i file del device).

CLIENT / SERVER

Il software client/server è composto dai due componenti omonimi che definiscono il suo nome. La componente client è installata sul nostro dispositivo personale, mentre la componente server si trova su un dispositivo remoto e fornisce un servizio al client.

Esempi di sistemi client/server:

- File server: per la condividere i file;
- FTP server: per l'upload/download dei file;
- Database server: per la gestione dei dati;

Questa architettura, però, ha un limite importante. Questo limite riguarda la necessità di installare il software della componente client su ciascun terminale che deve accedere a questo software. Questo problema si risolve con le applicazioni web.

APPLICAZIONE WEB

L'applicazione web è nata con lo scopo di migliorare l'utilizzo dei componenti client, senza la necessità di dover installare nuovi software. Un'applicazione web è accessibile mediante un normale browser.

L'applicazione web è un tipo di client/server evoluto, in cui è sostanzialmente il browser a fare da client, che scarica per noi tutte le varie informazioni (le pagine web) che ci servono per interagire con la componente server. In questo caso, il server e il client comunicano mediante protocollo HTTP.

Per accedere alle applicazioni web si utilizzano URL o link ipertestuali.

SVILUPPO SOFTWARE: WATERFALL VS AGILE

Dopo aver visto i vari tipi di software, adesso vediamo come si sviluppa un software. Non ha senso parlare di sviluppo nel momento in cui abbiamo a che fare con degli script o piccoli software che devono svolgere un compito particolare, ma solo nel momento in cui abbiamo a che fare con software complessi o strutturati per la gestione di un'intera azienda.

Usare una metodologia per lo sviluppo di un software complesso è importante, perché consente di realizzare il software in maniera più organizzata e strutturata, limitando gli errori che uscirebbero fuori nel caso di uno sviluppo senza criterio.

Le metodologie per lo sviluppo di un software sono 2: waterfall e agile.

METODOLOGIA WATERFALL

Nella metodologia waterfall (o classica) la sequenza delle fasi del ciclo di vita di un progetto software è strettamente sequenziale e, prima di passare alla fase successiva, è necessario che quella corrente sia terminata completamente.

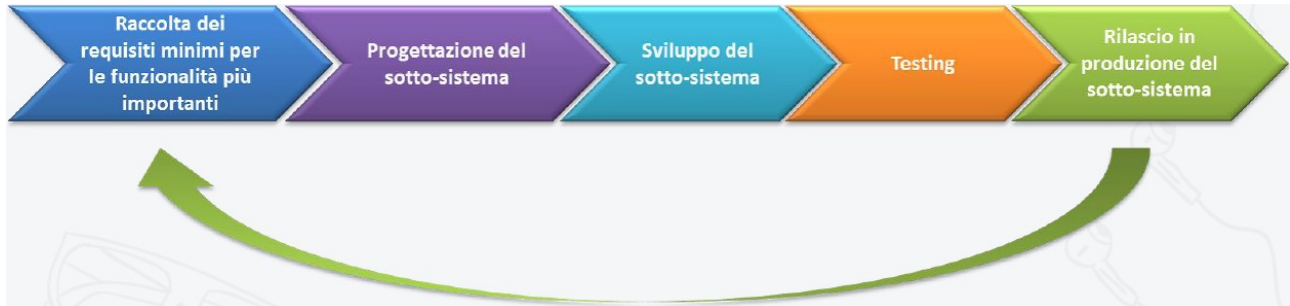


Quindi, in questo tipo di approccio raccogliamo i requisiti all'inizio, sulla base di quei requisiti viene sviluppato il software e, infine, facciamo vedere il nostro software completo solo al termine del rilascio in produzione.

Il limite principale di questa metodologia è quello di aver realizzato un prodotto che soddisfa perfettamente i requisiti inizialmente raccolti che, però, nel frattempo possono essere cambiati e quindi il software è diverso da quello atteso.

METODOLOGIA AGILE

Nella metodologia agile abbiamo le stesse fasi della metodologia waterfall, ma il ciclo di vita di un progetto software è visto come una sequenza di tante iterazioni, dove ogni iterazione è un arco temporale che va dalle 2 alle 4 settimane.



In ciascuna iterazione prendiamo un pezzettino del nostro software sul quale vengono eseguite tutte le fasi, dopodiché si prende un altro pezzettino e si ritorna indietro nelle fasi per rieseguirle tutte e così via, fino alla realizzazione di tutti i pezzettini che comporranno il software finale.

Vantaggi di questa metodologia:

- coinvolgimento attivo del committente e degli utenti nel processo di sviluppo;
- aggiornamenti regolari e frequenti sullo stato dell'applicazione;
- validazione continua dei requisiti (dopo ogni iterazione);
- consegna rapida delle funzionalità di base;
- pianificazione fissa dei tempi di consegna per funzionalità;
- maggiori test, software migliore.

COMPILAZIONE ED ESECUZIONE DEL PROGRAMMA

La compilazione è quel processo che consente di trasformare il codice sorgente, che si trova all'interno dei nostri file.java, in un linguaggio intermedio chiamato bytecode, che è il linguaggio interpretato dalla JVM.

Tutti i file compilati si trovano all'interno di file che hanno estensione .class . Quindi, a partire dal nostro sorgente .java, tramite la compilazione viene creato uno o più file .class .

Il tool messo a disposizione dalla JDK per compilare il nostro software è il tool javac.

FILE JAR

Un JAR è un file compresso (archivio o zip) che contiene al suo interno una serie di file compilati del nostro software, ed eventuali librerie aggiuntive. I vantaggi dell'uso del JAR sono:

- Compressione, perché i file vengono compressi in un unico file, quindi la dimensione complessiva del programma diminuisce.
- Firma, perché il file JAR contiene una cartella META-INF, che contiene a sua volta il file MANIFEST.MF. Questo file consente di identificare al meglio il nostro software, ossia identificare per esempio il Class-Path, il Main-Class, l'autore e così via.
- Portabilità, perché lo stesso JAR può essere eseguito su diversi sistemi operativi che contengono la JVM richiesta per l'esecuzione.

FILE MANIFEST

Alcuni attributi del file MANIFEST sono:

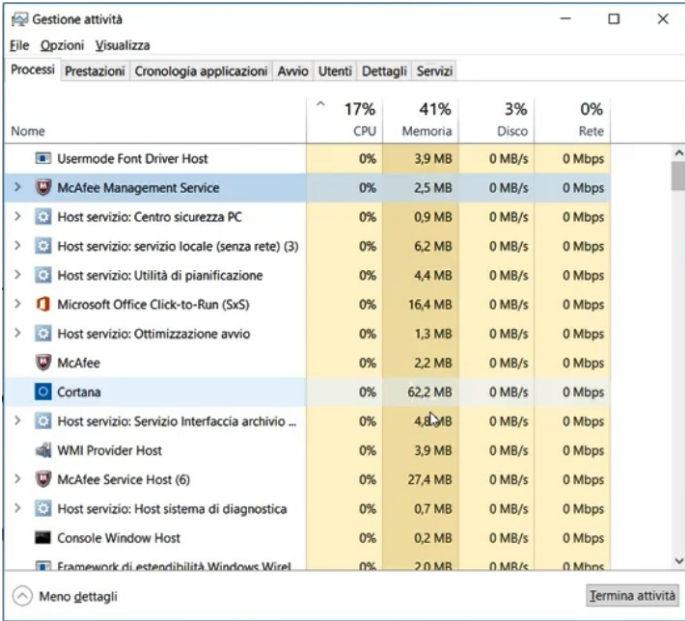
- Manifest-Version: definisce appunto la versione del file MANIFEST.
- Created-By: definisce la versione e il vendor del JDK utilizzato per creare il file.
- Class-Path: definisce il percorso delle librerie che sono necessarie per eseguire il software.
- Main-Class: contiene la classe che ha il metodo main() per avviare il nostro software.

THREAD E CONCORRENZA

PROCESSO

Prima di capire cosa è un thread, dobbiamo definire cosa è un processo.

Quando installiamo su un dispositivo un software, copiamo i file compilati del software stesso all'interno del dispositivo, indipendentemente dal tipo di sistema operativo che vi è installato sopra. Il software in stato di esecuzione genera un processo, che avrà un ID univoco e occuperà una porzione di RAM dove salverà le sue informazioni. Quindi, un processo può essere definito come un'istanza del software che contiene le istruzioni e i dati che vengono elaborati durante l'esecuzione del software. Un esempio lo possiamo vedere sulla gestione attività del PC, dove sono elencati tutti i processi attivi in quel dato momento:



The screenshot shows the Windows Task Manager window titled "Gestione attività". The "Processi" tab is selected. The table lists various background processes with their names, CPU usage, memory usage, disk usage, and network usage. The processes are sorted by CPU usage, with "Cortana" showing the highest usage at 17%.

Nome	17% CPU	41% Memoria	3% Disco	0% Rete
Usermode Font Driver Host	0%	3.9 MB	0 MB/s	0 Mbps
McAfee Management Service	0%	2.5 MB	0 MB/s	0 Mbps
Host servizio: Centro sicurezza PC	0%	0.9 MB	0 MB/s	0 Mbps
Host servizio: servizio locale (senza rete) (3)	0%	6.2 MB	0 MB/s	0 Mbps
Host servizio: Utilità di pianificazione	0%	4.4 MB	0 MB/s	0 Mbps
Microsoft Office Click-to-Run (SxS)	0%	16.4 MB	0 MB/s	0 Mbps
Host servizio: Ottimizzazione avvio	0%	1.3 MB	0 MB/s	0 Mbps
McAfee	0%	2.2 MB	0 MB/s	0 Mbps
Cortana	17%	62.2 MB	0 MB/s	0 Mbps
Host servizio: Servizio Interfaccia archivio ...	0%	4.8 MB	0 MB/s	0 Mbps
WMI Provider Host	0%	3.9 MB	0 MB/s	0 Mbps
McAfee Service Host (6)	0%	27.4 MB	0 MB/s	0 Mbps
Host servizio: Host sistema di diagnostica	0%	0.7 MB	0 MB/s	0 Mbps
Console Window Host	0%	0.2 MB	0 MB/s	0 Mbps
Framework di estendibilità Windows Wirel...	0%	2.0 MB	0 MB/s	0 Mbps

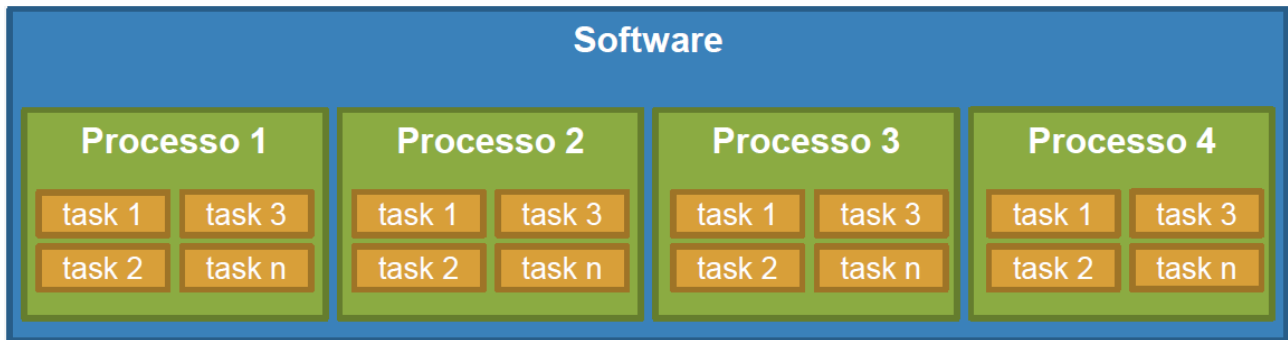
Siccome i sistemi operativi sono multitasking, possiamo eseguire più processi contemporaneamente e il sistema operativo assegnerà la CPU all'i-esimo processo, a seconda delle necessità.

THREAD

Quando abbiamo la necessità di suddividere un processo in più sottoprocessi (o task), i quali devono essere indipendenti tra loro, a quel punto parliamo di thread. Un thread è un sottoprocesso che ha vita propria e svolge un determinato compito. Più thread possono essere eseguiti in maniera concorrente, a seconda delle necessità.

Quindi, in linea generale lo scopo dei thread è quello di dividere un processo in tante piccole parti, dove ogni parte lavora in maniera autonoma e deve gestire un evento o una risorsa.

Vediamo un esempio:



In questo esempio abbiamo il nostro software e 4 processi, che sono intesi come 4 istanze dello stesso software. Il numero di processi equivale al numero di volte che l'utente avvia lo stesso software. Ogni processo avrà i suoi task per la gestione delle singole attività e dei singoli eventi.

Esempi di utilizzo dei thread sono:

Nel caso del browser web:

- un thread che si occupa di scrivere il testo e visualizzarlo a video;
- un thread che si occupa di effettuare la ricerca.

Nel caso del server web:

- un thread che si occupa di accettare le richieste e creare altri thread che le gestiscono;
- un thread che si occupa di gestire una richiesta.

Nel caso di word:

- un thread che si occupa di scrivere i comandi digitati dall'utente;
- un thread che si occupa di cercare i sinonimi e gli errori di scrittura;
- un thread che si occupa di visualizzare gli errori di ortografia.

CONCORRENZA

Ovviamente anche il sistema operativo stesso è composto da diversi software. Se il dispositivo non fosse in grado di gestire l'accesso concorrente a questi programmi, potremmo eseguire un solo programma alla volta. Ad esempio, non potremmo navigare su internet e al tempo stesso guardare un video, scrivere un documento, ascoltare musica e così via.

Si capisce a questo punto che la gestione della concorrenza diventa fondamentale, perché consente di sfruttare tutte le risorse hardware a disposizione per l'esecuzione di più software contemporaneamente.

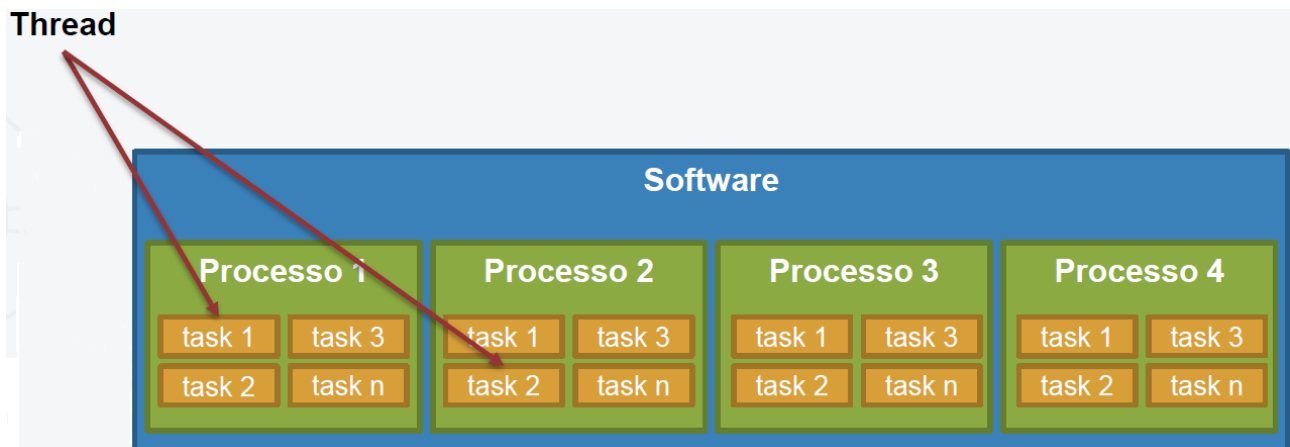
Quindi, la concorrenza è un insieme di concetti che consentono di rappresentare e descrivere l'esecuzione di due o più processi in maniera simultanea o non simultanea, a seconda della tipologia di architettura su cui stiamo eseguendo un software. Due o più processi, quindi due o più istanze di diversi software o dello stesso software, sono in esecuzione concorrente se vengono eseguiti in parallelo. A seconda del tipo di architettura, abbiamo 2 tipi di parallelismo:

- il parallelismo reale avviene quando più processi sono attivi su un dispositivo dotato di più processori, dove ogni processore prende in carico un processo, oppure quando più processi sono attivi su più dispositivi indipendenti tra loro e distribuiti.
- Il parallelismo apparente avviene quando più processi sono attivi su un dispositivo dotato di un processore.

CICLO DI VITA DI UN THREAD

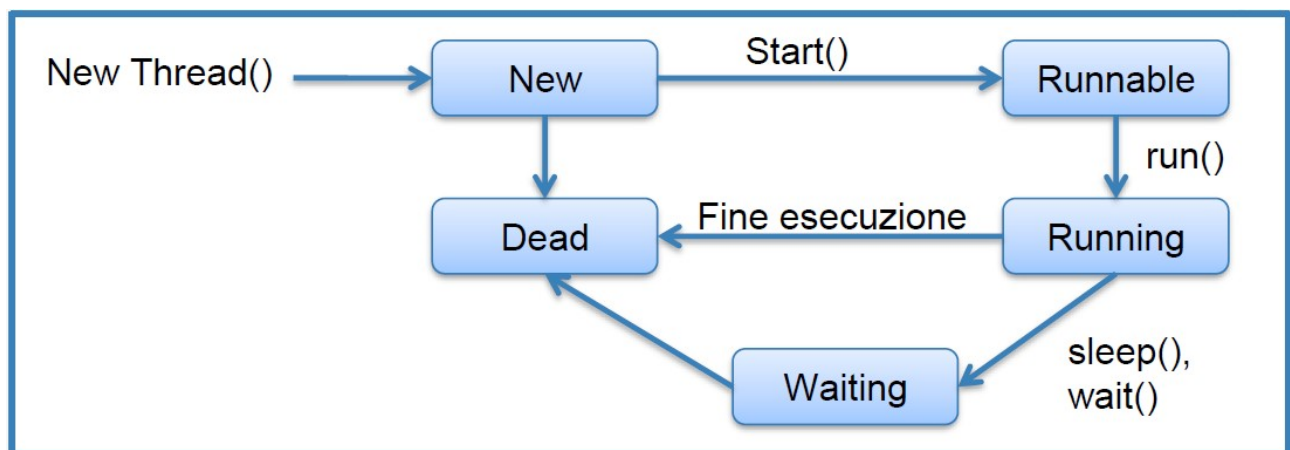
Un thread, dal punto di vista del processo, abbiamo già visto essere un sottoprocesso (o task), quindi un componente di un processo più ampio e complesso che si occupa di gestire una risorsa o un evento.

Thread



Un thread ha un suo ciclo di vita, che comprende i seguenti stati:

- New
- Runnable
- Running
- Waiting
- Dead



Il primo stato in cui si trova un thread è New quando viene istanziato, poi abbiamo Runnable quando viene avviato, Running quando viene eseguito. Arrivato allo stato di Running, un thread può arrivare a Waiting, oppure a Dead, a seconda del caso se vogliamo sospendere o terminare il suo ciclo di vita.

THREAD PRIORITY

La priority (o priorità) è l'informazione che indica allo scheduler il livello di importanza di un thread rispetto agli altri. Lo scheduler è un software che si occupa di gestire l'esecuzione dei thread durante l'esecuzione di un processo.

Facendo attenzione, però, su un concetto, non tutti i sistemi operativi garantiscono che l'ordine di esecuzione dei thread è determinato sulla base della thread priority che abbiamo stabilito. Tuttavia, se un certo numero di thread sono bloccati e in attesa di essere eseguiti, il primo che verrà sbloccato dallo scheduler sarà quello che ha priorità maggiore. In questo modo evitiamo la situazione che si chiama starvation, ovvero l'impossibilità di ottenere risorse da parte di un processo, perché tutti i task sono bloccati.

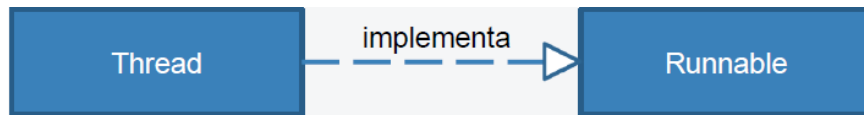
La priority di un thread in JAVA va da 1 a 10, dove la priorità minore è definita `MIN_PRIORITY(1)`, la priorità maggiore è `MAX_PRIORITY(10)` e la priorità normale, che è anche quella di default, è `NORM_PRIORITY(5)`.

CREARE UN THREAD IN JAVA

Per creare un thread in JAVA abbiamo due possibilità:

- creare una classe che estende la classe Thread
- creare una classe che implementa l'interfaccia Runnable

Ovviamente, la classe Thread implementa l'interfaccia Runnable.



Se creiamo un thread attraverso la classe Thread, la nostra classe deve ereditare attraverso la keyword `extends` la classe Thread. Questo permetterà alla nostra classe di ereditare i metodi della classe Thread, in particolare quello più importante di tutti che è il metodo `run()`. Il metodo `run()` deve essere riscritto nella nostra classe, cioè ci verrà richiesto di fare l'override di tale metodo. Il metodo `run()` contiene le istruzioni che devono essere eseguite dal thread. Il codice presente nel metodo `run()` viene eseguito in maniera concorrente ad altri thread presenti in un programma.

Vediamo un primo esempio di thread in JAVA:

```
public class EsempioThread extends Thread {
    /*Tutto quello che è il blocco di codice che dobbiamo eseguire,
    deve essere messo all'interno del metodo run()
    */
    @Override
    public void run() {
        System.out.println("sono un thread");
    }
}

class MainThread{
    public static void main(String[] args) {
        EsempioThread et = new EsempioThread();

        //Per eseguire il thread, dobbiamo invocare il metodo start()
        et.start(); //sono un thread
    }
}
```

Nell'esempio, l'esecuzione del metodo `start()` nel `main()` ha causato l'invocazione del metodo `run()`.

Quindi, per creare un thread dobbiamo:

- creare un'istanza della nostra classe che estende Thread;
- invocare il metodo `start()` che si occuperà, dopo la configurazione del thread, di invocare il metodo `run()`.

Altri metodi utili che si ereditano dalla classe Thread sono:

- `yield()` che suggerisce allo scheduler di liberare la CPU e renderla disponibile, ovviamente, ad altri thread;
- `sleep()` che mette in standby il thread per un determinato numero di millisecondi.

Possiamo eventualmente creare un thread anche utilizzando direttamente l'interfaccia Runnable. In questo caso la nostra classe deve implementare l'interfaccia Runnable. Questo secondo metodo viene utilizzato, piuttosto che utilizzare la classe Thread, quando la nostra classe estende già un'altra classe e, siccome non possiamo usare l'ereditarietà multipla con l'attributo extends, possiamo utilizzare l'interfaccia Runnable che ci consentirà comunque di implementare un thread. Ovviamente, implementando l'interfaccia, dobbiamo implementare il metodo run(), che è definita all'interno dell'interfaccia Runnable. Per creare un thread in questo caso dobbiamo utilizzare il costruttore della classe Thread, che prende in ingresso come parametro l'istanza della nostra classe che implementa Runnable. Vediamo un esempio:

```
public class EsempioRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("sono un thread runnable");
    }
}

class MainRunnable {
    public static void main(String[] args) {
        /*Per eseguire questo thread, dobbiamo creare un'istanza della
        classe Thread, passando in ingresso un'istanza della nostra
        classe che implementa l'interfaccia Runnable*/
        Thread t = new Thread(new EsempioRunnable());

        t.start();//sono un thread runnable
    }
}
```

MULTITHREADING

Multithreading vuol dire eseguire contemporaneamente più thread appartenenti allo stesso processo. Il multithreading può essere:

- collaborativo, ovvero i thread rimangono attivi fino a quando non terminano il task, oppure fino a quando non cedono le risorse occupate ad altri thread;
- preventivo, ovvero la macchina virtuale accede ad un thread attivo e lo controlla attraverso un altro thread.

Le specifiche JAVA stabiliscono che la JVM debba gestire i thread, utilizzando lo scheduling preemptive (o fixed-priority scheduling). Questo vuol dire che lo scheduler ha il compito di interrompere o ripristinare i thread a seconda del loro stato. Quindi, in base in cui si trovano i thread, lo scheduler può attivarli o disattivarli.

Come abbiamo già detto, ogni esecuzione della JVM corrisponde ad un processo, mentre tutto quello che viene eseguito dalla JVM corrisponde ad un thread.

Chiaramente il multithreading ha ragione di esistere perché, se eseguiamo le operazioni in parallelo, ovviamente riusciamo a raggiungere un risultato in maniera più rapida. Oltre a questo, un'altra caratteristica dei software multithreading è che i thread possono scambiarsi informazioni tra loro ed accedere a risorse condivise, ad esempio un database condiviso tra thread, oppure una risorsa hardware.

Vediamo un esempio in codice di multithreading:

```
public class EsempioMultithreading extends Thread{
    @Override
    public void run(){

        System.out.println("Sono il thread " + getName());

        for (int i = 0; i < 10; i++){
            System.out.println(i);
        }
        /*Utilizzo il metodo sleep per dare un tempo di pausa
        tra una stampa di i e un'altra*/
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class MainMultithreading{
    public static void main(String[] args) {

        EsempioMultithreading em1 = new EsempioMultithreading();
        //setName() serve per poter assegnare un nome al thread
        em1.setName("Thread1");

        EsempioMultithreading em2 = new EsempioMultithreading();
        em2.setName("Thread2");

        EsempioMultithreading em3 = new EsempioMultithreading();
        em3.setName("Thread3");

        EsempioMultithreading em4 = new EsempioMultithreading();
        em4.setName("Thread4");

        EsempioMultithreading em5 = new EsempioMultithreading();
        em5.setName("Thread5");

        em1.start();
        em2.start();
        em3.start();
        em4.start();
        em5.start();

    }
}
```

In questo codice abbiamo 5 thread che cicla la variabile i per 10 volte. Tutti e 5 i thread si eseguono in maniera concorrenziale tra di loro, richiedendo l'accesso alle risorse e, il più delle volte, interrompendosi tra di loro. Ragion per cui, eseguendo più volte questo codice, otterremo sempre un output diverso, perché cambierà sempre l'ordine di esecuzione dei thread e i momenti in cui uno sospenderà l'esecuzione dell'altro.

Vediamo un esempio dei tanti output che possiamo ottenere dal codice precedente:

```
Sono il thread Thread3
0
1
Sono il thread Thread1
0
1
2
3
4
5
6
7
8
9
Sono il thread Thread2
0
1
2
3
4
5
6
7
8
9
Sono il thread Thread4
0
1
2
3
4
5
6
7
8
9
Sono il thread Thread5
0
1
2
3
4
5
6
7
8
9
2
3
4
5
6
7
8
9
```

In questo output in particolare abbiamo:

- il thread 3 che è il primo ad essere eseguito, ma interrotto bruscamente dal thread 1 che ha richiesto l'accesso;
- in seguito, verranno eseguiti in quest'ordine il thread 1, il thread 2, il thread 4 e il thread 5, che non verranno mai interrotti durante la loro esecuzione;
- infine, il thread 3 completa la sua esecuzione dopo il thread 5, interrotta precedentemente dal thread 1.

Alla prossima esecuzione del codice, otterremo un output diverso.

CONCORRENZA IN JAVA

La concorrenza è la possibilità di eseguire più task in parallelo. L'utilizzo della concorrenza è molto utile quando i processi possono essere parallelizzati, senza necessità di serializzarli. Ad esempio, immaginiamo che in un browser possiamo caricare contemporaneamente più pagine. Se il browser non fosse stato un software multithreading, non avremmo potuto effettuare questa operazione, ma avremmo dovuto aspettare il caricamento di una pagina per poterne richiedere un'altra.

In JAVA la concorrenza si implementa tramite tre strumenti:

- I Thread, disponibili dalla versione 1.0 di JAVA.
- Il framework Executor, disponibile dalla versione 1.5 di JAVA.
- Il framework Fork/Join, disponibile dalla versione 1.7 di JAVA.

CONCORRENZA CON L'UTILIZZO DEI THREAD

Per creare un thread è necessario, abbiamo già detto, creare la classe che estenda Thread, oppure che implementi Runnable, e che implementi in entrambi i casi il metodo run(). All'interno del metodo run() dobbiamo inserire la logica relativa al task da eseguire.

L'avvio del thread si ha con il metodo start(), mentre l'attesa del completamento del task (l'attesa dell'esecuzione completa del thread) si ha con il metodo join().

Se vogliamo passare dei parametri d'ingresso, siccome il metodo run() non accetta parametri d'ingresso ed è un metodo void (cioè non restituisce parametri), dobbiamo definire un costruttore personalizzato che riceva in ingresso i parametri necessari. A quel punto, possiamo lavorare con i parametri ricevuti in ingresso dal costruttore.

Facciamo un esempio in cui supponiamo di creare una classe che consenta di recuperare l'output (l'html) di una pagina web:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

public class GetPaginaSito extends Thread {

    private String url;
    private String content;

    /*Creo il costruttore per poter passare un parametro in ingresso
    * alla logica all'interno di new()*/
    public GetPaginaSito(String url){
        super();
        this.url = url;
    }

    @Override
    public void run() {
```

```

    /*Dentro il metodo run() implemento la logica che permette
    di invocare una URL e di recuperare le informazioni*/
    try {
        /*Sto passando la url in ingresso dal costruttore*/
        URL u = new URL(url);

        /*Creo la connessione*/
        URLConnection con = u.openConnection();

        /*Attraverso l'invocazione del metodo getInputStream(), recuperiamo
        l'output del nostro sito, quindi la nostra pagina html*/
        InputStream is = con.getInputStream();

        /*Scrivo il contenuto della pagina html in un file, mediante il
        metodo getString() della classe Utils, che prende in ingresso
        un InputStream e scrive in uno StringBuilder tutto l'output
        della nostra pagina web*/
        setContent(Utils.getString(is));

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

/*Genero i metodi getter e setter per l'accesso alla variabili private url e
content*/
public String getUrl() {
    return url;
}

public void setUrl(String url) {
    this.url = url;
}

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}
}

class Utils {
    public static String getString(InputStream is) {
        BufferedReader br = null;
        StringBuilder sb = new StringBuilder();

        String line;
        try {
            br = new BufferedReader(new InputStreamReader(is));
            while ((line = br.readLine()) != null) {
                sb.append(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {

```

```

        try {
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

return sb.toString();
}

}

class MainConcorrenza{
    public static void main(String[] args) {
        GetPaginaSito s1 = new GetPaginaSito("http://www.paolopreite.it");
        GetPaginaSito s2 = new GetPaginaSito("http://www.google.it");

        s1.start();
        s2.start();

        /*Attendo il completamento di ciascun thread, prima di eseguire
        quello successivo, in modo da non farli interrompere a vicenda*/
        try {
            s1.join();
            s2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Output sito Paolo Preite");
        System.out.println(s1.getContent());

        System.out.println("Output sito Google");
        System.out.println(s2.getContent());
    }
}

```

CONCORRENZA CON L'UTILIZZO DEGLI EXECUTOR

Vediamo adesso come gestire la concorrenza attraverso gli Executor, che sono disponibili a partire dalla versione 1.5 di JAVA.

Dobbiamo creare innanzitutto una classe che implementi l'interfaccia Callable e che definisca il metodo call(). A questo punto, per eseguire i thread, dobbiamo creare un'istanza della classe ExecutorService ed utilizzare il metodo invokeAll(). Questo metodo ritorna una lista di oggetti Future, attraverso cui è possibile recuperare il valore di ritorno di ogni invocazione.

Vediamo di nuovo l'esempio della pagina web, ma con l'utilizzo degli Executor:

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class GetPaginaSitoExecutor implements Callable <String>{ /*String è il
tipo di output*/
    private String url;
    private String content;

    public GetPaginaSitoExecutor(String url){
        super();
        this.url = url;
    }

    @Override
    public String call() throws Exception {

        try {
            URL u = new URL(url);

            URLConnection con = u.openConnection();

            InputStream is = con.getInputStream();

            return Utils.getString(is);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getContent() {
```

```

        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}

class MainConcorrenzaExecutor{
    public static void main(String[] args) throws InterruptedException,
    ExecutionException {
        List<Callable<String>> siti = new ArrayList<Callable<String>>();

        siti.add(new GetPaginaSitoExecutor("http://www.paolopreite.it"));
        siti.add(new GetPaginaSitoExecutor("http://www.google.it"));

        ExecutorService ex = Executors.newSingleThreadExecutor();

        List<Future<String>> out = ex.invokeAll(siti);

        for(Future<String> future : out){ /*for each*/
            System.out.println(future.get());
        }

        ex.shutdown();/*Termino l'istanza dell'ExecutorService*/
    }
}

```

CONCORRENZA CON L'UTILIZZO DEL FRAMEWORK FORK/JOIN

Vediamo quest'ultimo in cui gestiamo la concorrenza tra thread attraverso il framework Fork/Join, disponibile dalla versione 1.7 di JAVA. Questo framework è una specializzazione del framework Executor.

In questo caso dobbiamo creare una classe che estenda la classe RecursiveTask ed implementi il metodo compute().

Per eseguire i thread dobbiamo creare un'istanza della classe ForkJoinPool ed utilizzare il metodo invoke().

Vediamo di nuovo l'esempio della pagina web, ma con l'utilizzo del framework Fork/Join:

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class GetPaginaSitoForkJoin extends RecursiveTask<String> {
    private String url;
    private String content;

    public GetPaginaSitoForkJoin(String url) {
        super();
        this.url = url;
    }

    @Override
    protected String compute() {
        try {
            URL u = new URL(url);

            URLConnection con = u.openConnection();

            InputStream is = con.getInputStream();

            return Utils.getString(is);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
```



```

        this.content = content;
    }
}

class MainConcorrenzaForkJoin{
    public static void main(String[] args) {
        ForkJoinPool f = new ForkJoinPool();

        System.out.println(f.invoke(new
GetPaginaSitoForkJoin("http://www.paolopreite.it")));
        System.out.println(f.invoke(new
GetPaginaSitoForkJoin("http://www.google.it")));

        f.shutdown();
    }
}

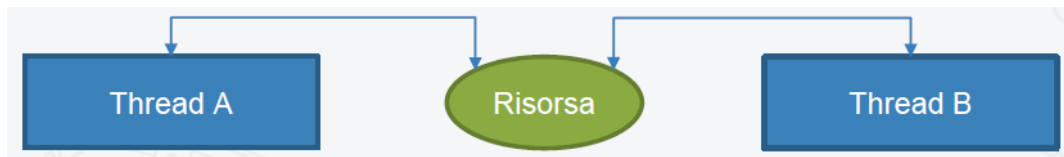
```

SINCRONIZZAZIONE

Per capire cosa è la sincronizzazione, vediamo un esempio. Supponiamo che Mario e Lucia abbiano un conto corrente bancario cointestato, quindi entrambi gli utenti possono accedere allo stesso conto corrente, hanno due bancomat e possono entrambi prelevare da due terminali diversi. Dobbiamo immaginare lo scenario in cui Mario e Lucia effettuano nello stesso istante la visualizzazione dell'estratto conto, vedono che il saldo è sufficiente ed effettuano il prelievo da due terminali diversi. Se non gestiamo il fatto che i due utenti accedono allo stesso conto corrente, Mario e Lucia possono effettuare il prelievo nello stesso istante, correndo il rischio che il saldo diventi negativo.

Mario e Lucia sono due thread che accedono alla stessa risorsa, ossia il conto corrente. Per poter prevenire eventuali situazioni anomale (ad esempio un saldo negativo sul conto corrente), è necessario gestire la sincronizzazione tra due thread che accedono alla stessa risorsa.

JAVA consente di gestire la sincronizzazione tra thread attraverso la keyword `synchronized`. Questa keyword consente di bloccare una risorsa, garantendo l'accesso esclusivo ad un thread. Quando questa risorsa è bloccata, nessun thread può accedervi finché la risorsa che l'ha bloccata non la libera.



Nell'esempio precedente, se Mario blocca la risorsa conto corrente durante il prelievo, Lucia non potrà prelevare finché Mario non avrà effettuato il prelievo. A quel punto verrà effettuato il controllo sul saldo e, quindi, se il saldo è disponibile, verrà effettuato il prelievo, altrimenti no.

Per quanto riguarda la keyword `synchronized`:

- in una classe possiamo definire più metodi `synchronized`, a seconda delle necessità;
- un metodo `synchronized` può essere eseguito solo da un thread alla volta;
- quando esistono più metodi `synchronized` in una classe, solo un metodo per volta può essere invocato;
- quando viene invocato un metodo `synchronized`, ovviamente il thread chiamante tecnicamente si dice che ottiene il lock, quindi blocca l'accesso da parte di altri thread a quella risorsa;
- i thread che vogliono accedere ad una risorsa bloccata rimangono in attesa di ricevere il lock, quindi rimangono in stato sospeso finché la risorsa non viene sbloccata dal thread che l'aveva occupata;
- quando un thread termina l'esecuzione di un metodo `synchronized`, il lock viene rilasciato e il metodo torna disponibile anche agli altri thread.

Vediamo l'esempio di Mario e Lucia con il codice:

Cliente.java

```
public class Cliente extends Thread {  
    private double sommaDaPrelevare;  
  
    public Cliente(String nomeCliente, double sommaDaPrelevare) {  
        super();  
    }  
}
```

```

        this.setName(nomeCliente);
        this.sommaDaPrelevare = sommaDaPrelevare;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "
arriva al bancomat");
        System.out.println("Quando arriva " +
Thread.currentThread().getName() + " il saldo è: " +
ContoCorrente.getInstance().getSaldo());
        System.out.println("La somma che vuole prelevare " +
Thread.currentThread().getName() + " è: " + sommaDaPrelevare);

        try {

            ContoCorrente.getInstance().prelievo(sommaDaPrelevare);
            System.out.println(Thread.currentThread().getName()
+ " TUTTO OK PRELIEVO EFFETTUATO");
        } catch (Exception e) {
            System.out.println(Thread.currentThread().getName()
+ " NOOOOOOOOOOOO NON HAI SOLDI!!!");
            e.printStackTrace();
        }
    }
}

```

ClienteNonSync.java

```

public class ClienteNonSync extends Thread {
    private double sommaDaPrelevare;

    public ClienteNonSync(String nomeCliente, double sommaDaPrelevare) {
        super();
        this.setName(nomeCliente);
        this.sommaDaPrelevare = sommaDaPrelevare;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "
arriva al bancomat");
        System.out.println("Quando arriva " +
Thread.currentThread().getName() + " il saldo è: " +
ContoCorrente.getInstance().getSaldo());
        System.out.println("La somma che vuole prelevare " +
Thread.currentThread().getName() + " è: " + sommaDaPrelevare);

        try {

            ContoCorrente.getInstance().prelievoNonSync(sommaDaPrelevare);
            System.out.println(Thread.currentThread().getName()
+ " TUTTO OK PRELIEVO EFFETTUATO");
        } catch (Exception e) {
            System.out.println(Thread.currentThread().getName()
+ " NOOOOOOOOOOOO NON HAI SOLDI!!!");
            e.printStackTrace();
        }
    }
}

```

ContoCorrente.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ContoCorrente {
    private static ContoCorrente cc;

    public static ContoCorrente getInstance() {
        if(cc == null)
            cc = new ContoCorrente();

        return cc;
    }

    public double getSaldo() {
        double saldo = 0;

        BufferedReader br = null;
        try {
            File fin = new File(new
File(".").getCanonicalPath() + File.separator + "db.txt");

            br = new BufferedReader(new FileReader(fin));

            String line = null;
            while ((line = br.readLine()) != null) {
                saldo = Double.parseDouble(line);
                break;
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(br != null)
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }

        return saldo;
    }

    public synchronized void prelievo(double somma) throws Exception {
        Thread.sleep(5000);

        BufferedWriter bw = null;
        FileWriter fw = null;

        try {
            double nuovoSaldo = getSaldo() - somma;

            if(nuovoSaldo > 0) {
                fw = new FileWriter(new
File(".").getCanonicalPath() + File.separator + "db.txt");
```

```

        bw = new BufferedWriter(fw);
        bw.write(nuovoSaldo+"");
    } else
        throw new Exception("Saldo
insufficiente!");
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (bw != null)
                bw.close();

            if (fw != null)
                fw.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

public void prelievoNonSync(double somma) throws Exception {
    Thread.sleep(5000);

    BufferedWriter bw = null;
    FileWriter fw = null;

    try {
        double nuovoSaldo = getSaldo() - somma;

        if(nuovoSaldo > 0) {
            fw = new FileWriter(new
File(".").getCanonicalPath() + File.separator + "db.txt");
            bw = new BufferedWriter(fw);
            bw.write(nuovoSaldo+"");
        } else
            throw new Exception("Saldo
insufficiente!");
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (bw != null)
                bw.close();

            if (fw != null)
                fw.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
}

```

Main.java

```
public class Main {

    public static void main(String[] args) throws InterruptedException {
        Cliente c1 = new Cliente("Mario", 20);
        Cliente c2 = new Cliente("Lucia", 50);

        // Avvio i Threads
        c1.start();
        c2.start();

        // Attendo il completamento
        c1.join();
        c2.join();

        /*****/

        // ClienteNonSync c3 = new ClienteNonSync("Mario", 20);
        // ClienteNonSync c4 = new ClienteNonSync("Lucia", 50);
        //
        // // Avvio i Threads
        // c3.start();
        // c4.start();
        //
        // // Attendo il completamento
        // c3.join();
        // c4.join();
    }

}
```

La classe Cliente e la classe ClienteNonSync sono entrambi thread, perché estendono entrambe la classe Thread. All'interno di entrambe abbiamo la variabile sommaDaPrelevare e il costruttore che prende in ingresso il nome del cliente e la somma da prelevare. Importante da ricordare è che, quando creiamo un thread che estende la classe Thread, per passare delle variabili come parametri in ingresso, è necessario dichiarare tali variabili all'interno della classe ed implementare il costruttore. All'interno del metodo run() di entrambi i thread, abbiamo delle istruzioni che ci dicono chi arriva al bancomat, quanto è il saldo nel momento in cui l'utente arriva al bancomat e quanto è la somma da prelevare. Dopodiché, sempre nel metodo run(), abbiamo il blocco try catch, dove nel try vengono prelevati rispettivamente i metodi prelievoNonSync per la classe ClienteNonSync e prelievo per la classe Cliente. A questi due metodi viene passato in ingresso ovviamente la somma da prelevare. Se il prelievo viene effettuato, abbiamo il messaggio di "prelievo effettuato", altrimenti abbiamo un messaggio di "saldo negativo".

La classe ContoCorrente utilizza il singleton pattern e ha tre metodi, ossia getSaldo() che ritorna il saldo, prelievo() e prelievoNonSync(). Il codice tra gli ultimi due metodi è pressoché identico, cambia solo l'utilizzo della keyword synchronized per il metodo prelievo(). In entrambi i casi, se il prelievo viene effettuato con successo, viene riscritto il file db.txt con il nuovo saldo, altrimenti abbiamo un'eccezione.

Nella classe Main ci sono rispettivamente due istanze dei thread Cliente e ClienteNonSync, vengono avviati con il metodo start() e si attende il loro completamento con il metodo join().

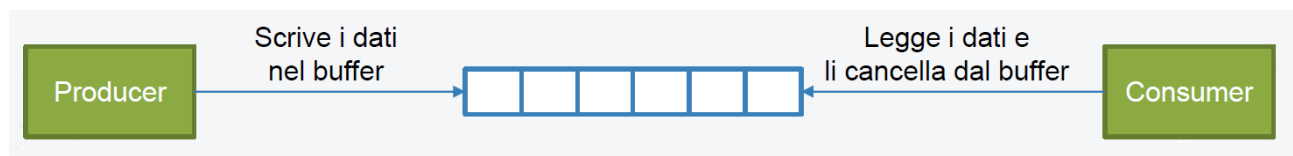
Con questo codice possiamo studiare il comportamento di due thread, sia quando abbiamo la sincronizzazione, sia quando non ce l'abbiamo. Quando abbiamo la sincronizzazione, solo uno dei due

thread entrerà nel metodo `prelievo()`, mentre l'altro resta in attesa. Quando non abbiamo la sincronizzazione, tutti e due i thread invece avranno libero accesso contemporaneamente al metodo `prelievoNonSync()`

ESEMPIO MULTITHREADING E CONCORRENZA : PRODUCER-CONSUMER

Il problema del producer-consumer (noto anche come problema del buffer limitato) è un classico esempio di sincronizzazione tra processi.

Ci sono i processi, producer e consumer, che condividono un buffer di dimensione fissa. Il buffer è una zona di memoria usata per compensare differenze di velocità nel trasferimento o nella trasmissione di dati, oppure per velocizzare l'esecuzione di alcune operazioni. Il producer genera dei dati e li scrive nel buffer, mentre il consumer legge i dati scritti dal producer e li cancella dal buffer. Quindi, da un lato abbiamo un utente che scrive, mentre dall'altro abbiamo un utente che legge e cancella.



Il problema consiste nell'assicurarsi che:

- il producer non elabori nuovi dati quando il buffer è pieno, ma che si metta in pausa;
- il consumer non cerchi di leggere dati quando il buffer è vuoto, ma che si metta in pausa.

La soluzione consiste nel:

- sospendere l'esecuzione del producer se il buffer è pieno. Quando il consumer preleva un elemento dal buffer, esso provvederà a svegliare il producer, che riprenderà a riempire il buffer;
- sospendere l'esecuzione del consumer se il buffer è vuoto. Quando il producer avrà inserito i dati nel buffer, esso provvederà a svegliare il consumer, che riprenderà a leggere e svuotare il buffer.

Questa soluzione può essere implementata mediante l'utilizzo dei semafori, che sono strategie di comunicazione tra processi. In questo caso bisogna fare molta attenzione perché, se non implementata correttamente, la soluzione porta ad avere una situazione di deadlock, in cui tutti e due i processi restano in attesa di essere risvegliati e non vengono risvegliati mai. Vediamo un esempio:

Consumer.java

```
import java.util.List;

public class Consumer implements Runnable {
    private final List<Integer> bufferCondiviso;

    public Consumer(List<Integer> bufferCondiviso, int size) {
        this.bufferCondiviso = bufferCondiviso;
    }

    @Override
    public void run() {
```

```

        while (true) {
            try {
                System.out.println("Il thread Consumer sta leggendo il buffer...");
                consume();
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }

        private void consume() throws InterruptedException {
            // il thread resta in stato wait se il buffer è vuoto

            while (bufferCondiviso.isEmpty()) {
                synchronized (bufferCondiviso) {
                    System.out.println("Il buffer è vuoto, il thread Consumer resta in attesa... la dimensione del buffer adesso è: " + bufferCondiviso.size());

                    bufferCondiviso.wait();
                }
            }

            // il buffer contiene elementi, quindi il thread può eliminarne uno e
            // notificarlo al producer
            synchronized (bufferCondiviso) {
                System.out.println("Il thread Consumer sta leggendo il buffer ed eliminando il seguente elemento: " + bufferCondiviso.remove(0) + " la dimensione del buffer adesso è: " + bufferCondiviso.size());

                bufferCondiviso.notifyAll();
            }
        }
    }
}

```


Producer.java

```
import java.util.List;

public class Producer implements Runnable {
    private final List<Integer> bufferCondiviso;
    private final int SIZE;
    private int i = 1;

    public Producer(List<Integer> bufferCondiviso, int size) {
        this.bufferCondiviso = bufferCondiviso;
        this.SIZE = size;
    }

    @Override
    public void run() {
        while(true) {
            try {
                produce();
                i++;
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

    private void produce() throws InterruptedException {
        // il thread resta in stato wait se il buffer è pieno
        while (bufferCondiviso.size() == SIZE) {
            synchronized (bufferCondiviso) {
                System.out.println("Il buffer è pieno, il
thread Producer resta in attesa... la dimensione del buffer adesso è: " +
bufferCondiviso.size());

                bufferCondiviso.wait();
            }
        }

        // il buffer non è pieno, quindi il thread può aggiungere un
nuovo elemento e notificarlo al consumer
        synchronized (bufferCondiviso) {
            bufferCondiviso.add(i);
            bufferCondiviso.notifyAll();

            System.out.println("Il thread Producer ha aggiunto
al buffer l'elemento: " + i + " la dimensione del buffer adesso è: " +
bufferCondiviso.size());
        }
    }
}
```

ProducerConsumerTest.java

```
import java.util.LinkedList;
import java.util.List;

public class ProducerConsumerTest {
    public static void main(String args[]) {
        List<Integer> bufferCondiviso = new LinkedList<Integer>();
        int size = 4;

        Thread prodThread = new Thread(new Producer(bufferCondiviso, size),
"Producer");
        Thread consThread = new Thread(new Consumer(bufferCondiviso, size),
"Consumer");

        prodThread.start();
        consThread.start();
    }
}
```

Abbiamo le classi Producer e Consumer che sono due thread e, infatti, implementano Runnable. Abbiamo in entrambe le classi la lista bufferCondiviso, che è condivisa tra le classi stesse.

Nel caso del Consumer, nel metodo run() viene invocato il metodo consume(). Nel metodo consume(), se il bufferCondiviso è vuoto, viene invocato il metodo synchronized con parametro il bufferCondiviso e il thread Consumer rimane in attesa (con l'utilizzo del metodo wait()) che il bufferCondiviso sia di nuovo disponibile con degli elementi. Se il bufferCondiviso, invece, contiene gli elementi, il thread Consumer può rimuovere il primo elemento della lista (bufferCondiviso.remove(0)) e notifica (bufferCondiviso.notifyAll()) al Producer, che è in attesa sulla variabile bufferCondiviso, che potrà continuare a inserire dati.

Nel caso del Producer, nel metodo run() viene invocato il metodo produce(). Nel metodo produce(), se il bufferCondiviso è pieno, viene invocato il metodo synchronized con parametro il bufferCondiviso e il thread Producer rimane in attesa (con l'utilizzo del metodo wait()) che il bufferCondiviso sia di nuovo disponibile. Se il bufferCondiviso, invece, è vuoto, il thread Producer può aggiungere elementi dalla lista (bufferCondiviso.add(i)) e notifica (bufferCondiviso.notifyAll()) al Consumer, che è in attesa sulla variabile bufferCondiviso, che il buffer contiene degli elementi.

Quindi, attraverso l'implementazione dell'esempio Producer-Consumer, due thread che accedono allo stesso buffer (alla stessa risorsa), rispettivamente in lettura dal lato Consumer e in scrittura dal lato Producer, abbiamo visto come implementare il multithreading e la concorrenza in JAVA. Come abbiamo notato, attraverso la parola chiave synchronized, abbiamo bloccato l'accesso ad una variabile (in questo caso bufferCondiviso). Quando il Producer effettua il lock sulla variabile, il Consumer resta in attesa che venga liberato il lock, quindi che la variabile venga resa disponibile e viceversa.

METODI WAIT(), NOTIFY(), NOTIFYALL()

I metodi wait(), notify() e notifyAll() sono definiti all'interno della classe Object. La classe Object è la superclasse da cui derivano tutte le altre classi di JAVA.

WAIT()

Il metodo wait():

- mette in attesa un thread;
- possiamo invocarlo solo su oggetti per il quale si ha il lock;
- possiamo invocarlo solo in un metodo o in un blocco di codice synchronized, altrimenti avremo l'eccezione IllegalMonitorStateException.

Quando viene invocato il metodo wait() su un oggetto, si hanno i seguenti effetti:

- sull'oggetto viene rilasciato il lock;
- il thread viene posto in stato blocked.

Analogamente al metodo wait(), anche sleep() mette in attesa il thread invocante. Tra i due, però, c'è una differenza:

- quando viene invocato il metodo sleep(), il lock sull'oggetto non viene rilasciato, quindi nessun thread può utilizzarlo;
- quando viene invocato il metodo wait(), invece, viene rilasciato il lock sull'oggetto, il quale diventa accessibile agli altri thread.

Esistono diverse implementazioni del metodo wait():

- wait() causa l'interruzione di un thread finché un altro thread non invoca il metodo notify() o notifyAll();
- wait(long timeout) causa l'interruzione di un thread finché un altro thread non invoca il metodo notify(), o notifyAll(), o se è stato raggiunto il timeout, espresso in millisecondi, impostato. Se il timeout passato in ingresso è 0, il comportamento è lo stesso del metodo wait();
- wait(long timeout, int nanos) è analogo a wait (long timeout), solo che al timeout in millisecondi è possibile aggiungere anche i nanosecondi.

NOTIFY() E NOTIFYALL()

Il metodo `notify()` risveglia un solo thread in attesa su un oggetto che si trovava in stato di lock.

Il metodo `notifyAll()` risveglia tutti i thread in attesa su un oggetto che si trovava in stato di lock.

Quando sono invocati questi due metodi, i thread che ricevono la notifica passano nello stato `Runnable`, quindi possono da quel momento in poi prendere il lock sulla risorsa. Per poter accedere o effettuare modifiche su una risorsa `synchronized`, i thread risvegliati devono acquisire il lock.

Un thread può invocare i due metodi `notify()` e `notifyAll()` solo se ha il lock sulla risorsa.

Quando viene invocato il metodo `notifyAll()` abbiamo che:

- tutti i thread in attesa vengono risvegliati;
- quando vengono risvegliati, tutti quei thread si mettono in coda e il primo che riesce ad acquisire la risorsa, effettua il lock;
- solo un thread per volta può prendere il lock, mentre gli altri dovranno di nuovo attendere che la risorsa venga rilasciata.

SINCRONIZZAZIONE AVANZATA CON LOCK E REENTRANTLOCK

Come abbiamo già visto, possiamo sincronizzare un blocco di codice, un metodo o una variabile attraverso la keyword `synchronized`. L'utilizzo di questa keyword consente di accedere in maniera esclusiva alla risorsa, impedendo che altri thread possano accedere alla stessa risorsa mentre è utilizzata.

L'utilizzo della keyword `synchronized`, tuttavia, ha delle limitazioni, ovvero:

- se abbiamo in una classe più metodi `synchronized`, possiamo effettuare il lock di una risorsa per volta;
- quando un thread effettua il lock su un metodo, non può effettuare l'`unlock` finché non ha eseguito tutto il blocco di codice;
- non possiamo effettuare l'esecuzione di un blocco di codice sincronizzato.

Attraverso l'interfaccia `Lock`, queste limitazioni vengono superate.

LOCK

L'Interfaccia `Lock`, appunto, è stata pensata per offrire gli elementi messi a disposizione dalla keyword `synchronized`, più altri elementi che consentono di superare le limitazioni di `synchronized`.

I metodi più importanti di questa interfaccia sono:

- `lock()`, che effettua il lock di una risorsa;
- `unlock()`, che libera una risorsa;
- `tryLock()`, che attende un certo periodo di tempo, prima di effettuare il lock.

REENTRANTLOCK

La classe `ReentrantLock` è un'implementazione dell'interfaccia `Lock` ed è disponibile dalla versione 1.5 di JAVA. Questa classe, oltre ad implementare i metodi dell'interfaccia `Lock`, contiene altri metodi utili.

Quando utilizziamo la classe `ReentrantLock`, il lock è rientrante. Questo vuol dire che un thread che ha un lock, può acquisire nuovamente il lock più volte.

Vediamo un esempio con il metodo `lock()`:

```
public class ReentrantLockEsempio {  
    private ReentrantLock lock = new ReentrantLock();  
    private int contatore = 0;  
  
    public int conta() {  
        lock.lock();  
  
        try {  
            System.out.println(Thread.currentThread().getName() + " contatore = " + contatore);  
            contatore++;  
  
            return contatore;  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Abbiamo una classe `ReentrantLockEsempio`, che ha una variabile `lock` e un metodo `conta()`. Tutto il codice che si trova dopo l'invocazione del metodo `lock`, deve essere inserita all'interno di un blocco `try`. All'interno del blocco `finally`, invece, bisogna inserire l'`unlock()`. Quando un thread acquisisce il lock sull'istanza della classe `ReentrantLock` (variabile `lock`), nessun altro thread può andare oltre la riga di codice in cui abbiamo la chiamata al metodo `lock()`. Quindi se un altro thread richiede il lock, rimane bloccato sulla riga di codice in cui viene richiamato il metodo `lock()`, finché il lock non viene sbloccato.

Vediamo un esempio con il metodo `tryLock()`:

```
public class ReentrantLockEsempio {  
    private ReentrantLock lock = new ReentrantLock();  
    private int contatore = 0;  
  
    public void somma() {  
        System.out.println("Il thread " + Thread.currentThread().getName() + " ha richiesto ...");  
  
        if(lock.tryLock()) {  
            try {  
                somma += contatore;  
                System.out.println(Thread.currentThread().getName() + " la somma vale = " + somma);  
            } finally {  
                lock.unlock();  
            }  
        } else {  
            .....  
        }  
    }  
}
```

In questo esempio abbiamo la stessa classe di prima, in cui però stavolta abbiamo il metodo `tryLock()`. Attraverso questo metodo, un thread può tentare di acquisire il lock sull'istanza della classe `ReentrantLock` (variabile `lock`), senza rimanere in sospeso come nell'esempio precedente a questo.

Esiste anche un altro metodo `tryLock()`, che prende in ingresso il parametro `timeout` che specifica il tempo di attesa prima di effettuare un altro tentativo di lock, e `lockunit` che specifica l'unità di tempo relativa al `timeout`(secondi, millisecondi...).

UTILIZZARE IL BLOCCO TRYFINALLY CON I THREAD

Il problema consiste nell'avere un thread, al suo interno abbiamo un blocco `try/catch/finally` e il thread viene interrotto durante la sua esecuzione. Si deve capire cosa succede quando avviene questa interruzione.

Partiamo con il distinguere due tipi di interruzione:

- se un thread sta eseguendo le istruzioni all'interno del `try` o del `catch` e viene terminato (cioè killed), il blocco `finally` potrebbe non essere eseguito. Questo è il caso in cui il processo, associato alla JVM, muore durante l'esecuzione del thread;
- se un thread sta eseguendo le istruzioni all'interno del `try` o del `catch` e viene interrotto (tramite il metodo `interrupt()`), il blocco `finally` viene eseguito.

Vediamo un esempio:

EsempioTryCatchFinally.java

```
public class EsempioTryCatchFinally extends Thread {
    private long sleep;

    public EsempioTryCatchFinally(long sleep) {
        super();
        this.sleep = sleep;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(sleep);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Entrato nel finally!");
        }
    }
}
```

Main2.java

```
public class Main2 {  
  
    public static void main(String[] args) throws InterruptedException {  
        EsempioTryCatchFinally t1 = new EsempioTryCatchFinally(10);  
  
        // Avvio il Thread  
        t1.start();  
  
        // Interrompo il thread  
        t1.interrupt();  
    }  
}
```

Qui abbiamo una classe `EsempioTryCatchFinally`, che estende la classe `Thread`, in cui abbiamo un blocco `try/catch/finally`. Nella classe `Main2` creo un'istanza della classe `EsempioTryCatchFinally`, avvio il thread e lo interrompo con il metodo `interrupt()`. L'output che ci esce fuori ci avvisa che è riuscito ad entrare nel blocco `finally`.

THREAD POOL

I thread pool sono dei componenti software che si occupano di gestire i thread, con l'obiettivo di ottimizzare e semplificarne l'utilizzo.

Quando scriviamo un software multi-threading, abbiamo visto che ogni thread ha il compito di eseguire un determinato task. Attraverso il thread pool possiamo gestire l'esecuzione di una lista di thread.

Ovviamente il thread pool è dotato di una sua coda interna, che consente di aggiungere più thread, accodandoli tra loro. La gestione dell'esecuzione dei vari thread viene lasciata in carico al thread pool.

Per decidere quale thread deve essere eseguito per primo e quanti eseguirne, esistono diversi algoritmi. In JAVA sono state implementate diverse classi che implementano questi algoritmi e consentono di definire diversi thread pool, a seconda della necessità.

I motivi per cui utilizzare il thread pool sono i seguenti:

- abbiamo un aumento delle prestazioni delle applicazioni che li utilizzano, poiché il thread pool ottimizza l'utilizzo della RAM e della CPU;
- I task vengono eseguiti in maniera più veloce, perché le operazioni vengono parallelizzate, avendo un thread pool che consente di eseguire più thread contemporaneamente;
- dal punto di vista del codice sorgente, abbiamo una maggiore eleganza di scrittura. Quindi il codice risulta essere più pulito, perché non dobbiamo occuparci della creazione e della gestione dei thread, ovvero non dobbiamo fare più una cosa del genere:

```
public static void main(String[] args) {  
  
    EsempioMultithreading em1 = new EsempioMultithreading();  
    //setName() serve per poter assegnare un nome al thread  
    em1.setName("Thread1");  
  
    EsempioMultithreading em2 = new EsempioMultithreading();  
    em2.setName("Thread2");  
  
    EsempioMultithreading em3 = new EsempioMultithreading();  
    em3.setName("Thread3");  
  
    EsempioMultithreading em4 = new EsempioMultithreading();  
    em4.setName("Thread4");  
  
    EsempioMultithreading em5 = new EsempioMultithreading();  
    em5.setName("Thread5");  
  
    em1.start();  
    em2.start();  
    em3.start();  
    em4.start();  
    em5.start();  
  
}
```

Per implementare i thread pool in JAVA abbiamo l'interfaccia Executor, l'interfaccia ExecutorService e la classe Executors.

L'interfaccia Executor è l'interfaccia base che definisce quali sono i meccanismi principali per la gestione di

un thread pool. Ovviamente, essendo un' interfaccia, definisce dei metodi, ma non li implementa. Il metodo principale `execute(thread da eseguire)` consente di aggiungere nuovi thread al pool.

L'interfaccia `ExecutorService` estende l'interfaccia `Executor` ed aggiunge una serie di metodi per ottimizzare la gestione del pool, tra cui `shutdown()` che indica al pool di avviare la chiusura di tutti i thread. Dopo il metodo `shutdown()`, non posso aggiungere più nuovi thread al pool.

La classe `Executors` è una classe factory che consente di creare varie istanze di pool (`Executor`, `ExecutorService`,...).

Facciamo un esempio, riprendendo il caso in cui dobbiamo ottenere l'output in html di una pagina web:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class GetPaginaSitoPool extends Thread {
    private String url;
    private String content;

    public GetPaginaSitoPool(String url) {
        super();
        this.url = url;
    }

    @Override
    public void run() {
        try {
            URL site = new URL(url);
            URLConnection con = site.openConnection();

            InputStream in = con.getInputStream();
            String encoding = con.getContentEncoding();
            encoding = encoding == null ? "UTF-8" : encoding;

            System.out.println("*****");
            System.out.println("CONTENUTO DELLA PAGINA WEB: " + url);
            System.out.println(getString(in));

            System.out.println("*****");
        } catch (IOException e) {
            e.printStackTrace();
        }

        private String getString(InputStream is) {
            BufferedReader br = null;
            StringBuilder sb = new StringBuilder();

            String line;
            try {
                br = new BufferedReader(new InputStreamReader(is));
                while ((line = br.readLine()) != null) {
                    sb.append(line);
                }
            }
        }
    }
}
```

```

        }

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    return sb.toString();
}

public String getContent() {
    return content;
}
}

class MainPool{
    public static void main(String[] args) {
        /*creo il thread pool*/
        ExecutorService pool = Executors.newCachedThreadPool();

        /*aggiunge i thread al pool*/
        pool.execute(new GetPaginaSitoPool("http://www.google.com"));
        pool.execute(new GetPaginaSitoPool("https://www.ibconline.it"));

        /*spengo il thread pool*/
        pool.shutdown();
    }
}

```

Nella classe MainPool creiamo un'istanza di ExecutorService, che crea un thread pool. All'interno del pool aggiungiamo, attraverso il metodo execute(), i thread che ci interessano. In alternativa al CachedThreadPool(), che è un pool in cui potenzialmente possiamo inserire infiniti thread, abbiamo il newFixedThreadPool(int n Threads), in cui il numero di thread che si possono inserire in questo pool è dato da quello specificato dal parametro in ingresso n Threads. Sempre nel caso del newFixedThreadPool(int n Threads), se ci sono più thread di quelli ammissibili, i restanti verranno accodati in attesa di essere eseguiti.

É caldamente consigliato di utilizzare sempre i thread pool quando lavoriamo in ambiente multithreading, perché consentono di effettuare dei lavori sui thread in maniera molto più semplice rispetto ai thread classici delle prime versioni di JAVA.

CLASSI ARRAYBLOCKINGQUEUE E LINKEDBLOCKINGQUEUE

Prima di parlare di queste due classi, vediamo quali sono le interfacce messe a disposizione da JAVA per rappresentare una generica coda. Le due classi `ArrayBlockingQueue` e `LinkedBlockingQueue` le utilizziamo, appunto, per la gestione delle code di thread.

In JAVA, una generica coda è rappresentata dall'interfaccia `Queue`, che estende l'interfaccia `Collection`. Estendendo l'interfaccia `Collection`, ne eredita tutte le caratteristiche (ad esempio i metodi `add(e)`, `remove(e)`, `size()` e così via).

In aggiunta ai metodi definiti dall'interfaccia `Collection`, l'interfaccia `Queue` ne definisce altri, tra cui:

- `peek()`, che recupera il primo elemento della coda, senza eliminarlo. Questo metodo ritorna l'elemento recuperato, oppure `null` se la coda è vuota;
- `element()`, che recupera, come il metodo `peek()`, il primo elemento della coda, con la differenza che se la coda è vuota, genera l'eccezione `NoSuchElementException` al posto di `null`;
- `poll()`, che recupera e rimuove il primo elemento della coda. Questo metodo ritorna l'elemento rimosso, oppure `null` se la coda è vuota.

L'interfaccia `Queue` viene estesa dall'interfaccia `BlockingQueue`, che rappresenta anch'essa una generica coda bloccante, solo che quest'ultima definisce dei metodi che devono garantire l'esecuzione sicura delle operazioni. In particolare, tali metodi sono:

- `put()`, che inserisce un oggetto alla fine della coda. Se la coda è piena, il metodo si blocca e mette il thread corrente in attesa, riattivandolo quando viene rimosso un elemento dalla coda;
- `take()`, che restituisce il primo elemento della coda. Se la coda è vuota, il metodo si blocca e mette il thread corrente in attesa, riattivandolo quando viene inserito un nuovo elemento alla coda.

Chiaramente, definendo la possibilità di mettere in attesa i thread, i software che utilizzano questa interfaccia sono sincronizzati.

Badiamo bene che si parla di interfaccia e non di classe, quindi l'interfaccia `BlockingQueue` non implementa questa logica, ma la definisce. L'interfaccia `BlockingQueue`, infatti, è implementata da diverse classi. Le classi che la implementano sono Thread Safe e sono:

- `ArrayBlockingQueue`
- `LinkedBlockingQueue`

CLASSE ARRAYBLOCKINGQUEUE

La classe `ArrayBlockingQueue` è un array circolare di tipo bloccante. Un oggetto di tipo `ArrayBlockingQueue` ha la particolarità di avere una capacità fissa, definita in fase di inizializzazione. Quindi nel costruttore dobbiamo passare la capacità e quella rimarrà.

Gli elementi sono ordinati all'interno della coda secondo le specifiche FIFO (First-In First-Out), cioè il primo elemento in ingresso è il primo ad uscire.

Riprendiamo l'esempio del Produttore/Consumatore, sostituendo la variabile `synchronized` con un `BlockingQueue`:

Producer.java

```
import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {
    private BlockingQueue<String> queue;

    public Producer(BlockingQueue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        int i = 0;

        while(true) {
            String elem = "Elemento numero " + i;

            /* provo ad aggiungere un elemento alla coda */
            boolean aggiunto = queue.offer(elem);

            System.out.println("L'elemento " + i + " è stato aggiunto? " +
aggiunto);

            i++;

            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Consumer.java

```
import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {
    private BlockingQueue<String> queue;

    public Consumer(BlockingQueue<String> queue) {
        this.queue = queue;
    }

    @Override
```

```

        public void run() {
            while(true) {
                if (queue.remainingCapacity() > 0) {
                    System.out.println("E' possibile
aggiungere ancora " + queue.remainingCapacity() + " su " + queue.size());
                } else if (queue.remainingCapacity() == 0) {
                    String elementoRimosso = queue.remove();

                    System.out.println("E' stato rimosso
l'elemento " + elementoRimosso);
                }

                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

EsempioArrayBlockingQueue.java

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class EsempioArrayBlockingQueue {
    public static void main(String[] args) {
        // Creo una coda che puo contenere al massimo 10 elementi.
        BlockingQueue<String> queue = new ArrayBlockingQueue<String>(10);

        // Producer e Consumer accedono alla stessa coda...
        Thread prod = new Thread(new Producer(queue));
        Thread cons = new Thread(new Consumer(queue));

        prod.start();
        cons.start();
    }
}

```

Quindi, nella classe Producer abbiamo una variabile queue, che è di tipo BlockingQueue. Questa variabile viene inizializzata nel costruttore. All'interno del metodo run() abbiamo un ciclo while(true), che è un loop infinito finché non interrompiamo il software. All'interno del loop, il thread Producer prova ad aggiungere alla coda un elemento di tipo stringa, attraverso l'invocazione del metodo offer() definito dall'interfaccia BlockingQueue.

Nella classe Consumer abbiamo, analogamente alla classe Producer, una variabile queue di tipo BlockingQueue. Questa variabile viene inizializzata nel costruttore. All'interno del metodo run() abbiamo un ciclo while(true), che è un loop infinito finché non interrompiamo il software. All'interno del loop, il thread Consumer legge la coda e, se è piena, rimuove il primo elemento aggiunto (logica FIFO).

Poi abbiamo la classe EsempioArrayBlockingQueue che crea una variabile queue di tipo BlockingQueue, utilizzando la classe ArrayBlockingQueue e passando in ingresso la capacità massima di 10 elementi. Dopodiché vengono creati due thread, uno per il produttore e uno per il consumatore, e vengono avviati.

CLASSE LINKEDBLOCKINGQUEUE

La classe `LinkedBlockingQueue`, a differenza della classe `ArrayBlockingQueue`, consente di creare istanze senza specificare la capacità, cioè ha il costruttore che non prende in ingresso la capacità. In questo caso, la capacità massima sarà `Integer.MAX_VALUE`, ovvero $2^{31}-1$ elementi (2.147.483.647).

Chiaramente, se la coda non è mai piena, il metodo `put()` (o il metodo `offer()`), che inserisce elementi, non si può mai bloccare. Riprendiamo l'esempio precedente, ma sostituendo la classe

`EsempioArrayBlockingQueue` con `EsempioLinkedBlockingQueue`

`EsempioLinkedBlockingQueue.java`

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class EsempioLinkedBlockingQueue {
    public static void main(String[] args) {
        // Creo una coda senza specificare la capacità.
        BlockingQueue<String> queue = new LinkedBlockingQueue<String>();

        // Producer e Consumer accedono alla stessa coda...
        Thread prod = new Thread(new Producer(queue));
        Thread cons = new Thread(new Consumer(queue));

        prod.start();
        cons.start();
    }
}
```

In questo caso, il `Consumer` entrerà in gioco solo quando il `Producer` avrà inserito 2.147.483.647.

Notiamo che, in entrambi i casi, non abbiamo utilizzato la keyword `synchronized`, perché l'interfaccia `BlockingQueue` definisce dei metodi che sono già sincronizzati e sono implementati nelle classi `ArrayBlockingQueue` e `LinkedBlockingQueue`. Quindi, tali metodi consentono di gestire in automatico le code, senza l'utilizzo della keyword `synchronized`.

GESTIONE BANCONC SALUMI MEDIANTE THREAD POOL

Partiamo dallo scenario di riferimento:

Siamo i proprietari di un supermercato. Al bancone dei salumi e formaggi abbiamo 3 dipendenti. Nel supermercato ci sono 30 clienti che devono acquistare salumi e formaggi. Ovviamente i 30 clienti arrivano vicino al bancone e prendono dal distributore del ticket il proprio numero. Preso il numero, il cliente si mette in attesa di essere servito.

Dobbiamo considerare che:

- ogni cliente ha la propria lista della spesa, quindi ognuno di essi impiegherà un certo numero di minuti per essere servito, che può essere diverso dal tempo impiegato dagli altri clienti;
- tutti e 30 i clienti non possono essere serviti contemporaneamente, perché abbiamo solo 3 dipendenti;
- appena si libera un dipendente, il prossimo cliente viene servito.

Per la gestione del bancone dei salumi e formaggio in codice, utilizziamo ciò che segue:

- la classe `ArrayBlockingQueue`: utilizzata per creare l'oggetto che rappresenta la nostra coda al bancone dei salumi e formaggi;
- la classe `ExecutorService`: è la classe factory utilizzata per creare il thread pool. Avremo tanti thread per quanti sono i dipendenti disponibili al bancone;
- la classe `Cliente`: è la classe che rappresenta il generico cliente. Ogni cliente è un thread, pertanto questa classe implementa l'interfaccia `Runnable`.

Vediamo il codice per questo esempio:

Cliente.java

```
import java.util.Random;

public class Cliente implements Runnable {
    private int numeroTicket;

    public Cliente(int numeroTicket) {
        System.out.println("E' arrivato un nuovo cliente ed ha preso
il numero " + numeroTicket);

        this.numeroTicket = numeroTicket;
    }

    public void run() {
        /* il cliente ordina i prodotti al dipendente presente al
bancone */
        richiediProdotti();
    }

    private void richiediProdotti() {
        System.out.println("Viene servito il cliente numero " +
numeroTicket);

        /* imposto una durata random per ciascun cliente... */
        Random r = new Random();
```

```

        /* per semplicità ipotizzo che ogni cliente impieghi tra 5 e
        20 secondi per acquistare salumi e formaggi */
        int tempoImpiegatoPerAcquisto = (r.nextInt(15) + 5)*1000;

        try {
            /*
            * il thread viene sospeso per
            tempoImpiegatoPerAcquisto millisecondi.
            * Quest'attesa equivale al cliente che sta
            effettuando l'ordine al dipendente del bancone
            */
            Thread.sleep(tempoImpiegatoPerAcquisto);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Il cliente che aveva il numero " +
        numeroTicket + " ha completato il suo acquisto in " +
        tempoImpiegatoPerAcquisto/1000 + " secondi");
    }
}

```

BanconeSalumeriaFormaggi.java

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BanconeSalumeriaFormaggi {
    private final static int CLIENTI_DA_SERVIRE = 30;
    private final int DIPENDENTI_AL_BANCONE = 3;

    private BlockingQueue<Runnable> codaBancone = new
    ArrayBlockingQueue<Runnable>(30, true);
    private ExecutorService dipendentiDisponibili =
    Executors.newFixedThreadPool(DIPENDENTI_AL_BANCONE);

    public static void main(String[] args) {
        System.out.println("Nel supermercato ci sono " +
        CLIENTI_DA_SERVIRE + " clienti che stanno andando al bancone");

        BanconeSalumeriaFormaggi bancone = new
        BanconeSalumeriaFormaggi();
        bancone.arrivoClientiAlBancone();
        bancone.servizioClienti();
    }

    private void arrivoClientiAlBancone() {
        for (int i = 1; i <= CLIENTI_DA_SERVIRE; i++) {
            try {
                /* il cliente viene inserito in coda */
                codaBancone.put(new Cliente(i));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```



```

        private void servizioClienti() {
            new Thread(new Runnable() {
                public void run() {
                    while(true) {
                        try {
                            /* il primo cliente
disponibile viene servito ... */

                            dipendentiDisponibili.execute(codaBancone.take());
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }).start();
        }
    }

```

La classe Cliente implementa Runnable e ha una sola variabile, cioè numeroTicket. Per ogni istanza di Cliente, assegniamo un ticket, simulando proprio lo scenario in cui il cliente arriva al bancone e prende il proprio numero. Nel metodo run() abbiamo la chiamata al metodo richiediProdotti(). Il metodo richiediProdotti stampa che viene servito il cliente, imposta un tempo random per ciascun cliente, suppone che ogni cliente impieghi tra i 5 e 20 secondi per acquistare e, infine, il thread viene sospeso per il tempo impiegato per l'acquisto, simulando l'acquisto al bancone. Dopo che viene effettuato l'acquisto, il thread si risveglia e viene stampata che " il cliente che aveva il numero n ha completato il suo acquisto in m secondi".

Nella classe BanconeSalumeriaFormaggi abbiamo il numero di clienti da servire (30), il numero di dipendenti al bancone(3), la coda ArrayBlockingQueue che ha capacità 30 e i dipendenti disponibili e un thread pool di dimensione 3. Abbiamo anche i due metodi arrivoClientiAlBancone() e servizioClienti(). Quando viene creata, nel metodo main(), l'istanza della classe BanconeSalumeriaFormaggi, il primo metodo che viene invocato è arrivoClientiAlBancone(), che è un metodo che itera da 1 a CLIENTI_DA_SERVIRE e aggiunge alla coda un'istanza della classe Cliente (un nuovo thread), mentre il secondo metodo che viene invocato è servizioClienti(), in cui abbiamo un thread che viene avviato e permette di servire il primo cliente disponibile.

L'output del codice è il seguente:

Nel supermercato ci sono 30 clienti che stanno andando al bancone

```

E' arrivato un nuovo cliente ed ha preso il numero 1
E' arrivato un nuovo cliente ed ha preso il numero 2
E' arrivato un nuovo cliente ed ha preso il numero 3
E' arrivato un nuovo cliente ed ha preso il numero 4
E' arrivato un nuovo cliente ed ha preso il numero 5
E' arrivato un nuovo cliente ed ha preso il numero 6
E' arrivato un nuovo cliente ed ha preso il numero 7
E' arrivato un nuovo cliente ed ha preso il numero 8
E' arrivato un nuovo cliente ed ha preso il numero 9
E' arrivato un nuovo cliente ed ha preso il numero 10
E' arrivato un nuovo cliente ed ha preso il numero 11
E' arrivato un nuovo cliente ed ha preso il numero 12
E' arrivato un nuovo cliente ed ha preso il numero 13
E' arrivato un nuovo cliente ed ha preso il numero 14

```

E' arrivato un nuovo cliente ed ha preso il numero 15
E' arrivato un nuovo cliente ed ha preso il numero 16
E' arrivato un nuovo cliente ed ha preso il numero 17
E' arrivato un nuovo cliente ed ha preso il numero 18
E' arrivato un nuovo cliente ed ha preso il numero 19
E' arrivato un nuovo cliente ed ha preso il numero 20
E' arrivato un nuovo cliente ed ha preso il numero 21
E' arrivato un nuovo cliente ed ha preso il numero 22
E' arrivato un nuovo cliente ed ha preso il numero 23
E' arrivato un nuovo cliente ed ha preso il numero 24
E' arrivato un nuovo cliente ed ha preso il numero 25
E' arrivato un nuovo cliente ed ha preso il numero 26
E' arrivato un nuovo cliente ed ha preso il numero 27
E' arrivato un nuovo cliente ed ha preso il numero 28
E' arrivato un nuovo cliente ed ha preso il numero 29
E' arrivato un nuovo cliente ed ha preso il numero 30
Viene servito il cliente numero 1
Viene servito il cliente numero 2
Viene servito il cliente numero 3
Il cliente che aveva il numero 1 ha completato il suo acquisto in 6 secondi
Viene servito il cliente numero 4
Il cliente che aveva il numero 3 ha completato il suo acquisto in 10 secondi
Viene servito il cliente numero 5
Il cliente che aveva il numero 4 ha completato il suo acquisto in 5 secondi
Viene servito il cliente numero 6
Il cliente che aveva il numero 2 ha completato il suo acquisto in 14 secondi
Viene servito il cliente numero 7
....

Da qui capiamo che i 30 clienti vengono aggiunti alla coda, dopodichè vediamo quale cliente viene servito e in quanti secondi.

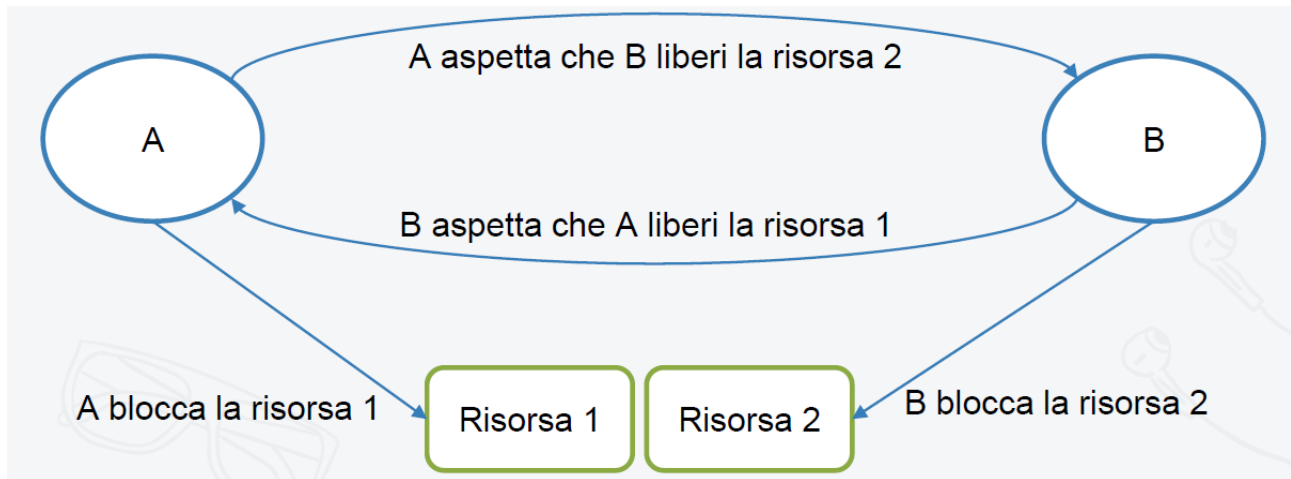
Riassumendo, in questo esempio abbiamo utilizzato la classe `ArrayBlockingQueue`, la coda `BlockingQueue` e il thread pool che indica il numero di thread che possiamo gestire simultaneamente. La coda è riempita da tutti i clienti. Un cliente è semplicemente un thread che effettua delle operazioni. Una volta serviti tutti e 30 i clienti, il software terminerà.

DEADLOCK, STARVATION E LIVELOCK

DEADLOCK

Il deadlock si ha quando un thread, che chiamiamo thread A, si blocca in attesa che il thread B liberi una risorsa condivisa tra i due e, a sua volta, il thread B resta bloccato in attesa che il thread A liberi un'altra risorsa. Quindi, il thread A aspetta che il thread B liberi una risorsa, mentre il thread B aspetta che il thread A liberi un'altra risorsa, bloccandosi a vicenda.

Il deadlock è uno di quei casi in cui non c'è via d'uscita.



STARVATION

La starvation si ha quando un thread non riesce mai ad acquisire le risorse di cui necessita , oppure ci riesce dopo troppo tempo, perché sono bloccate da altri thread.

Ad esempio, supponiamo di avere 3 thread A, B, C che accedono alla stessa risorsa R1. Supponiamo inoltre che i thread A e B hanno una priorità più alta di C, quindi hanno l'attributo priority con valore più alto rispetto C.

Finché A e B tentano di acquisire la risorsa R1, ovviamente C non può effettuare il lock e non la può utilizzare, perché A e B hanno la priorità più alta rispetto a C.

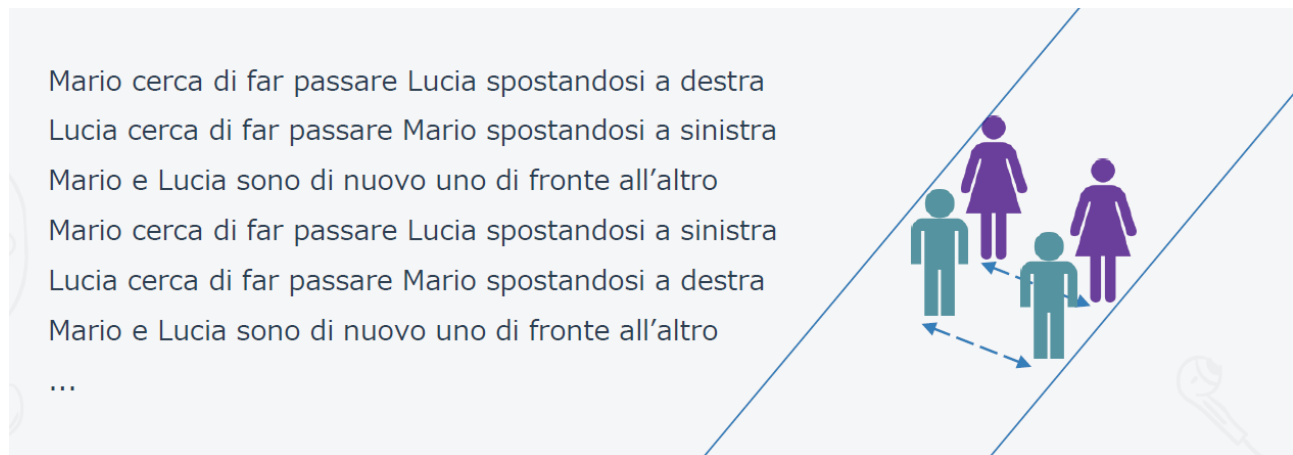
In questo caso C è soggetto a starvation.

LIVELOCK

Il livelock si ha quando 2 thread sono impegnati a risponderci reciprocamente e non sono in grado di proseguire nell'esecuzione del task.

In questo caso i thread non sono bloccati come accade nel deadlock, ma sono semplicemente occupati a fare altro.

L'esempio classico è quello di Mario e Lucia, che si incontrano sul marciapiede, uno di fronte all'altro.



Ovviamente, se uno tra Mario e Lucia non cambia strategia, non si andrà avanti.

Quando uno dei due si arresta e fa passare l'altro, a quel punto si ha lo sblocco della situazione.