3. PROGRAMMAZIONE CONCORRENTE, PROCESSI E THREAD

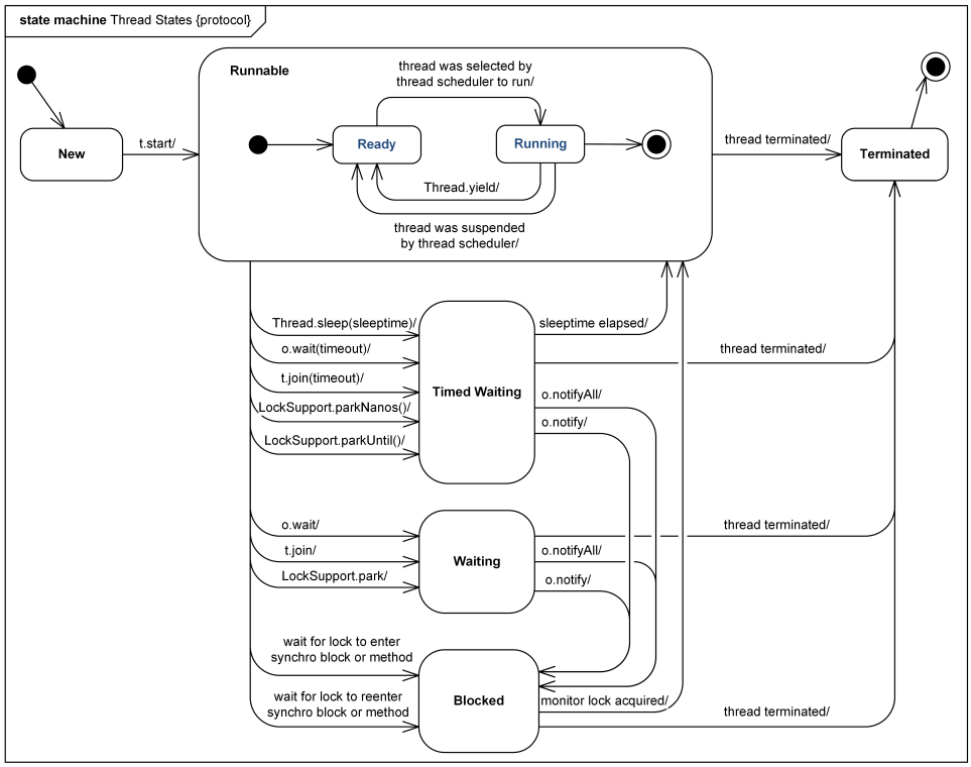
Nella programmazione concorrente, ci sono due unità di base di esecuzione: **processi** e **thread**, un sistema informatico ha molti processi e thread attivi. Esistono 3 tipi di programmazione concorrente:

1. Programmazione concorrente eseguita ***su calcolatori diversi***;
2. Processi concorrenti ***sulla stessa macchina***(***multitasking***);
3. Processo padre che ***genera processi figli per fork()***.

Un **processo** ha un ambiente di esecuzione autonomo, ovvero uno spazio di memoria privato di risorse di runtime di base.
Un singolo programma, in effetti può essere un insieme finito di processi cooperanti. Per facilitare la comunicazione tra i processi, la maggior parte dei sistemi operativi supporta le risorse **IPC** (**Inter Process Communication**), come pipe e socket.
I **thread** (anche chiamati *processi leggeri*) esistono all'interno di un processo (ne ha almeno uno, chiamato **main thread**) ed è un'unità di programma che viene eseguita indipendentemente, in più condividono le risorse del processo stesso, inclusa la memoria e file aperti, attraverso **memoria condivisa**, ciò rende la comunicazione efficiente, ma potenzialmente problematica.
Una **differenza sostanziale tra thread e processi** consiste nel modo con cui essi condividono le risorse, mentre i processi sono di solito fra loro indipendenti, utilizzando diverse aree di memoria ed interagendo soltanto mediante appositi meccanismi di comunicazione messi a disposizione dal sistema, al contrario i thread di un processo tipicamente condividono le medesime informazioni di stato, la memoria ed altre risorse di sistema.
L'altra differenza è che la creazione di un nuovo processo è sempre onerosa per il sistema, in quanto devono essere allocate risorse necessarie alla sua esecuzione (memoria, periferiche, e così via), il thread invece è parte di un processo e quindi una sua nuova attivazione viene effettuata in tempi ridotti e a costi minimi.
Il **multitasking** è la capacità di un sistema operativo di eseguire più compiti (processi) simultaneamente. Il **multithread** è l'estensione del multitask riferita ad un singolo programma in grado di eseguire più thread "contemporaneamente", principalmente esiste un "**Main Thread**" che a sua volta è capace di creare altri thread.
Alcune applicazioni che usano il **multithread** sono i browser che devono caricare dal server diverse immagini, visualizzare la pagina e reagire all'eventuale pulsante premuto dall'utente, oppure un'applicazione di rete che chiede dati ad un'altra applicazione, fornisce dati a chi li richiede e tenere informato l'utente dell'andamento delle operazioni.

3.1 THREAD IN JAVA

I thread in Java sono oggetti, istanze quindi di una classe **Thread**, ed hanno un proprio ciclo di vita:



New - Il thread viene creato.
Ready - Il thread è pronto *per essere eseguito*.
Running - Il thread (le sue istruzioni) *è in esecuzione*.
Timed waiting - Il thread è in attesa di un dato evento.
Waiting - Il thread entra in un'attesa indefinita
Blocked - Il thread è in stato blocked quando sta aspettando di acquisire il lock da un monitor.
Terminated - Il thread ha completato la sua esecuzione.

Un thread passa nello stato *runnable* quando viene chiamato il metodo *start()*.
Un thread passa nello stato **TimeWaiting**, **Waiting** o **Blocked** quando vengono chiamate le funzioni scritte sulle frecce.
Dopo che il thread ha completato l'esecuzione del metodo *run()*, esso viene trasferito nello stato **Terminated** .

Esistono due modalità di **gestione dei thread**:

1. Istanziare un oggetto thread ogni volta che serve un task asincrono (creazione e gestione a cura del programmatore);
2. Astrarre la gestione, passando un task ad un executor.

Noi ci focalizziamo sulla prima modalità, di base:

1. Estendere la classe **java.lang.Thread**;
2. Riscrivere (ridefinire, override) il **metodo run()** nella sottoclasse di Thread;
3. **Creare un'istanza** di questa classe derivata;
4. Richiamare il **metodo start()** su questa istanza.

Con questo approccio non è possibile estendere tale classe.

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Per quanto riguarda la seconda modalità:

1. Implementare l'**interfaccia Runnable**;
2. Riscrivere il **metodo run()** che viene eseguito ogni volta che si lancia il Thread;
3. L'oggetto istanziato è passato al **costruttore di Thread** e lanciato.

Tale approccio è utilizzabile anche per l'approccio con executors.

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Alcuni metodi utili:

- **`Thread.sleep`** fa sì che il thread corrente sospenda l'esecuzione per un periodo specificato. Tuttavia, non è garantito che questi tempi di sospensione siano precisi, perché sono limitati dalle funzionalità fornite dal sistema operativo sottostante. Inoltre, il periodo di **sleep** può essere interrotto da un **interrupt**.
  L'esempio utilizza sleep per stampare i messaggi a intervalli di quattro secondi, in più, il main può lanciare una eccezione che può generare la sleep quando un altro thread interrompe il thread corrente mentre sleep è attivo.

```
public class SleepMessages {
    public static void main(String args[])
            throws InterruptedException {
        String importantInfo[] = {
            "Mares", "eat", "Little", "kid"
        };
        for (int i=0; i<importantInfo.length; i++) {
            Thread.sleep(4000);
            System.out.println(importantInfo[i]);
        }
    }
}
```

Un **interrupt** indica ad un thread che dovrebbe fermare quello che sta facendo e fare qualcosa altro. Spetta al programmatore decidere esattamente come un thread risponde a un interrupt, ma è molto comune che il thread termini. Un thread invia un interrupt richiamando interrupt sull'oggetto Thread. Affinché il meccanismo di interruzione funzioni correttamente, il thread interrotto deve supportare la propria interruzione. Se il thread richiama frequentemente metodi che generano InterruptedException, vengono restituiti dal metodo run dopo aver rilevato l'eccezione.
Ad esempio, nella sleep() si intercetta l'eccezione e in tal caso si esce altrimenti si stampa.

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

- **`Thread.interrupted()`**, un thread che non invoca un metodo che lancia l'eccezione InterruptedException può controllare se è stato interrotto. Nell'esempio, se è stato ricevuto un interrupt allora si lancia l'eccezione gestita in una unica catch() centralizzata.

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

- **`t.join()`** consente a un thread di attendere il completamento di un altro thread. Se *t* è un oggetto Thread il cui thread è attualmente in esecuzione, allora *t.join()* fa sì che il thread corrente sospenda l'esecuzione fino a quando il thread *t* non termina.

```
//…
   t.join();
//…
```

Esempio SimpleThread:

```
public class SimpleThreads {
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }
    private static class MessageLoop implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares", "eat", "Little", "kid"
            };
            try {
                for (int i=0; i<importantInfo.length; i++) {
                    Thread.sleep(4000);
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }
    public static void main(String args[]) throws InterruptedException{
        long patience = 1000 * 60 * 60;
        if (args.length > 0) {
            try {
                patience = Long.parseLong(args[0]) * 1000;
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
            }
        }
        threadMessage("Starting MessageLoop thread");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();
        threadMessage("Waiting for MessageLoop thread to finish");
        while(t.isAlive()) {
            threadMessage("Still waiting...");
            t.join(1000);
            if (((System.currentTimeMillis()-startTime)>patience)
                  && t.isAlive()) {
                threadMessage("Tired of waiting!");
                t.interrupt();
                t.join();
            }
        }
        threadMessage("Finally!");
    }
}
```

SimpleThreads consiste di due thread, il primo è il thread principale di ogni applicazione Java che crea un nuovo thread dall'oggetto *Runnable*, *MessageLoop*, e attende che finisca.
Se il thread *MessageLoop* impiega troppo tempo per terminare, il thread principale lo interrompe.

Tale programma mostra un messaggio col nome del thread, grazie a format, eseguiamo una stampa formattata.
Implementiamo un'interfaccia e riscriviamo run() che viene eseguito allo start del thread.
Al suo interno abbiamo 4 stringhe e poi una sleep() interna ad un blocco try/catch.
Tale blocco stamperà le stringhe con un ritardo di 4 secondi.
Se, nel mentre, si verifica un interrupt, e quindi un'eccezione, il try la cattura e stamperà "*I wasn't done!*".

Osservando il main, questo può lanciare eccezioni.
All'interno viene calcolata la variabile *patience* che conserva il ritardo della stampa.
All'interno del *catch* si effettua il controllo del formato.

La variabile *startTime* prende il tempo di inizio.

Infine, viene creato l'oggetto Thread da *MessageLoop* e lo si fa partire richiamando *start()*.

Mentre *MessageLoop* è in vita, grazie al *while* che lo controlla con *isAlive()*, aspetta al più 1 secondo, specificato dal metodo *join()*.

Se la "pazienza" è scaduta (controllato dall'if) e il thread è ancora vivo, allora lo si chiude con *t.interrupt()*, e se ne attende la "fine" tramite *t.join()*, ed infine si esce stampando "*Finally*!".

3.2 PROBLEMATICHE COI THREAD

I thread comunicano principalmente condividendo accesso a campi (tipi primitivi) e campi che contengono riferimenti a oggetti.
Questa forma di comunicazione è estremamente efficiente, ma rende possibili due tipi di errori:

1. **Interferenza** di thread;
2. **Inconsistenza** della memoria.

La **prima** problematica (l'**interferenza**) si verifica quando due operazioni, eseguite in thread diversi che agiscono sugli stessi dati, si *interfogliano*.
Avendo questo blocco di codice, se *Counter* fa riferimento a un oggetto da più thread, l'interferenza tra i thread può impedire che ciò accada come previsto.

```
class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

Supponiamo che il thread A invoca *increment* circa nello stesso momento in cui invoca *decrement*:

1. Thread A: Recupera c.
2. Thread B: Recupera c.
3. Thread A: incremento del valore recuperato; il risultato è 1.
4. Thread B: decrementa del valore recuperato; il risultato è -1.
5. Thread A: Memorizza il risultato in c; c ora è 1.
6. Thread B: Memorizza il risultato in c; c è ora -1.

Il risultato del thread A viene perso, sovrascritto dal thread B, ma può succedere anche il contrario.
Questo scenario porta al cosiddetto **race condition**, cioè quando il risultato di una operazione dipende dall'ordine di esecuzione di diversi thread.
La **seconda** problematica (l'**inconsistenza**) avviene quando thread diversi hanno visioni diverse degli stessi dati.
Fortunatamente, il programmatore non ha bisogno di una comprensione dettagliata di queste cause, tutto ciò che serve è una strategia per evitarli.
Una di queste è la relazione **happens-before**, che è una garanzia che la memoria scritta da un thread è visibile da un altro thread.
Un esempio →
Il campo *counter* è condiviso tra due thread, A e B, se le due istruzioni vengono eseguite in thread separati, il valore stampato *potrebbe* essere "0", perché non c'è garanzia che la modifica del thread A sarà visibile al thread B, e quindi bisogna stabilire una relazione happens-before.

```
int counter = 0;
//…
counter++;
//…
System.out.println(counter);
```

Esempio Inconsistenza della memoria:

```
class Foo {
   int bar = 0;
   public static void main(String args[]){
      (new Foo()).unsafeCall();
   }
   void unsafeCall () {
      final Foo thisObj = this;
      Runnable r = new Runnable () {
         public void run (){
            thisObj.bar = 1;
         }
      };
      Thread t = new Thread(r);
      t.start( );
      try{
         Thread.sleep(1000);
      }catch(InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println("bar="+ bar );
   }
}
```

Dichiariamo una variabile *bar*, definiamo *unsafeCall* nel main.

unsafeCall definisce un thread che mette la variabile *bar* a "1".

Dopo si fa partire il thread, ed il main thread attende 1 sec.

Se si lancia questo programma, parte il main thread che lancia il thread interno che imposta la variabile *bar* ad "1", e il main thread attende 1 sec e poi termina.
La stampa non è precisa, non è detto che il thread fa in tempo a stampare il valore corretto della variabile, potrebbe essere 0 o 1.

Per realizzare happens-before lo si fa tramite la **sincronizzazione**, più precisamente, tramite il metodo ***start()*** e ***join()***.
Un'altra soluzione è rendere la **variabile volatile**, cioè modificarne il valore influisce immediatamente sulla memoria. Ciò garantisce che quando un thread modifica la variabile, tutti gli altri thread vedano immediatamente il nuovo valore. La **keyword volatile** è di solito associata ad una variabile il cui valore viene salvato e ricaricato in memoria ad ogni accesso senza utilizzare i meccanismi di **caching**.

Esempio variabile volatile:

```
public class VolatileTest {
  volatile boolean running = true; // notare la kyword volatile
  public void test() {             //lancio primo Thread
    new Thread(new Runnable() {
      public void run() {
        int counter = 0;
        while (running)
          counter++;
        System.out.println("Thread 1 concluso.Contatore "+counter);
      }
    }).start();
    new Thread(new Runnable() {    //lancio secondo Thread
      public void run() {
        try {
          Thread.sleep(100);
        } catch (InterruptedException ignored) { }
        System.out.println("Thread 2 concluso");
        running = false;
      }
    }).start();
  }
  public static void main(String[] args) {
    new VolatileTest().test();
  }
}
```

All'interno di test(), lanciamo il primo thread che mette il contatore a "0", poi finché running è "true", incrementa il contatore e dopo stampa il suo valore.

Lo sleep() nel secondo thread è necessario per dare al primo thead la possibilità di partire. Questo secondo thread mette running a "false".

Se la variabile *running* non era volatile, il primo thread non termina mai, perché viene letto sempre il valore della cache, ovvero "true", cioè le modifiche che fa il secondo thread non vengono viste dal primo thread.
Dichiarando volatile la variabile running invece si costringe il Thread (o chi per esso) ad aggiornare di volta in volta il valore senza memorizzarlo in cache.

3.3 SINCRONIZZAZIONE DEI THREAD IN JAVA

Il linguaggio di programmazione Java fornisce due idiomi di sincronizzazione di base: **metodi sincronizzati** e **synchronized statements**.

METODI SINCRONIZZATI:

I **metodi sincronizzati** (***synchronized***) sono un costrutto del linguaggio Java, che permette di risolvere gli errori di concorrenza, al costo di inefficienza.

Per rendere un metodo sincronizzato, basta aggiungere ***synchronized*** alla sua dichiarazione →

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment(){
        c++;
    }
    public synchronized void decrement(){
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Grazie a questa tecnica non è possibile che due esecuzioni dello stesso metodo sullo stesso oggetto siano *interfogliate*. Quando un thread esegue un metodo sincronizzato per un oggetto, gli altri thread che invocano metodi sincronizzati dello stesso oggetto sono sospesi fino a quando il primo thread non ha finito. Quando un thread esce da un metodo sincronizzato, allora si stabilisce una relazione **happens-before** con tutte le successive invocazioni dello stesso metodo sullo stesso oggetto, cioè *i cambi di stato effettuati dal thread appena uscito sono visibili a tutti i thread*.

SYNCHRONIZED STATEMENT:

Per realizzare i **synchronized statements** si utilizza il **lock intrinseco**, che è una entità associata ad ogni oggetto e garantisce sia accesso esclusivo sia accesso consistente (relazione **happens-before**).

Per convenzione, un thread che necessita di un accesso esclusivo deve acquisire il lock intrinseco dell'oggetto prima di accedervi e quindi rilasciare il lock quando ha terminato. Finché un thread possiede un **lock intrinseco**, nessun altro thread può acquisire lo stesso blocco (o statement), l'altro thread si bloccherà quando tenterà di acquisire il blocco.

Quando un thread esegue un metodo sincronizzato di un oggetto ne acquisisce il lock, e lo rilascia al termine (anche se c'è una eccezione).

A differenza dei metodi sincronizzati, gli statement sincronizzati devono specificare l'oggetto che fornisce il lock intrinseco.
In questa maniera, si sincronizzano gli accessi solo durante la modifica, ma poi si provvede in maniera concorrente all'inserimento in lista.

Gli statement sincronizzati sono utili anche per migliorare la concorrenza con la sincronizzazione.

Nelle funzioni inc1() e inc2(), se c'era synchronized prima del nome, si sequenzializza tutto, mentre, se c'era synchronized(this), non sarebbero stati indipendenti.
In questo modo, i lock vengono acquisiti sugli oggetti.

```
public void addName(String name){
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

L'ultimo meccanismo per garantire la sincronizzazione, sono le **azioni atomiche**.

Nella programmazione, un'**azione atomica** è un'azione che effettivamente avviene tutta in una volta e non sono interrompibili. Un'azione atomica non può fermarsi nel mezzo: o viene completamente, o non avviene affatto. Nessun effetto collaterale di un'azione atomica è visibile fino al suo completamento. Ad esempio, le operazioni di read e write.

Tuttavia, ciò non elimina la necessità di sincronizzare le azioni atomiche, poiché sono ancora possibili errori di coerenza della memoria. L'utilizzo delle variabili volatili riduce il rischio di errori di consistenza della memoria, poiché qualsiasi scrittura su una di essa stabilisce una relazione happens-before con le letture successive della variabile stessa. Ciò significa che le modifiche a una variabile volatile sono sempre visibili agli altri thread.

Stesso codice di prima, ma col package ***java.util.concurrent.atomic***, questo ci garantisce azioni atomiche, e quindi la sincronizzazione.

```
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

3.4 PROBLEMATICHE DELLA SICRONIZZAZIONE DI THREAD

Possono avvenire alcune problematiche durante la sincronizzazione di thread, una di queste è il **deadlock**, che descrive una situazione in cui due o più thread sono bloccati per sempre, in attesa l'uno dell'altro.

Un esempio →

Alphonse e Gaston sono due amici molto cortesi ed hanno una rigida regola di cortesia: quando ti inchini ad un amico, devi rimanere inchinato finché il tuo amico non ha una possibilità di restituire l'inchino. Ma se i metodi *bow* (inchino) vengono invocati assieme, entrambi i thread si bloccheranno invocando *bowBack*.

Quando Deadlock viene eseguito, è estremamente probabile che entrambi i thread si blocchino quando tentano di invocare *bowBack*. Nessuno dei due blocchi finirà mai, perché ogni thread aspetta che l'altro esca da *bow*.

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name){    this.name = name;   }
        public String getName(){    return this.name;    }

        public synchronized void bow(Friend bower) {
            System.out.format("%s:%s"+" has bowed to me!%n",this.name,bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s:%s"+" has bowed back to me!%n",this.name,bower.getName());
        }
    }
    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");

        new Thread(new Runnable(){ public void run() { alphonse.bow(gaston); }}).start();
        new Thread(new Runnable(){ public void run() { gaston.bow(alphonse); }}).start();
    }
}
```

Riscriviamo i metodi syncronized in questa maniera →

Così che il lock degli oggetti sono espliciti.

A questo punto Alphonse acquisisce il suo lock e cerca di acquisire quello di Gaston, questo fa lo stesso, ovvero prima il suo e poi quello di Alphonse.

```
public void bow(Friend bower){
    synchronized(this) {
        System.out.format("%s:%s has bowed to me!%n", this.name, bower.getName());
        bower.bowBack(this);
    }
}
public void bowBack(Friend bower){
    synchronized(this) {
        System.out.format("%s:%s has bowed back to me!%n",this.name, bower.getName());
    }
}
```

Altre problematiche, meno comuni, sono:

- **Starvation** (*ingordo*), descrive una situazione in cui un thread non è in grado di ottenere l'accesso regolare alle risorse condivise e non è in grado di fare progressi. Ciò accade quando le risorse condivise vengono rese non disponibili per lunghi periodi da thread "*avidi*". Ad esempio, si supponga che un oggetto fornisca un metodo sincronizzato che spesso impiega molto tempo per essere restituito. Se un thread richiama questo metodo frequentemente, altri thread che richiedono anche un accesso sincronizzato frequente allo stesso oggetto verranno spesso bloccati.
- **Livelock**, un thread A può reagire ad azioni di un altro thread B che reagisce con una risposta verso A. I due thread non sono bloccati (non è un deadlock!) ma sono occupati a rispondere alle azioni dell'altro, anche se sono in esecuzione, non c'è progresso. Un esempio è due persone che si incontrano in un corridoio stretto, sullo stesso lato, se le due persone hanno un "*attitudine belligerante*", entrambi aspettano che l'altro si sposti, ma questo provoca un deadlock, mentre se hanno un "*attitudine garbata*", entrambi si possono spostare sullo stesso lato, bloccandosi ancora e continuare a spostarsi all'infinito, e questo è un livelock.

3.5 EFFICIENZA DEI THREAD

Una non efficienza dei thread è il cosiddetto **race condition**, ovvero che il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti.

È necessario comprendere bene in che maniera i metodi synchronized sono eseguiti in mutua esclusione. Per questo motivo presentiamo un esempio, semplice, che al variare di alcuni semplici keyword si comporta in maniera diversa.

Nell'esempio →

Abbiamo un oggetto *Example*, creiamo 2 *SimpleThread* e li lanciamo col metodo *start()*. Questo metodo farà partire l'esecuzione dei 2 thread, ovvero richiameranno il metodo *run()*.

Nella classe *Example* abbiamo due metodi sincronizzati (*firstWaitingRoom* e *secondWaitingRoom*), che non fanno altro che stampare l'ingresso all'interno dei metodi, si attende qualche secondo specificato ed infine stampa quando si esce.

```
import java.util.logging.Logger;
public class Example {
    private static Logger log = Logger.getLogger("Example");
    public static void main(String[] args) {
        Example ob = new Example();
        log.info("Creating threads");
        SimpleThread one = new SimpleThread("one", ob,2);
        SimpleThread two = new SimpleThread("two", ob,2);
        one.start();
        two.start();
    }
    public synchronized void firstWaitingRoom(int sec, String name){
        log.info(name+"..entering.");
        try{
            Thread.sleep(sec*1000);
        }catch(InterruptedException e) {
            e.printStackTrace();
        }
        log.info(name+"..exiting.");
    }
    public synchronized void secondWaitingRoom(int sec, String name){
        log.info(name+"..entering.");
        try{
            Thread.sleep(sec*1000);
        }catch(InterruptedException e) {
            e.printStackTrace();
        }
        log.info(name+"..exiting.");
    }
}
```

La classe *SimpleThread* prevede, oltre a variabili di istanza e costruttore, il metodo *run()* che calcola un valore a caso e se questo valore è "0" allora viene avviato il metodo *firstWaitingRoom* altrimenti *secondWaitingRoom*.

```
public class SimpleThread extends Thread {
    private String name;
    private Example object;
    private int delay;
    public SimpleThread(String n,Example obj,int del){
        name = n;
        object = obj;
        delay = del;
    }
    public void run() {
        double y = Math.random();
        int x = (int) (y * 2);
        if(x==0)
            object.firstWaitingRoom(delay, name);
        else//x==1
            object.secondWaitingRoom(delay, name);
    }
}
```

In questo esempio, abbiamo vari casi possibili di esecuzione, in quanto ci sono 2 thread che cercano di accedere a due metodi(*firstWaitingRoom* e *secondWaitingRoom*) oppure entrambi allo stesso.

In generale, si possono provare strategie differenti per ***provare*** a risolvere l'interfogliamento:

1. **DUE METODI DI INSTANZA SINCRONIZZATI**:

Supponiamo che una volta debbano accedere alla stessa sala d'attesa e un'altra a diverse sale, ed entrambi i metodi sono sincronizzati.
Nel momento in cui entrambi sono sincronizzati, succede che è come diventassero sequenziali.

```
public synchronized void firstWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
      Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
      e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
public synchronized void secondWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
      Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
      e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
```

Output di accesso alla stessa sala

```
6-ott-2013 17.14.51 Example main
INFO: Creating threads
6-ott-2013 17.14.51 Example secondWaitingRoom
INFO: one.. entering.
6-ott-2013 17.14.53 Example secondWaitingRoom
INFO: one.. exiting.
6-ott-2013 17.14.53 Example secondWaitingRoom
INFO: two.. entering.
6-ott-2013 17.14.55 Example secondWaitingRoom
INFO: two.. exiting.
```

Output di accesso a sale diverse

```
6-ott-2013 17.15.08 Example main
INFO: Creating threads
6-ott-2013 17.15.08 Example firstWaitingRoom
INFO: one.. entering.
6-ott-2013 17.15.10 Example firstWaitingRoom
INFO: one.. exiting.
6-ott-2013 17.15.10 Example secondWaitingRoom
INFO: two.. entering.
6-ott-2013 17.15.12 Example secondWaitingRoom
INFO: two.. exiting.
```

2. **UN SOLO METODO SINCRONIZZATO**:

Supponiamo che un solo metodo sia sincronizzato e che si accede a due sale di attesa diverse, non ci sarà mutua esclusione.

```
public synchronized void firstWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
      Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
      e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
public void secondWaitingRoom(int sec, String name){
    log.info(name+"..entering.");
    try{
      Thread.sleep(sec*1000);
    }catch(InterruptedException e) {
      e.printStackTrace();
    }
    log.info(name+"..exiting.");
}
```

Output:

```
6-ott-2013 17.14.04 Example main
INFO: Creating threads
6-ott-2013 17.14.04 Example secondWaitingRoom
INFO: one.. entering.
6-ott-2013 17.14.04 Example firstWaitingRoom
INFO: two.. entering.
6-ott-2013 17.14.06 Example firstWaitingRoom
INFO: two.. exiting.
6-ott-2013 17.14.06 Example secondWaitingRoom
INFO: one.. exiting.
```

Esempio efficienza Thread:

```
public class ArrayInit extends Thread{
    public static int[] data;
    public static final int SIZE = 10000000;
    public static void main(String[] args){
        data = new int[SIZE];
        long begin, end;
        int j;
        begin = System.currentTimeMillis();
        for(int i = 0; i < SIZE; i++){
            for(j=0; j < 10000; j++)
                data[i] = i;
        }
        end = System.currentTimeMillis();
        System.out.println("Time:"+ (end - begin) + "ms");
    }//end main
}//end class
```

Tale esempio non fa altro che inizializzare 10 mila volte 10 milioni di elementi, e permette ci calcolare il tempo impiegato.

Possibile output:

```
Time 20 ms
```

```
public class EffInit extends Thread {
    private int start;
    private int dim;
    public static int[] data;
    public static final int SIZE = 10000000;
    public static final int MAX_THR = 16;
    public static void main(String[] args){
        data = new int[SIZE];
        long begin, end;
        int start, j;
        EffInit[] threads;
        for(int numThread = 1; numThread <= MAX_THR; numThread++) {
            begin = System.currentTimeMillis();
            start = 0;
            threads = new EffInit[numThread];
            for (j = 0; j < numThread; j++) {
                threads[j] = new EffInit(start, SIZE/numThread);
                threads[j].start();
                start += SIZE / numThread;
            }//endfor
        }
```

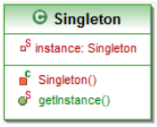
Ora vogliamo ottimizzare questa operazione, e quindi dividiamo l'array in blocchi e si fanno partire diversi thread, così che ognuno di questi thread si occuperà di inizializzare un sottoinsieme dell'array, ottimizzando così il tempo.

Possibile output:

```
1 Thread(s): 65ms
2 Thread(s): 11ms
3 Thread(s): 6ms
4 Thread(s): 7ms
5 Thread(s): 6ms
6 Thread(s): 4ms
7 Thread(s): 7ms
8 Thread(s): 5ms
```

```
            for (int numThread = 1; numThread <= MAX_THR; numThread++) {
                begin = System.currentTimeMillis();
                start = 0;
                threads = new EffInit[numThread];
                for (j = 0; j < numThread; j++) {
                    threads[j] = new EffInit(start, SIZE / numThread);
                    threads[j].start();
                    start += SIZE / numThread;
                }//end for
                for (j = 0; j < numThread; j++) {
                    try {
                        threads[j].join();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }//end for
                end = System.currentTimeMillis();
                System.out.println(numThread+"Thread(s): "+(end-begin)+"ms");
            }
    }//end main
        public EffInit(int start,int size){
            this.start = start;
            this.dim = size;
        }//end costruttore
        public void run(){
            int j;
            for(int i = 0; i <this.dim; i++){
                for(j=0; j < 10000; j++)
                    data[this.start + i] = i;
            }//end for
        }
}
```

3.6 SINGLETON E DOUBLE-CHECKED LOCKING



Il **Singleton**, noto design pattern della programmazione ad oggetti, permette di restringere l'istanziazione da parte di una classe ad un solo oggetto.
Dei possibili usi sono le memorizzazioni di stato, come Printer, File, Resource Manager, ….
Questo pattern, però, soffre si **lazy allocation**, ovvero l'allocazione avviene solo quando utilizzato per la prima volta.
Ci sono diverse possibili implementazioni della classe **Singleton**:

1.
C'è una variabile di classe che mantiene l'istanza, un metodo di classe che restituisce un Singleton, se non è stato ancora creata una istanza allora la crea, alla fine, restituisci la nuova istanza a disposizione.
Con questa implementazione, se ci sono più thread, il loro *interfogliamento* può creare errori, perché si potrebbero creare più Singleton, cosa inaccettabile.

```
class Singleton {
    private static Singleton instance;
    private Singleton() { /*...*/ }
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

2.
Simile al precedente, ma il metodo è sincronizzato.
Questo dovrebbe risolvere il problema, ma non è così, perché la sincronizzazione serve solo la prima volta che viene invocato il metodo.

```
class Singleton {
  private static Singleton instance;
  private Singleton() { /*...*/ }
  public static synchronized Singleton getInstance(){
      if(instance == null) {
          instance = new Singleton();
      }
      return instance;
  }
}
```

3.
Sempre simile alla prima, ma col lock esplicito interno al metodo.
Ma anche questo non funziona…, perché se due thread A e B arrivano ad eseguire l'if, assumiamo che A entra nella sezione critica, crea l'istanza ed esce restituendo il "primo" Singleton, ma a questo punto B entra e crea un nuovo Singleton che sovrascrive il primo.

```
class Singleton {
  private static Singleton instance;
  private Singleton() { /*...*/ }
  public static Singleton getInstance(){
      if(instance == null) {
          synchronized(Singleton.class) {
              instance = new Singleton();
          }
      }
      return instance;
  }
}
```

DOUBLE-CHECKING LOCKING:
Con questa strategia effettuiamo due volte il controllo.
Nonostante pubblicato e ampiamente utilizzato, non è garantito che funzioni.
La chiamata al costruttore "sembra" essere prima dell'assegnazione di *instance*, ma questo non è detto che accada. Un Singleton non inizializzato potrebbe essere assegnato a *instance* e se un altro thread controlla il valore poi lo può usare (in maniera scorretta).

```
public static Singleton getInstance(){
    if(instance == null) {
        synchronized(Singleton.class) {
            if(instance == null)
                instance = new Singleton();
        }
    }
    return instance;
}
```

Per risolvere questi problemi è possibile rendere la variabile *instance* **volatile**, altrimenti si possono usare le classi statiche "**Initialization-on-demand holder**".

Domande che fa all'orale:

- **Ciclo di vita di un thread?**
- **Cosa significa interferenza e inconsistenza della memoria?**
- **Cos'è la relazione happens-before?**