I Thread in Java

I threads in Java

- Ogni programma Java contiene almeno un *singolo thread*, corrispondente all'esecuzione del metodo *main()* sulla *JVM.*
- E' possibile creare dinamicamente nuovi thread *attivando concorrentemente* le loro esecuzioni all'interno del programma.

Due possibilita` di creazione:

1. Thread come oggetti di sottoclassi della classe **`Thread`**
2. Thread come oggetti di classi che implementano l'interfaccia **`runnable`**

Thread come oggetti di sottoclassi della classe `Thread`

- I threads sono oggetti che derivano dalla **classe Thread** (fornita dal package **java.lang).**
- Il metodo **run** della classe di libreria **Thread** definisce *l'insieme di statement Java* che ogni thread (oggetto della classe) eseguirà *concorrentemente* con gli altri thread.
- Nella classe **Thread** l'implementazione del suo metodo **run** è vuota.
- In ogni sottoclasse derivata da **Thread** deve essere ridefinito (*override*) il metodo **run** in modo da fargli eseguire ciò che è richiesto dal programma

Possibile schema

```
class AltriThreads extends Thread {
  public void run() {
    <corpo del programma eseguito>
    <da ogni thread di questa classe>
    }
}

public class EsempioConDueThreads
{     public static void main (string[] args)
      {  AltriThreads t1=new AltriThread();
         t1.start();
         <resto del programma eseguito
            dal thread main>
      }
}
```

- La classe **AltriThread** (estensione di Thread) implenta i nuovi thread *ridefinendo il metodo* **run***.*
- La classe EsempioConDueThreads fornisce il **main** nel quale viene creato il thread **t1** come oggetto derivato dalla classe **Thread.**
- Per **attivare** il thread deve essere eseguito il metodo **start()** che invoca il metodo **run()** (il metodo **run()** non può essere chiamato direttamente, ma solo attraverso **start()**).
- JVM gestisce *due thread concorrenti*: il thread principale associato al **main** ed il thread **t1.**

E se occorre definire thread che non siano necessariamente **sottoclassi di Thread?**

Thread come classe che implementa l'interfaccia `runnable`

Interfaccia Runnable: maggiore **flessibilità** → thread come sottoclasse di qualsiasi altra classe

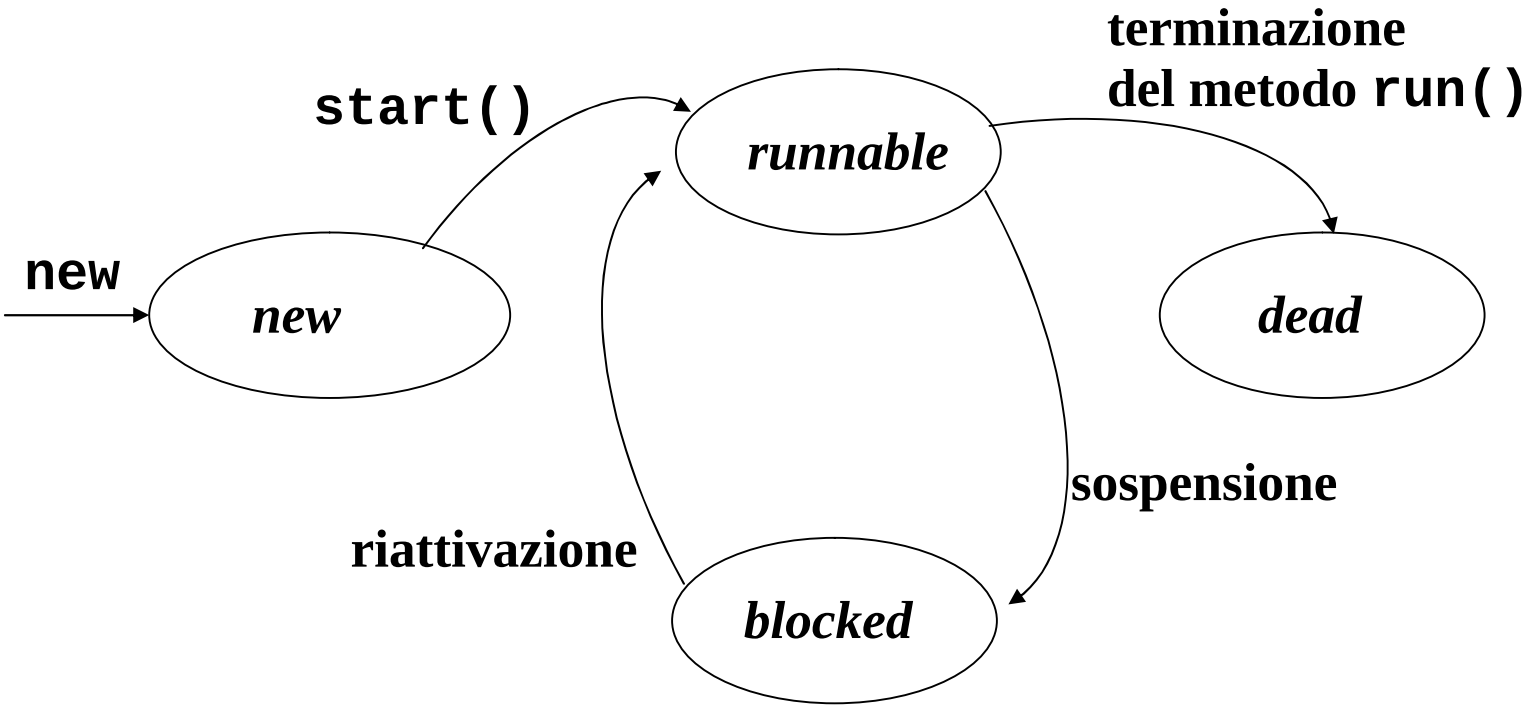
- implementare il metodo **`run()`** nella classe che implementa l'interfaccia **`Runnable`**
- creare un'istanza della classe tramite **`new`**
- creare un'istanza della classe **`Thread`** con un'altra **`new`**, passando come parametro l'istanza della classe che si è creata
- invocare il metodo **`start()`** sul thread creato, producendo la chiamata al suo metodo **`run()`**

Esempio di classe `EsempioRunnable` che *implementa l'interfaccia Runnable* **ed è sottoclasse di `MiaClasse`:**

```
class EsempioRunnable extends MiaClasse implements Runnable
{
   // non e' sottoclasse di Thread
   public void run()
   {
        for (int i=1; i<=10; i++)
             System.out.println(i + " " + i*i);
   }
}

public class Esempio
{
   public static void main(String args[])
   {
        EsempioRunnable e = new EsempioRunnable();
        Thread t = new Thread(e);
        t.start();
   }
}
```

Grafo di stato di un thread



Priorità e scheduling

- **Preemptive** priority scheduling con priorità fisse (crescenti verso l'alto).
- MIN-PRIORITY, MAX-PRIORITY: costanti definite nella classe thread.
- Ogni thread eredita, all'atto della sua creazione, la priorità del processo padre.
- Metodo **`set-priority`** per modificare il valore della priorità

JVM esegue l’algoritmo di scheduling:

- quando il thread correntemente in esecuzione esce dallo stato *runnable* (sospensione o terminazione);

- quando diventa *runnable* un thread a priorità più alta (preemption).

➔ JVM non supporta l'assegnazione della CPU per quanti di tempo (*round robin*).

- SE:
  - S.O. adotta round-robin: i thread di uguale priorità vengono gestiti *round robin* (anziché FIFO).
  - S.O non adotta round-robin: è possibile simulare a programma tale comportamento. Metodo *yield() (cooperative multithreading)*.

- *Trasparenza della JVM rispetto alla gestione dei quanti di tempo*: potenziale problema per quanto riguarda la portabilità di applicazioni Java che adottano diversi criteri di scheduling

Metodi per il controllo di thread

- **`start()`** fa ***partire*** l'esecuzione di un thread. La macchina virtuale Java invoca il metodo `run()` del thread appena creato
- **`stop()`** ***forza*** la ***terminazione*** dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente ***liberate*** (lock inclusi), come effetto della propagazione dell'eccezione `ThreadDeath`
- **`suspend()`** ***blocca*** l'esecuzione di un thread in attesa di una successiva operazione di **`resume()`**. Non libera le risorse impegnate dal thread (lock inclusi)
- **`resume()`** ***riprende*** l'esecuzione di un thread precedentemente ***sospeso***. Se il thread riattivato ha una priorità maggiore di quello correntemente in esecuzione, avrà subito accesso alla CPU, altrimenti andrà in coda d'attesa

- **`sleep(long t)`** blocca per un ***tempo specificato*** (`time`) l'esecuzione di un thread. Nessun *lock* in possesso del thread viene rilasciato.
- **`join()`** ***blocca*** il thread chiamante in attesa della ***terminazione*** del thread di cui si invoca il metodo. Anche con ***timeout***
- **`yield()`** ***sospende*** l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in ***coda d'attesa***

  I metodi precedenti interagiscono con il ***gestore della sicurezza*** della macchina virtuale Java

Il problema di `stop()` e `suspend()`

`stop()` e `suspend()` rappresentano azioni “brutali” sul ciclo di vita di un thread → rischio di determinare situazioni inconsistenti o di blocco critico (***deadlock***)

- se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva***, tale risorsa rimane ***bloccata*** e non è utilizzabile da altri, perché il thread sospeso non ha avuto modo di rilasciare il *lock* su di essa
- se il ***thread interrotto*** stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera ***atomica***, l’interruzione può condurre ad uno ***stato inconsistente*** del sistema

➔ JDK 1.4, pur supportandoli ancora per ragioni di *back-compatibility*, ***sconsiglia*** l'utilizzo dei metodi `stop(), suspend() e resume()` (***metodi deprecated***)

Si consiglia invece di realizzare tutte le azioni di ***controllo*** e ***sincronizzazione*** fra thread tramite gli stumenti specifici per la sincronizzazione (object locks, **`wait(), notify(), notifyAll()`** e variabili condizione)

Sincronizzazione in Java

Modello a memoria comune:

I threads di una applicazione condividono lo spazio di indirizzamento.

➔ Ogni tipo di interazione tra thread avviene tramite *oggetti* comuni:

- Interazione di tipo competitivo (*mutua esclusione*): meccanismo degli **objects locks.**
- Interazione di tipo cooperativo:
  - meccanismo **wait-notify.**
  - **variabili condizione**

Mutua esclusione

- Ad ogni oggetto viene associato dalla JVM un **lock** (analogo ad un semaforo binario).
- E' possibile denotare alcune sezioni di codice che operano su un oggetto come *sezioni critiche* tramite la parola chiave **`synchronized`.**

➔ Il compilatore inserisce :

  – un prologo in testa alla sezione critica per **l'acquisizione del lock** associato all'oggetto.
  – un epilogo alla fine della sezione critica per **rilasciare il lock.**

Blocchi synchronized

Con riferimento ad un oggetto x si può definire un blocco di statement come una sezione critica nel seguente modo (**synchronized blocks):**

```
synchronized (oggetto x) {<sequenza di statement>;}
```

Esempio:

```
Object mutexLock= new Object;
  …..
  public void  M( ) {
      <sezione di codice non critica>;
      synchronized (mutexlock){
      < sezione di codice critica>;
      }
     <sezione di codice non critica>;
  }
```

- all'oggetto **`mutexLock`** viene implicitamente associato un lock, il cui valore puo` essere:
  - **libero**: il thread può eseguire la sezione critica
  - **occupato**: il thread viene sospeso dalla JVM in una coda associata a **`mutexLock`** *(entry set).*

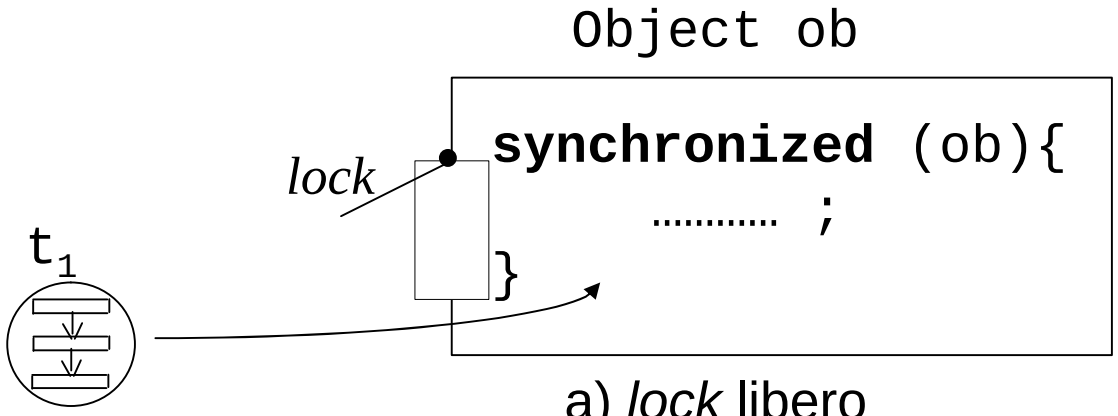
Al termine della sezione critica:

- se *non ci sono thread in attesa*: il lock viene reso libero .
- se *ci sono thread in attesa*: il lock rimane occupato e viene scelto uno di questi .

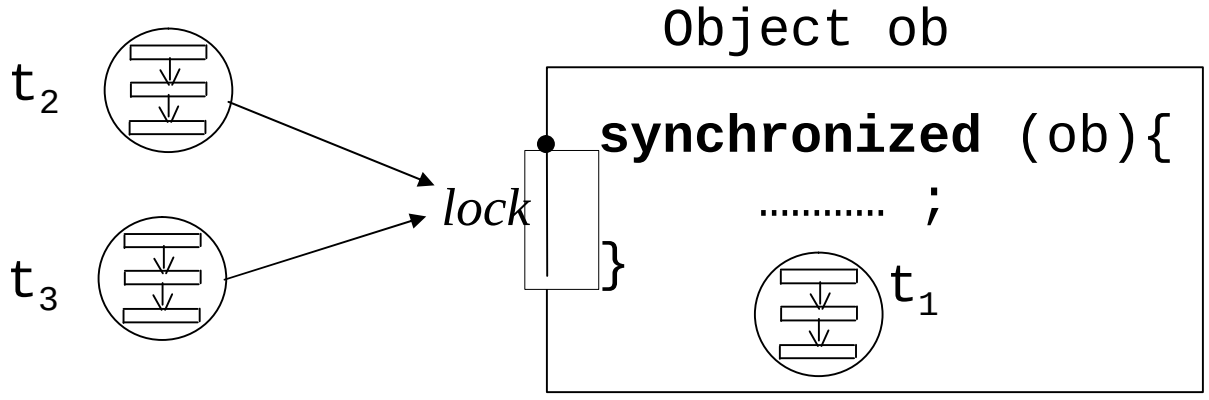
synchronized block

- esecuzione del blocco **mutuamente esclusiva** rispetto:
  - ad altre esecuzioni dello *stesso blocco*
  - all'esecuzione di *altri blocchi* sincronizzati sullo stesso oggetto

Entry set di un oggetto



a) *lock* libero



a) *lock* occupato: t2
e t3 vengono inseriti
nell'entry set di ob

Metodi synchronized

- **Mutua esclusione** tra i metodi di una classe

```
public class intVar {
private int i=0;
public synchronized void incrementa()
{ i ++; }
public synchronized void decrementa()
{i--; }
}
```

- Quando un metodo viene invocato per operare su un oggetto della classe, l'esecuzione del metodo avviene in **mutua esclusione** utilizzando il *lock dell'oggetto.*

Sincronizzazione diretta: `wait` e `notify`

***wait set**:* coda di thread associata ad ogni oggetto, inizialmente vuota.

- I thread entrano ed escono dal ***wait set*** utilizzando i metodi **`wait()`** e **`notify().`**
- **`wait`** e **`notify`** possono essere invocati da un thread **solo all'interno di un *blocco sincronizzato* o di un *metodo sincronizzato*** (possesso del lock dell'oggetto).

wait, notify, notifyall

`wait` comporta il rilascio del lock, la sospensione del thread ed il suo inserimento in **wait set.**

`notify` comporta l'estrazione di un thread da **wait set** ed il suo inserimento in **entry set.**

`notifyall` comporta l'estrazione di **tutti** i thread da **wait set** ed il loro inserimento in **entry set**.

NB: notify e **notifyall** non provocano il rilascio del lock:
➔ i thread risvegliati devono attendere.

➔ Politica **signal&continue**: il rilascio del lock avviene al completamento del *blocco o del metodo sincronizzato* da parte del thread che ha eseguito la notify

```
//Esempio: mailbox con capacita`=1
public class Mailbox{
    private int contenuto;
    private boolean pieno=false;

   public synchronized int preleva()
    {  while (pieno==false)
              wait ( );
       pieno=false;
       notify();
       return contenuto;
   }

   public synchronized void deposita(int valore)
   {   while (pieno==true)
          wait();
       contenuto=valore;
       pieno=true;
       notify();
   }
}
```

```
//Mailbox di capacita` N
public class Mailbox {
  private int[]contenuto;
  private int contatore, testa, coda;

  public mailbox(){
    contenuto = new int[N];
    contatore = 0;
    testa = 0;
    coda = 0;
  }
  public synchronized int preleva (){
    int elemento;
    while (contatore == 0)
      wait();
    elemento = contenuto[testa];
    testa = (testa + 1)%N;
    --contatore;
    notifyAll();
    return elemento;
  }
  public synchronized void deposita (int valore){
    while (contatore == N)
      wait();
    contenuto[coda] = valore;
    coda = (coda + 1)%N;
    ++contatore;
    notifyAll();
  }
}
```

Semafori in Java

- Java non prevede i semafori; tuttavia essi possono essere facilmente costruiti mediante i meccanismi di sincronizzazione standard.

- Le primitive *P e V* (*wait e signal* sui semafori) si possono ottenere dichiarandole come i **syncronized methods** all'interno della classe *semafor*o.

```
public class Semaphore {
  private int value;
  public Semaphore (int initial){
    value = initial;
  }

synchronized public void V()//signal sul semaforo
{    ++value;
    notify();
  }

synchronized public void P() //wait sul semaforo
{       throws InterruptedException
        {    while (value == 0) wait();
             --value;
        }
}
```

wait&notify

Principale limitazione **:**

- **<u>unico wait-set</u>** per un oggetto sincronizzato

➔ non e` possibile sospendere thread su differenti code!

Variabili condizione

- Nelle versioni più recenti di Java (versione 5.0) esiste la possibilità utilizzare le variabili condizione. Ciò è ottenibile tramite l'uso un'apposita interfaccia (definita in **`java.util.concurrent.locks)`** :

```
public interface Condition{

//Public instance methods
void await ()throws InterruptedException;
void signal();
void signalAll();
}
```

- dove i metodi **`await`**, **`signal`**, e **`signalAll`** sono del tutto equivalenti ai metodi wait, notify e notify_all, (ovviamente riferiti alla coda di processi associata alla condition sulla quale vengono invocati)

Mutua esclusione: lock

- Oltre a metodi/blocchi synchronized, la versione 5.0 di java prevede la possibilita` di utilizzare il concetto di *lock*, mediante l'interfaccia (definita in **`java.util.concurrent.locks`**) :

```
public interface Lock{
//Public instance methods
void lock();
void unlock();
Condition newCondition();
}
```

Uso di Variabili Condizione

- Ad ogni variabile condizione deve essere associato un lock, che:
  - al momento della sospensione del thread mediante **`await`** verra` liberato;
  - al risveglio di un thread, verra` automaticamente rioccupato.

➔ La creazione di una condition deve essere effettuata mediante in metodo newCondition del lock associato ad essa.

In pratica, per creare un oggetto Condition :

```
Lock L=new Reentrantlock(); //Reentrantlock è una
                              classe che implementa
                              l'interfaccia Lock

Condition C=lockvar.newCondition();
```

Monitor

Possiamo definire **classi** che rappresentano monitor:

- dati:
  - le variabili condizione
  - 1 lock per la mutua esclusione dei metodi "entry", da associare a turre le variabili condizione
  - variabili interne: stato delle risorse gestite
- metodi:
  - metodi "entry"
  - metodi privati
  - costruttore

Esempio: gestione di buffer circolare

```
public class Mailbox
{ //dati:
private int[] contenuto;
private int contatore,testa,coda;
private Lock lock= new ReentrantLock();
private Condition non_pieno= lock.newCondition();
private Condition non_vuoto= lock.newCondition();

//Costruttore:
public Mailbox( ) {
contenuto=new int[N];
contatore=0;
testa=0;
coda=0;
}
```

```
//metodi "entry":

public int preleva()throws InterruptedException
{  int elemento;
   lock.lock();
   try
  {    while (contatore= = 0)
              non_vuoto.await();
       elemento= contenuto[testa];
       testa=(testa+1) %N;
       --contatore;
       non_pieno.signal ( );
   } finally{ lock.unlock();}
   return element;
}
```

```
public void deposita (int valore)throws InterruptedException
{  lock.lock();
   try
   {   while (contatore==N)
              non_pieno.wait();
       contenuto[coda] = valore;
       coda=(coda+1)%N;
       ++contatore;
       non_vuoto.signal( );
   } finally {  lock.unlock();}
}
}
```

Programma di test:

```
public class Produttore extends Thread
{  int messaggio;
   Mailbox m;
   public  Produttore(Mailbox M){this.m =M;}
   public void run()
   {    while(1)
        { <produci messaggio>
          m.deposita(messaggio);
        }

   }
}

public class Consumatore extends Thread
{  int messaggio;
   Mailbox m;
   public  Consumatore(Mailbox M){this.m =M;}
   public void run()
   {    while(1)
        {       messaggio=m.preleva();
                <consuma messaggio>
        }

   }
```

```
public class BufferTest{

   public static void main(String args[])
   {   Mailbox M=new Mailbox();
       Consumatore C=new Consumatore(M);
       Produttore P=new Produttore(M);
       C.start();
       P.start();
       ...
   }
```