Esercitazione 2:
Java Thread

Thread

Un thread è un ***singolo flusso sequenziale*** di controllo all'interno di un processo

Un thread (o processo leggero) è un'unità di esecuzione che ***condivide codice e dati*** con altri thread ad esso associati

Un ***thread***:

- ***NON*** ha uno spazio di indirizzamento riservato: tutti i thread appartenenti allo stesso processo condividono lo ***stesso spazio di indirizzamento***
- ha ***execution stack e program counter privati***

Java Thread

Due modalità per implementare i thread in Java:

1. ***estendendo la classe `Thread`***
2. ***implementando l'interfaccia `Runnable`***

1) Come sottoclasse di Thread

- la sottoclasse deve ***ridefinire il metodo*** **`run()`**
- si crea ***un'istanza del thread*** tramite **`new`**
- si esegue un thread ***invocando il metodo*** **`start()`** che a sua volta invoca il metodo **`run()`**

Java Thread

```
public class SimpleThread extends Thread{

            public SimpleThread(String str)
            {super(str);}

        public static void main(String[] args)
              {
                SimpleThread st1=new
                          SimpleThread("Thread 1");
                st1.start();
              }
```

Java Thread

```
public void run()
      {
       for(int i=0; i<10; i++)
            {
            System.out.println(i+ " " +getName());
            try{
                sleep((int)Math.random()*1000);
               } catch (InterruptedException e){}
            }
       System.out.println("DONE! "+getName());
      }

}
```


Java Thread

E se occorre definire thread che estendano una classe diversa da Thread?

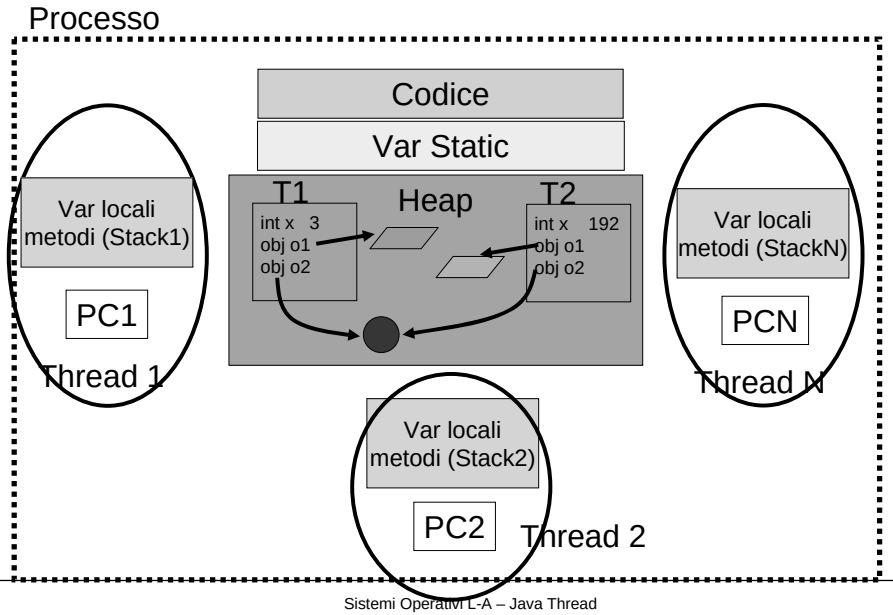
➡ Come classe che implementa interfaccia **Runnable**

1. la sottoclasse deve ***ridefinire il metodo*** ***`run()`***
2. si crea un'istanza di tale sottoclasse tramite **`new`**
3. si crea ***un'istanza della classe Thread*** con **`new`**, passandole come ***parametro l'oggetto che implementa*** ***`Runnable`***
4. si esegue il thread invocando il metodo **`start()`** ***sull'oggetto con classe Thread creato***

Java Thread

```
class EsempioRunnable extends MiaClasse
                implements Runnable {
  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();
  }
}
```


Thread



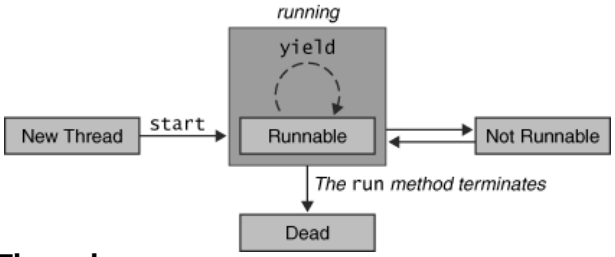
Interferenza

```
public class SharingThread extends Thread
  {
   private int cond=3;
   private SharedObject obj=new SharedObject();

   public SharingThread (String my_name, int x,
                         SharedObject obj){
       super(my_name);
       this.cond=x;
       this.obj=obj;
   }
   public static void main(String[] args){
       int val=128;
       SharedObject obj=new SharedObject();
```


Interferenza

```
   SharingThread st1=
            new SharingThread ("Thread 1", val, obj);
   SharingThread st2=
            new SharingThread ("Thread 2", val, obj);
   st1.start();
   st2.start();
   }

public void run(){
   if(getName().equals("Thread 1")){
                          cond=1; obj.setVal(1);}
   if(getName().equals("Thread 2")){
                          cond=2; obj.setVal(2);}
   }
}
```


Ciclo di vita di un thread



- **New Thread**
  - Subito dopo l'istruzione **new**
  - Il costruttore ***alloca e inizializza le variabili di istanza***
- **Runnable**
  - Il thread è eseguibile ma potrebbe non essere in esecuzione

Ciclo di vita di un thread

- **Not Runnable**
  - Il thread non può essere messo in esecuzione
  - Entra in questo stato quando è ***in attesa della terminazione di un'operazione di I/O***, cerca di accedere ad un ***metodo "synchronized" di un oggetto bloccato***, o dopo aver invocato uno dei seguenti metodi: **`sleep(), wait(), suspend()`**
  - Esce da questo stato quando si verifica la condizione complementare
- **Dead**
  - Il thread giunge a questo stato per "**morte naturale"** o perché un altro thread ha invocato il suo metodo **`stop()`**

Sincronizzazione di thread

Differenti thread ***condividono lo stesso spazio di memoria (heap)***

→ è possibile che più thread accedano **contemporaneamente** ad uno stesso oggetto, ***invocando un metodo che modifica lo stato dell'oggetto***

→ lo stato **finale** dell'oggetto sarà ***funzione dell'ordine*** in base al quale i thread accedono ai dati

- Servono meccanismi di ***sincronizzazione***

Accesso esclusivo

Per evitare che ***thread diversi interferiscano*** durante l'accesso ad oggetti condivisi si possono ***imporre accessi esclusivi***

- JVM supporta la definizione di ***lock sui singoli oggetti*** tramite la keyword **`synchronized`**
- Synchronized può essere definita:
  - ***su metodo***
  - su singolo blocco di codice

Synchronized

In pratica:

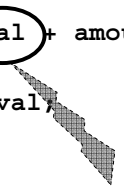
- Ad ogni ***oggetto Java*** è automaticamente ***associato un lock***
- Quando un thread vuole accedere ad un metodo/blocco synchronized, si deve ***acquisire il lock dell'oggetto*** (impedendo così l'accesso ad ogni altro thread)
- Il ***lock viene automaticamente rilasciato*** quando il ***thread esce dal metodo/blocco synchronized*** (o se viene interrotto da un'eccezione)
- Un thread che non riesce ad acquisire un lock ***rimane sospeso sulla richiesta della risorsa*** fino a che il lock non è disponibile

Synchronized

- ***Ad ogni oggetto viene assegnato un solo lock***: due thread non possono accedere contemporaneamente a due metodi/blocchi synchronized diversi di uno stesso oggetto
- Tuttavia altri thread sono ***liberi di accedere a metodi/blocchi non synchronized*** associati allo stesso oggetto

Esempio uso di `synchronized`

```
public class MyThread extends Thread{
  private SharedObject obj;
  ...
  public void run(){ obj.incr(100); }
}
public class SharedObject {
  int val=0;
  public int synchronized incr (int amount) {
      val = val + amount;
      return val;
  }
}
```



Sincronizzazione

Esistono situazioni in cui ***non è sufficiente impedire accessi concorrenti***

Supponiamo che un metodo **`synchronized`** sia l'unico modo per variare lo stato di un oggetto

➡ Che cosa accade se il ***thread che ha acquisito il lock si blocca all'interno del metodo*** stesso in attesa di un cambiamento dello stato?

`wait()`

`wait()` - il thread che la invoca

- si ***blocca*** in attesa che un altro thread invochi **`notify()`** o **`notifyAll()`** per quell'oggetto
- deve essere ***in possesso del lock sull'oggetto***
- ***al momento della invocazione rilascia il lock***

`notifyAll()`

- **`notify()`** - il thread che la invoca
  - risveglia uno dei thread in attesa, scelto **arbitrariamente**
- **`notifyAll()`** - il thread che la invoca
  - risveglia tutti i thread in attesa: essi competeranno per l'accesso all'oggetto
- **`notifyAll()` è preferibile** (e può essere necessaria) se più thread possono essere in attesa

Alcune regole empiriche

1. Se due o più thread possono modificare lo stato di un oggetto, è necessario ***dichiarare `synchronized` metodi di accesso***
2. Se deve attendere la ***variazione dello stato di un oggetto***, un thread deve invocare il metodo **`wait()`**
3. Ogni volta che un metodo attua una ***variazione dello stato di un oggetto***, esso deve invocare **`notifyAll()`**
4. È necessario verificare che ad ogni chiamata a **`wait()` corrisponda** una chiamata a **`notifyAll()`**

Esempio produttori-consumatori

```
public synchronized int get() {
   while (available == false)
      try { wait(); // attende un dato dai Produttori
          } catch (InterruptedException e) {}
   }
. . .
   available = false;
   notifyAll(); // notifica i produttori del consumo
}
```


Esempio produttori-consumatori

```
public synchronized void put(int value) {
   while (available == true)
      try { wait(); // attende il consumo del dato
          } catch (InterruptedException e) {}
   }
. . .
   available = true;
   notifyAll(); // notifica i consumatori della
                // produzione di un nuovo dato
}
```


I problemi di `stop()` e `suspend()`

`stop()`

- forza la ***terminazione*** di un thread
- tutte le ***risorse utilizzate vengono immediatamente liberate*** (lock compresi)

Se il ***thread interrotto*** stava compiendo un insieme di operazioni da eseguirsi in maniera ***atomica***, l'interruzione può condurre ad uno ***stato inconsistente del sistema***

I problemi di `stop()` e `suspend()`

`suspend()`

- ***blocca l'esecuzione di un thread***, in attesa di una successiva invocazione di **`resume()`**
- non libera le risorse impegnate dal thread (***non rilascia i lock***)

Se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva*** (ad es., interrotto durante l'esecuzione di un metodo synchronized), tale ***risorsa rimane bloccata***

Priorità dei thread

La classe **`Thread`** fornisce i metodi:

- **`setPriority(int num)`**
- **`getPriority()`**

dove num ∈ [MIN_PRIORITY,MAX_PRIORITY]

In generale un thread rimane in esecuzione fino a quando:

- ***smette di essere runnable*** (e diviene Not runnable o Dead)
- un thread con ***priorità superiore diviene eseguibile***
- si ***esaurisce il quanto di tempo*** assegnato
- cede il controllo ***chiamando il metodo `yield()`***

Priorità dei thread

Le specifiche JVM ***suggeriscono*** che

- vengano messi in esecuzione per primi i thread a ***priorità più elevata*** tra quelli in ***stato runnable***
- si proceda secondo una ***politica round-robin*** tra i thread con **identica priorità**

- Alcune implementazioni di ***JVM delegano lo scheduling dei thread al SO***, ove supportati
- Il comportamento reale diviene dunque ***dipendente dalla coppia JVM e SO***

Gruppi di thread

Obiettivo: raccogliere una ***molteplicità di thread*** all'interno di ***un solo gruppo*** per facilitare le operazioni di ***gestione***

JVM associa ogni thread ad un ***gruppo all'atto della creazione del thread*** (tale associazione è ***permanente*** e non può essere modificata)

```
ThreadGroup myThreadGroup = new ThreadGroup("MyGroup");
Thread myThread = new Thread(myThreadGroup,"My1");
```

Default: ***stesso gruppo del creatore*** (all'inizio JVM crea il gruppo "**main**")

Altri metodi di interesse per Java thread

- **`sleep(long ms)`**
  - sospende thread per il # di ms specificato
- **`interrupt()`**
  - invia un evento che produce l'interruzione di un thread
- **`interrupted()/isInterrupted()`**
  - verificano se il thread corrente è stato interrotto
- **`join()`**
  - ***attende la terminazione del thread specificato***
- **`isAlive()`**
  - true se thread è stato avviato e non è ancora terminato
- **`yield()`**
  - ***costringe il thread a cedere il controllo della CPU***