Threads in Java

Concetto di processo

- Nei primi sistemi di calcolo era consentita l'esecuzione di un solo programma alla volta. Tale programma aveva il completo controllo del sistema e accesso a tutte le sue risorse.

- Gli attuali sistemi di calcolo consentono, invece, che più programmi siano caricati in memoria ed eseguiti concorrentemente. Essi sono detti ***sistemi time-sharing*** o ***multitasking***. In tali sistemi più programmi vengono eseguiti dalla CPU, che commuta la loro esecuzione con una frequenza tale da permettere agli utenti di interagire con ciascun programma durante la sua esecuzione.

- L'unità di lavoro dei moderni sistemi time-sharing è il ***processo***.

Definizione di processo

- ***Un processo può essere definito come un programma in esecuzione***. Un processo esegue le proprie istruzioni in modo sequenziale, ovvero in qualsiasi momento viene eseguita al massimo un'istruzione del processo.

- ***Un programma di per sé non è un processo***: un programma è un'entità *passiva*, come il contenuto di un file memorizzato su disco, mentre un processo è una entità attiva, con un ***program counter*** che specifica l'attività attuale (ovvero, quale sia la prossima l'istruzione da eseguire) ed un insieme di risorse associate.

- Due processi possono essere associati allo stesso programma, ma sono da considerare due sequenze di esecuzione distinte.

Processi indipendenti e cooperanti

I processi in esecuzione nel sistema operativo possono essere indipendenti o cooperanti:

- Un processo è ***indipendente*** se non può influire su altri processi nel sistema o subirne l'influsso. Un processo che non condivida dati temporanei o permanenti con altri processi è indipendente.

- Un processo è ***cooperante*** se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Ovviamente, qualsiasi processo che condivida dati con altri processi è un processo cooperante.

Cooperazione tra processi

Ci sono diversi motivi per fornire un ambiente che consenta la cooperazione tra processi:

- **Condivisione di informazioni**. Consente a più utenti di condividere le stesse informazioni (ad esempio un file).

- **Accelerazione del calcolo**. Si divide un problema in sottoproblemi che possono essere eseguiti in parallelo. Un'accelerazione di questo tipo è ottenibile solo se il computer dispone di più elementi di elaborazione (come più CPU o canali di I/O)

- **Modularità**. Consente la realizzazione di un sistema modulare che divide le funzioni in processi distinti.

Thread

- Un ***thread***, anche detto ***lightweight process***, è un flusso sequenziale di esecuzione di istruzioni all'interno di un programma/processo.

- In un programma si possono far partire più thread che sono eseguiti concorrentemente. Nei computer a singola CPU la concorrenza viene simulata con una ***politica di scheduling*** che alterna l'esecuzione dei singoli thread.

- Tutti i thread eseguono all'interno del contesto di esecuzione di un solo processo, ovvero condividono le stesse variabili del programma.

- Ciascun ***processo*** tradizionale, o ***heavyweight***, ha invece il proprio contesto di esecuzione.

Thread in Java

- Tutti i programmi Java comprendono almeno un thread. Anche un programma costituito solo dal metodo main viene eseguito come un singolo thread. Inoltre, Java fornisce strumenti che consentono di creare e manipolare thread aggiuntivi nel programma

Esistono due modi per implementare thread in Java:

- Definire una sottoclasse della classe **Thread**.

- Definire una classe che implementa l'interfaccia **Runnable**. Questa modalità è più flessibile, in quanto consente di definire un thread che è sottoclasse di una classe diversa dalla classe Thread.

Definire una sottoclasse della classe Thread

1. Si definisce una nuova classe che estende la classe Thread. La nuova classe deve ridefinire il metodo ***run()*** della classe Thread.

2. Si crea un'istanza della sottoclasse tramite ***new***.

3. Si chiama il metodo ***start()*** sull'istanza creata. Questo determina l'invocazione del metodo run() dell'oggetto, e manda in esecuzione il thread associato.

Esempio di sottoclasse di Thread

```
class Saluti extends Thread {
  public Saluti(String nome) {
    super(nome);
  }
  public void run() {
    for (int i = 0; i < 10; i++)
      System.out.println("Ciao da "+getName());
  }
}
public class ThreadTest {
  public static void main(String args[]) {
    Saluti t = new Saluti("Primo Thread");
    t.start();
  }
}
```

Definire una classe che implementa Runnable

1. Si definisce una nuova classe che implementa l'interfaccia Runnable. La nuova classe deve implementare il metodo ***run()*** dell'interfaccia Runnable.

2. Si crea un'istanza della classe tramite ***new***.

3. Si crea un'istanza della classe Thread, passando al suo costruttore un riferimento all'istanza della nuova classe definita.

4. Si chiama il metodo ***start()*** sull'istanza della classe Thread creata, determinando l'invocazione del metodo run() dell'oggetto Runnable associato.

Esempio di classe che implementa Runnable

```
class Saluti implements Runnable {
  private String nome;
  public Saluti(String nome) {
    this.nome = nome;
  }
  public void run() {
    for (int i = 0; i < 10; i++)
      System.out.println("Ciao da "+nome);
  }
}
public class RunnableTest {
  public static void main(String args[]) {
    Saluti s = new Saluti("Secondo Thread");
    Thread t = new Thread (s);
    t.start();
  }
}
```

La classe Thread (1)

Costruttori principali:

Thread (): crea un nuovo oggetto Thread.

Thread (String name): crea un nuovo oggetto Thread con nome **name**.

Thread (Runnable target): crea un nuovo oggetto Thread a partire dall'oggetto **target**.

Thread (Runnable target, String name): crea un nuovo oggetto Thread con nome **name** a partire dall'oggetto **target**.

Metodi principali:

String getName(): restituisce il nome di questo Thread.

void join() *throws InterruptedException*: attende fino a quando questo Thread non termina l'esecuzione del proprio metodo run.

La classe Thread (2)

void join(long millis) *throws InterruptedException*: attende, per un tempo massimo di **millis** millisecondi, fino a quando questo Thread non termina l'esecuzione del proprio metodo run.

void run(): specifica le operazioni svolte dal Thread. Deve essere ridefinito dalla sottoclasse, altrimenti non effettua alcuna operazione. Se il Thread è stato costruito a partire da un oggetto Runnable, allora verrà invocato il metodo run di tale oggetto.

static void sleep(long millis) *throws InterruptedException*: determina l'interruzione dell'esecuzione del Thread corrente per un tempo di **millis** millisecondi.

void start(): fa partire l'esecuzione del Thread. Viene invocato il metodo run di questo Thread.

static void yield(): determina l'interruzione temporanea del Thread corrente, e consente ad altri Thread di essere eseguiti.

La classe Thread (3)

static Thread currentThread(): restituisce un riferimento all'oggetto Thread attualmente in esecuzione.

void setPriority(int newPriority): cambia la priorità di questo Thread.

int getPriority(): restituisce la priorità di questo Thread.

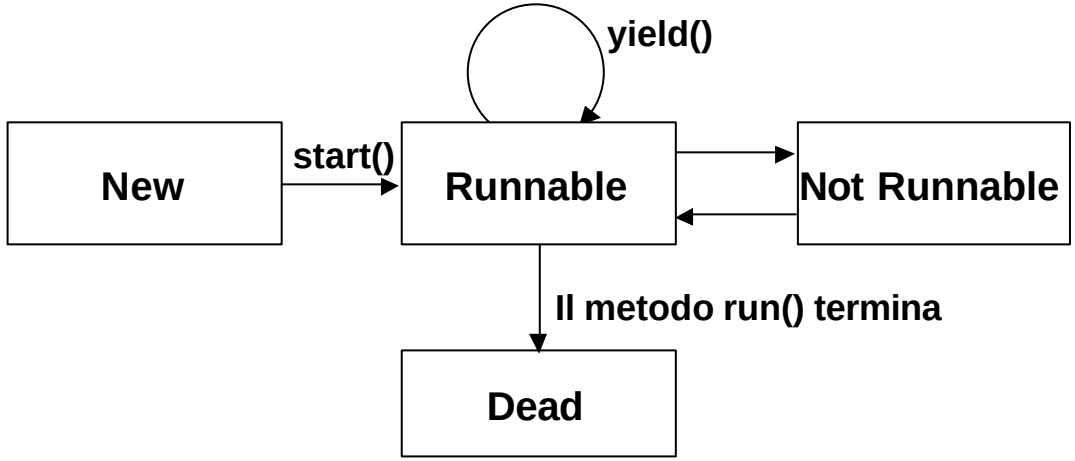
Costanti della classe:

static final int MAX_PRIORITY: la massima priorità (pari a 10) che un Thread può avere.

static final int MIN_PRIORITY: la minima priorità (pari ad 1) che un Thread può avere.

static final int NORM_PRIORITY: la priorità (pari a 5) che viene assegnata di default ad un Thread.

Ciclo di vita di un Thread



- **New**: subito dopo l'istruzione new le variabili sono state allocate e inizializzate; il thread è in attesa di passare nello stato Runnable.
- **Runnable**: il thread è in esecuzione o in coda di attesa per ottenere l'utilizzo della CPU.
- **Not Runnable**: il thread non può essere messo in esecuzione dallo scheduler. Entra in questo stato quando è in attesa di operazioni di I/O, oppure dopo l'invocazione del metodo sleep(), o del metodo wait(), che verrà discusso in seguito.
- **Dead**: al termine dell'esecuzione del suo metodo run().

Scheduling dei Thread

La Java Virtual Machine (JVM) è in grado di eseguire una molteplicità di thread su una singola CPU:

- lo scheduler della JVM sceglie il thread in stato Runnable con priorità più alta;
- se più thread in attesa di eseguire hanno uguale priorità, la scelta dello scheduler avviene con una modalità ciclica (**round-robin**).

Il thread messo in esecuzione dallo scheduler viene interrotto se e solo se:

- il metodo run termina l'esecuzione;
- il thread esegue yield();
- un thread con priorità più alta diventa Runnable;
- il quanto di tempo assegnato si è esaurito (solo su sistemi che supportano time-slicing).

Un programma sequenziale (1)

```
class Printer {
  private int from;
  private int to;

  public Printer (int from, int to) {
    this.from = from;
    this.to = to;
  }

  public void print () {
    for (int i = from; i <= to; i++)
      System.out.print (i+"\t");
  }
}

public class PrinterApp {
  public static void main (String args[]) {
    Printer p1 = new Printer (1,10);
    Printer p2 = new Printer (11,20);
    p1.print();
    p2.print();
    System.out.println ("Fine");
  }
}
```

Un programma sequenziale (2)

L'esecuzione di **PrinterApp** genera sempre il seguente output:

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
Fine

I metodi:

p1.print()

p2.print()

System.out.println("Fine")

sono eseguiti in modo sequenziale.

Un programma threaded (1)

```
class TPrinter extends Thread {
  private int from;
  private int to;

  public TPrinter (int from, int to) {
    this.from = from;
    this.to = to;
  }

  public void run () {
    for (int i = from; i <= to; i++)
      System.out.print (i+"\t");
  }
}

public class TPrinterApp {
  public static void main (String args[]) {
    TPrinter p1 = new TPrinter (1,10);
    TPrinter p2 = new TPrinter (11,20);
    p1.start();
    p2.start();
    System.out.println ("Fine");
  }
}
```

Un programma threaded (2)

L'esecuzione di **TPrinterApp** potrebbe generare il seguente output:

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
Fine

oppure:

Fine
1 2 3 **11** 4 **12** 5 **13** 6 **14**
7 **15** 8 **16** 9 **17** 10 **18** **19** **20**

oppure:

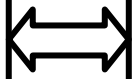
1 2 **11** **12** 3 4 ***Fine***
5 **13** **14** 6 7 **15** 8 **16** **17** 9
18 **19** **20** 10

e così via: non è possibile fare alcuna assunzione sulla velocità relativa di esecuzione dei thread. L'unica certezza è che le operazioni all'interno di un dato thread procedono in modo sequenziale.

Imporre la sequenzialità

Per imporre la sequenzialità nell'esecuzione dei diversi thread si può fare uso del metodo join().

```
public class TPrinterApp {
  public static void main (String args[]) {
    TPrinter p1 = new TPrinter (1,10);
    TPrinter p2 = new TPrinter (11,20);
    p1.start();
    try {
      p1.join();
    } catch (InterruptedException e) {
      System.out.println(e);
    }
    p2.start();
    try {
      p2.join();
    } catch (InterruptedException e) {
      System.out.println(e);
    }
    System.out.println ("Fine");
  }
}
```



```
public class PrinterApp {
  public static void main (String args[]) {
    Printer p1 = new Printer (1,10);
    Printer p2 = new Printer (11,20);
    p1.print();
    p2.print();
    System.out.println ("Fine");
  }
}
```

Somma in concorrenza (1)

```
class Sommatore extends Thread {
  private int da;
  private int a;
  private int somma;
  public Sommatore (int da, int a) {
    this.da = da;
    this.a = a;
  }
  public int getSomma() {
    return somma;
  }
  public void run () {
    somma = 0;
    for (int i = da; i <= a; i++)
      somma += i;
  }
}
```

Somma in concorrenza (2)

```
public class Sommatoria {
  public static void main (String args[]) {
    int primo = 1;
    int ultimo = 100;
    int intermedio = (primo+ultimo)/2;
    Sommatore s1 = new Sommatore (primo,intermedio);
    Sommatore s2 = new Sommatore (intermedio+1,ultimo);
    s1.start();
    s2.start();
    try {
      s1.join();
      s2.join();
    } catch (InterruptedException e) { System.out.println (e); }
    System.out.println (s1.getSomma()+s2.getSomma());
  }
}
```

Thread con attività ciclica (1)

Ciclo finito:

```
public void run ()
{
  for (int i = 0; i < n; i++)
  {
    istruzioni
  }
}
```

Ciclo infinito:

```
public void run ()
{
  while (true)
  {
    istruzioni
  }
}
```

Thread con attività ciclica (2)

Ciclo con terminazione:

```
class  myThread extends Thread {
  private boolean continua;
  public myThread () {
    continua = true;
  }
  public vodi termina ()  {
    continua = false;
  }
  public void run () {
    while (continua)
    {
       istruzioni
    }
  }
}
```

Una classe Clock (1)

```
class Clock extends Thread {
  private boolean continua;
  public Clock () {
    continua = true;
  }
  public void block () {
    continua = false;
  }
  public void run () {
    int i = 1;
    while (continua) {
      try {
        sleep (1000);
      } catch (InterruptedException e){System.out.println (e);}
      if (continua)
        System.out.print ("\n"+i);
      i++;
    }
  }
}
```

Una classe Clock (2)

```
class ClockController extends Thread {
  private Clock clock;
  public ClockController (Clock clock) {
    this.clock = clock;
  }
  public void run () {
    Console.readString ("Press enter to start");
    clock.start();
    Console.readString ("Press enter to stop");
    clock.block();
  }
}

public class ClockTest {
  public static void main (String args[]) {
    Clock t = new Clock();
    new ClockController(t).start();
  }
}
```