Corso sul linguaggio Java

Modulo JAVA9

B2 – Multithreading

Prerequisiti

- Programmazione base in Java
- Significato di *multithreading*
- Concetti di base sui sistemi operativi
- Attesa attiva
- Attesa passiva

Introduzione

In questa Unità approfondiamo il concetto di programmazione *multithreading*, attraverso esempi di applicazioni Java.

In particolare, affrontiamo la differenza pratica tra attesa attiva e attesa passiva, l'uso dell'interfaccia **Runnable** (che consente il *multithreading* in ambiente grafico) ed un primo esempio di sincronizzazione, mediante il metodo **join**().

Esempio con due *thread*

```
public class MultiThread extends Thread
{   public void run()
    {       System.out.println("Sono il thread " + this);    }
    public MultiThread()
    {       // azioni per il costruttore
    }
    public static void main (String args[])
    {       Thread t0 = new MultiThread();
            Thread t1 = new MultiThread();
            t0.setName("cip");
            t1.setName("ciop");
            t0.start();
            t1.start();
    }
}
```

Si creano due *thread* indipendenti

Si assegna a ciascun *thread* un nome

Le stampe di ciascun *thread* avvengono in modo casuale, in base alla politica dello schedulatore.

I *thread* vengono avviati

Esempio con due *thread*

I *thread* t0 e t1 potrebbero svolgere compiti diversi; frequente è il caso in cui vengono eseguite in modo concorrente operazioni multiple e indipendenti, con ovvi vantaggi rispetto ai tempi di esecuzione.

I *thread* t0 e t1 vengono eseguiti in parallelo; evolvono in modo realmente parallelo su architetture HW *multicore*, multiprocessore o in ambiente operativo Window o Linux.

I *thread* t0 e t1 possono:

- interagire e comunicare (argomento che vedremo in seguito)
- proseguire in modo totalmente indipendente.

Busy wait e **sleep**()

Il ritardo casuale inserito nel metodo **run**() in molti dei precedenti esempi, produce un avanzamento casuale dei *thread* in gioco e può essere realizzato in due modi:

Pone il *thread* in stato **Not Runnable**

- **public static void sleep** (**long** msToSleep) **throws InterruptedException** (come negli esempi precedenti)
- **long** startTime = **System.currentTimeMillis**();
  long stopTime = startTime + 60000;
  while (**System.currentTimeMillis**() < stopTime)
  { // corpo del ciclo }

Sebbene i due metodi producano lo stesso effetto, il primo non coinvolge il processore (**attesa passiva**) e può essere interrotto, mentre il secondo ne prevede l'uso (**attesa attiva** o **busy wait**) dovendo svolgere dei cicli.

Busy wait e **sleep**()

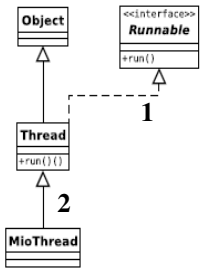
```
public class ThreadSleep extends Thread
{   public void run()
    { for (int i = 0; i < 20; ++i)
          {System.out.println("Nuovo thread");
           try { Thread.sleep (200); }
           catch (InterruptedException e)  { return; }
          }
    }
    public static void main (String[] args) throws InterruptedException
    {     new ThreadSleep().start();
          for (int i = 0; i < 2000; ++i)
            System.out.println("Main thread");
    }
}
```

Il *thread* va in attesa passiva (**Not Runnable**) e può essere interrotto.

Il *thread* va in attesa attiva ed esegue di continuo.

Interfaccia **Runnable**

Poiché le classi viste in precedenza ereditano da **Thread** e poiché Java non supporta l'ereditarietà multipla, non possiamo costruire classi che ereditano ulteriormente da altre classi (es. **Canvas**)



Come si può risolvere il problema?

1. Creare una classe che implementa l'interfaccia **java.lang.Runnable** e implementare in essa il metodo **run**()
2. creare un'istanza di **Thread** passandogli un'istanza della classe **Runnable** e richiamare su essa il metodo **start**().

Uso dell'interfaccia **Runnable**

Vediamo l'uso di **Runnable** in un'applicazione che lancia 2 *thread* di classe *EsempioThread* che procedono in modo indipendente.

```
public class EsempioRunnable
{   public static void main( String args[] )
    {       Runnable t1 = new EsempioThread("thread1"),
                      t2 = new EsempioThread("thread2");
            Thread  thread1 = new Thread (t1),
                    thread2 = new Thread (t2),
            System.out.println( "Ora avvio i thread …" );
            thread1.start();
            thread2.start();
    }
}
```

Crea due oggetti **Runnable**

Crea due oggetti **Thread** ereditati da **Runnable**

Interfaccia Runnable

```
class EsempioThread implements Runnable
{   String name;
    public EsempioThread( String name )
   { this.name = name;  }
   public void run()
  {         for (int i = 0; i < 5; ++i)
    {       System.out.println (name + " : in esecuzione.");
            try { Thread.sleep (sleepTime); }
            catch (InterruptedException e) { }
    }       System.err.println( name + " finito" );
  }
}
```

È sempre preferibile usare l'interfaccia **Runnable** anche quando non si vuole ereditare da altre classi, come mostra questo esempio.

Thread e grafica

Grazie all'interfaccia **Runnable**, i *thread* e la grafica possono essere utilizzati insieme per realizzare applicazioni grafiche concorrenti, alla base di molti programmi di videogioco, simulazione e realtà virtuale.

Vediamo un esempio: due palline di colore diverso, si muovono a velocità diverse, su due percorsi distinti.

Ogni pallina opera in un frame distinto e il movimento di ognuna di esse è controllato da un *thread* apposito.

Applicazione DuePalline
- il main()

```
import java.awt.*;
public class DuePalline
{   public static void main (String args[])
    {       Frame f1=new Frame("Pallino blu");
            caratteristiche del frame;
            creazione oggetto mobile t1 derivato da Runnable;;
            f1.add(t1);
            Thread muovi1= new Thread (t1);   // creazione thread per t1
            f1.setVisible(true);
            muovi1.start();                   // avvio thread t1
            analoghe operazioni per il frame f2;
    }  // end main
}  // end class
```

Applicazione DuePalline
- la classe oggettoMobile

```
class oggettoMobile extends Canvas implements Runnable
{   int   pos_iniz,          // posizione iniziale
          pos,               // posizione corrente
          raggio;            // dimensione pallina
          double velocita;   // velocità della pallina
          Color colore;      // colore pallina
   public oggettoMobile(int x, int r, double v, Color c)
   {      pos_iniz=x;        // costruttore pallina
          raggio=r;
          velocita=v;
          colore=c;
    }
   public void run() {…}                    // metodo da ridefinire
   public void paint (Graphics g) {…}       // disegna la pallina
} // end class oggetto mobile
```


Applicazione DuePalline
- il metodo run()

```
public void run()
{   while (true)                                    // ciclo continuo
    {
        for (pos=pos_iniz;pos<=300;pos+=1)          // avanza a destra
        {   try
            {   repaint();                          // effetto animazione
                Thread.sleep((int)(1/velocita));    // sleep(msec)
            }
            catch (Exception e)
            {   System.out.println("Errore");   }
        }
    }
}
```

Applicazione DuePalline

- il metodo paint()

```
public void paint (Graphics g)          // disegna la pallina con
{                                       //...gli attributi impostati
        g.setColor(colore);
        g.fillOval(pos, 10, raggio/2, raggio/2);
}
```


Esempio di sincronizzazione

Consideriamo un'applicazione *SommaProdotto* che lancia due thread:

- il primo fa la somma *s* di 2 numeri;
- il secondo fa il prodotto *p* di 2 numeri.

La procedura *SommaProdotto* evidentemente esegue in parallelo il calcolo di *s* e di *p* e deve stamparne la somma.

Cosa è necessario che faccia la procedura *Espressione* per funzionare correttamente?

Il processo padre deve attendere che entrambi i thread *s* e *p* terminino e gli comunichino i propri risultati, realizzando quindi una prima forma di **sincronizzazione**.

Esempio di sincronizzazione

```
public class SommaProdottoJoin
{   static int totale, somma, prodotto;
    public void setSomma(int s)     {   somma = s;   }
    public void setProdotto(int p)   {   prodotto=p; }
    public static void main (String[] args) throws Exception
    {  sommaThread st = new sommaThread("Sommatore", 4, 5);
       prodottoThread pt = new prodottoThread("Moltiplicatore", 3, 7);
       st.start();              // avvia thread
       pt.start();              // avvia thread
       st.join();
       pt.join();
       System.out.println("Totale = " + (somma+prodotto));
    }
}
```

Il metodo **join**() sincronizza il processo padre con i due *thread*.

L'istanza del metodo **join**() mette il *thread* corrente in attesa della terminazione del *thread* su cui si istanzia.

Esempio di sincronizzazione

La classe *sommaThread* si occupa del calcolo della somma.

```
class sommaThread extends Thread
{   int x, y, somma;
    String nome;
    public sommaThread (String n, int x, int y)
    {  nome=n;    this.x=x;    this.y=y;     }
    public void run()
    {  try { Thread.sleep(1000);    // attesa
             somma = x + y;
             System.out.println("Esegue " + nome + "\t somma = " + somma);
             SommaProdottoJoin sp = new SommaProdottoJoin();
             sp.setSomma(somma);
          }
       catch (Exception e) {System.out.println("Errore...");}
    }
}
```

Riporta il risultato nel *thread* padre

Esempio di sincronizzazione

```
class prodottoThread extends Thread
{   int x, y, prodotto;
    String nome;
    public prodottoThread (String n, int x, int y)
    {  nome=n;    this.x=x;   this.y=y;     }
    public void run()
    {  try
      { Thread.sleep(1000);           // attesa
        prodotto = x * y;;
        System.out.println("Esegue " + nome + "\t prodotto = " + prodotto);
        SommaProdottoJoin sp = new SommaProdottoJoin();
        sp.setProdotto(prodotto);
          }
      catch (Exception e) {System.out.println("Errore...");}
    }
}
```

La classe *prodottoThread* si occupa del calcolo del prodotto.

Riporta il risultato nel *thread* padre

Esempio di sincronizzazione

Il vantaggio in termini di tempo di esecuzione delle applicazioni parallele si può facilmente verificare. Ad esempio:

1. cronometrare l'applicazione *SommaProdottoJoin* precedente, che opera in parallelo
2. cronometrare un'analoga applicazione che faccia la stessa cosa, in modo sequenziale.
3. confrontare i tempi ottenuti.

Argomenti

- Esempio con due *thread*
- Esempio di *multithreading*
- *Busy wait* e **sleep**()
- Interfaccia **Runnable**
- Thread e grafica
- Applicazione DuePalline
  - il main()
  - la classe oggettoMobile
  - il metodo run()
  - il metodo paint()
- Esempio di sincronizzazione

Altre fonti di informazione

- P.Gallo, F.Salerno – Informatica Generale 1, ed. Minerva Italica
- M.Romagnoli, P.Ventura – Linguaggio C/C++, ed. Petrini
- M. Bigatti – Il linguaggio Java, ed. Hoepli