



Lezione n.1

LPR-A-09

JAVA: Tasks e Threads

22/9/2009

Vincenzo Gervasi

(basato su materiale di Laura Ricci e Marco Danelutto)

Logistica del corso

- Lezione di teoria: martedì alle 14:00, aula E
- Laboratorio: lunedì, due turni: alle 14:00 e alle 16:00, aula H-Lab
- Non c'è obbligo di frequenza
- Frequenza ai due turni LIBERA, compatibilmente con la disponibilità di posti nei laboratori
- Esercitazioni individuali o a coppie se necessario
- Assistente: Giacomo Baldi
- Modalità d'esame: progetto individuale + orale
- Orario di ricevimento:<
 - Ricevimento settimanale: da fissare
 - Ricevimento su appuntamento via email (in caso sia impossibile usufruire dell'orario "ufficiale")

PROGRAMMA DEL CORSO

Threads: Attivazione, Classe Thread, Interfaccia Runnable, Thread Pooling, Threads, Sincronizzazione su strutture dati condivise: metodi synchronized, wait, notify, notifyall

Thread Pooling: Meccanismi di gestione di pools di threads

Streams: Proprietà degli Streams, Tipi di streams, Composizione di streams, ByteArrayInputStream, ByteArrayOutputStream

Indirizzamento IP: Gestione degli Indirizzi IP in JAVA: La classe InetAddress

Meccanismi di Comunicazione in Rete: Sockets Connectionless e Connection Oriented

Connectionless Sockets: La classe Datagram Socket: creazione di sockets, generazione di pacchetti, timeouts, uso degli streams per la generazione di pacchetti di bytes, invio di oggetti su sockets connectionless.

PROGRAMMA DEL CORSO

Multicast: La classe MulticastSocket, Indirizzi di Multicast, Associazione ad un gruppo di multicast. Proposte di reliable multicast (FIFO multicast, causal multicast, atomic multicast).

Connection Oriented Sockets: Le classi ServerSocket e Socket. Invio di oggetti su sockets TCP.

Il Paradigma Client/Server: Caratteristiche del paradigma client/server, Meccanismi per l'individuazione di un servizio, architettura di un servizio di rete.

Oggetti Distribuiti: Remote Method Invocation, Definizione di Oggetti Remoti, Registrazione di Oggetti, Generazione di Stub e Skeletons.

Meccanismi RMI Avanzati: Il meccanismo delle callback.

Materiale didattico

- Lucidi
- Harold, **Java Network Programming**, terza edizione (come riferimento per la programmazione)
- Testi del corso di RC (come riferimento per la teoria)
- Per le basi di Java:
 - Bruni, Corradini, Gervasi, **Programmazione in Java**
 - Horstmann, **Concetti di Informatica e Fondamenti di Java 2**
 - Qualunque altro testo abbiate usato per i corsi di LIP
- Sito web su Didawiki (in fase di allestimento)

MULTITHREADING: DEFINIZIONI

Definizioni:

Thread: flusso sequenziale di esecuzione

Multithreading:

- consente di definire più flussi di esecuzione (threads) all' interno dello stesso programma
- i threads possono essere eseguiti
- in parallelo (**simultaneamente**) se il programma viene eseguito su un multiprocessor
- in modo concorrente (**interleaving**) se il programma viene eseguito su un uniprocessor, ad esempio mediante **time-sharing**

MULTITHREADING: MOTIVAZIONI

Migliorare le prestazioni di un programma:

- dividere il programma in diverse parti ed assegnare l'esecuzione di ogni parte ad un processore diverso
- su architetture di tipo uniprocessor:
 - può migliorare l'uso della CPU quando il programma si blocca (es: I/O)
 - implementazione di interfacce utente reattive

Web Server:

- Assegna un thread ad ogni richiesta
- Utilizza più CPU in modo da gestire in parallelo le richieste degli utenti

Interfacce reattive:

- gestione asincrona di eventi generati dall'interazione con l'utente

MULTITHREADING: MOTIVAZIONI

Migliorare la progettazione del codice:

- Non solo una panacea per aumentare le prestazioni, ma uno strumento per sviluppare software **robusto** e **responsivo**

Esempi:

- progettazione di **un browser** : mentre viene caricata una pagina, mostra un'animazione. Inoltre mentre carico la pagina posso premere il bottone di stop ed interrompere il caricamento. Le diverse attività possono essere associati a threads diversi.
- Progettazione di applicazioni complesse che richiedono la gestione contemporanea di più attività.
- applicazioni interattive distribuite (giochi multiplayer): si devono gestire eventi provenienti dall'interazione con l'utente, dalla rete...

TASKS E THREADS IN JAVA

- L'interfaccia `java.lang.Runnable` contiene un solo metodo
`void run()`
- Un `task` è un oggetto (di una classe) che implementa l'interfaccia `Runnable`: la sua esecuzione inizia con l'invocazione di `run()`
- Un `thread` è un oggetto della classe `java.lang.Thread`, che implementa `Runnable`
- `Attivare un thread` significa iniziare un nuovo flusso di esecuzione per eseguire un determinato `task`
- Un thread viene attivato invocando il metodo
`void start()`
- La JVM (Java Virtual Machine) inizia l'esecuzione del thread invocandone il metodo `run()`

THREAD IN UN PROGRAMMA JAVA

- IL thread `main` viene creato dalla JVM per avviare l'esecuzione di un'applicazione JAVA: il task eseguito è il metodo `main(String [])` della classe invocata, ad esempio `MyClass` se abbiamo eseguito da linea di comando

```
java MyClass
```

- Altri thread sono attivati automaticamente dalla JVM (sono thread `daemons`: gestore eventi interfaccia, garbage collector, ...)
- Ogni thread durante la sua esecuzione può `attivare` altri thread
- Un thread è un `daemon` se e solo se il thread che lo ha creato è un `daemon`
- Un programma JAVA termina quando sono terminati tutti i suoi thread che non sono `daemons`.

ATTIVAZIONE DI THREADS

Per definire un **task** e eseguirlo in un nuovo thread si usano due tecniche:

- Si **estende** la classe **Thread** sovrascrivendo il metodo **run()**: **questo è possibile solo se non dobbiamo ereditare da un'altra classe**. Per attivare il thread basta invocare **start()** su una istanza della classe.

oppure

- Si definisce una classe **C** che implementa **Runnable**. Quindi si crea una nuova istanza di **Thread**, passando una istanza **O** di **C** come argomento, e si invoca **start()** sul thread.
- Come funziona? Vediamo il metodo **run()** di **Thread**:

```
public Thread (Runnable target){
    this.target = target}

public void run( ) {
    if (target != null) { target.run( ); } }
```

IL TASK DECOLLO

Esempio: Implementare un task `Decollo` che implementi un 'conto alla rovescia' e che, alla fine del conto, invii un segnale 'Via!'

```
public class Decollo implements Runnable {
    int countdown = 10; // Predefinito
    private static int taskCount = 0;
    final int id = taskCount++; // identifica il task
    public Decollo( ) { }
    public Decollo (int countdown) {
        this.countdown = countdown; }
    public String status ( ) {
        return "#" + id + "(" +
            (countdown > 0 ? countdown: "Via!!!")+"),"; }
}
```

IL TASK DECOLLO

```
public void run( )    {
    while (countDown-- > 0){
        System.out.print(status( ));
        try{ Thread.sleep(100);}
            catch(InterruptedException e){ }
        }}}

public class MainThread {
    public static void main(String[] args){
        decollo d= new Decollo(); d.run( );
        System.out.println ("Aspetto il decollo");}}
```

OutputGenerato

#0(9),#0(8),#0(7),#0(6),#0(5),#0(4),#0(3),#0(2),#0(1),#0(Via!!!),Aspetto il decollo

UN TASK NON E' UN THREAD!

- **NOTA BENE:** Nell'esempio precedente non viene creato alcun thread per l'esecuzione del metodo `run()`
- Il metodo `run()` viene eseguito all'interno del thread `main`, attivato per il programma principale
- Invocando direttamente il metodo `run()` di un oggetto di tipo `Runnable`, non si attiva alcun thread ma si esegue il task definito dal metodo `run()` nel thread associato al flusso di esecuzione del chiamante
- Per associare un nuovo thread di esecuzione ad un Task, occorre creare un oggetto di tipo `Thread` e passargli il Task

DECOLLO IN UN THREAD INDIPENDENTE

```
public class MainThread {  
    public static void main(String [ ] args) {  
        Decollo d = new Decollo();  
        Thread t = new Thread(d);  
        t.start();  
        System.out.println("Aspetto il Decollo"); }  
}
```

Output generato (con alta probabilità, comunque può dipendere dallo schedulatore):

Aspetto il decollo

#0(9),#0(8),#0(7),#0(6),#0(5),#0(4),#0(3),#0(2),#0(1),#0(Via!!!),

DECOLLO IN PIU' THREAD

```
public class MoreThreads {  
    public static void main(String [ ]args) {  
        for (int i=0; i<5; i++)  
            new Thread(new Decollo()).start();  
        System.out.println("Aspetto il decollo");  
    }  
}
```

Possibile output generato:

```
#0(9),#1(9),Aspetto il decollo  
#2(9),#4(9),#3(9),#1(8),#0(8),#3(8),#4(8),#2(8),#1(7),#0(7),#2(7),#4(7),  
#3(7),#0(6),#4(6),#2(6),#3(6),#1(6),#2(5),#1(5),#3(5),#4(5),#0(5),#2(4),  
#3(4),#0(4),#4(4),#1(4),#2(3),#0(3),#3(3),#4(3),#1(3),#3(2),#0(2),#2(2),  
#4(2),#1(2),#3(1),#2(1),#0(1),#1(1),#4(1),#3(Via!!!),#4(Via!!!),#2(Via!!!),  
#0(Via!!!),#1(Via!!!),
```


LA CLASSE `java.lang.Thread`

La classe `java.lang.Thread` contiene membri per:

- costruire un thread interagendo con il sistema operativo ospite
- attivare, sospendere, interrompere i thread
- non contiene i metodi per la sincronizzazione tra i thread, che sono definiti in `java.lang.Object`.

Costruttori:

- Vari: differiscono per parametri utilizzati (esempio: task da eseguire, nome del thread, gruppo cui appartiene il thread: vedere API)

Metodi

- Possono essere utilizzati per interrompere, sospendere un thread, attendere la terminazione di un thread + un insieme di metodi set e get per impostare e reperire le caratteristiche di un thread
 - esempio: assegnare nomi e priorità ai thread

LA CLASSE `java.lang.Thread`: `start()`

Il metodo `start()`

- segnala allo schedulatore della JVM che il thread può essere attivato (invoca un metodo nativo). L'ambiente del thread viene inizializzato
- ritorna immediatamente il controllo al chiamante, senza attendere che il thread attivato inizi la sua esecuzione.
 - **NOTA:** la stampa del messaggio "Aspetto il decollo" è nel mezzo di quelle effettuate dai threads. Questo significa che il controllo è stato restituito al thread chiamante (il thread associato al main) prima che sia terminata l'esecuzione dei threads attivati

LA CLASSE `java.lang.Thread`: `run()`

- La classe `Thread` implementa l'interfaccia `Runnable` e quindi contiene l'implementazione del metodo `run()`

```
public void run() {  
    if (target != null) { target.run(); } }  
}
```

`target` = riferimento all'oggetto `Runnable` passato al momento della creazione oppure null.

- L'attivazione di un thread mediante la `start()` causa l'invocazione del metodo `run()` precedente. A sua volta, viene invocato il metodo `run()` sull'oggetto che implementa `Runnable` (se questo è presente).
- Qualsiasi istruzione eseguita dal thread fa parte di `run()` o di un metodo invocato da `run()`. Inoltre il thread termina con l'ultima istruzione di `run()`.
- Dopo la terminazione un thread non può essere riattivato

ESTENSIONE DELLA CLASSE THREADS

Creazione ed attivazione di threads: un approccio alternativo

- creare una classe *C* che estenda la classe *Thread*
- effettuare un *overriding* del metodo *run()* definito all'interno della classe *Thread*
- Istanziare un oggetto *O* di tipo *C*. *O* è un thread il cui comportamento è programmato nel metodo *run()* riscritto in *C*
- Invocare il metodo *start()* su *O*. Tale metodo attiva il thread ed invoca il metodo riscritto.

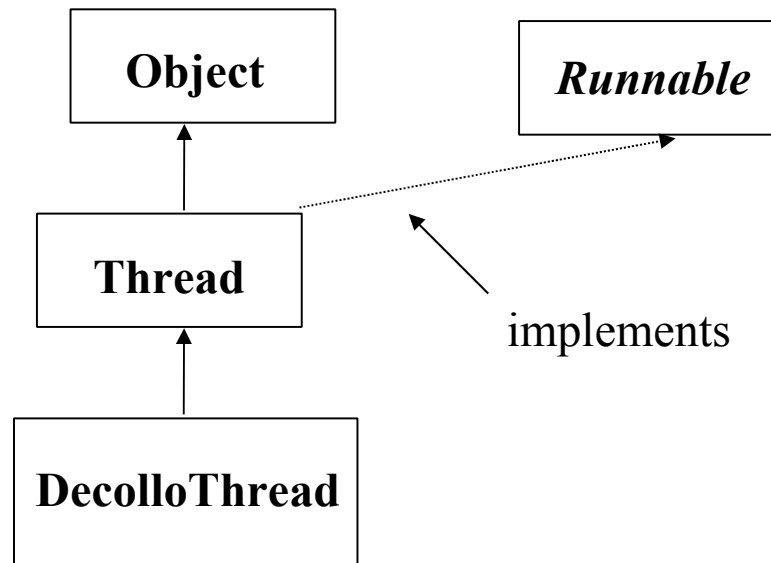
DECOLLO COME SOTTOCLASSE DI THREAD

```
public class DecolloThread extends Thread {.....
    public void run( )    {
        while (countDown-- > 0){
            System.out.print(status( ));
            try{ Thread.sleep(100);
            } catch(InterruptedException e){ } }}}

public class MainDecolloThread {
    public static void main(String [ ]args) {
        DecolloThread d = new DecolloThread( );
        d.start();
        System.out.println("Aspetto il Decollo"); }}
```

GERARCHIA DELLE CLASSI

- `Thread` estende `Object` e implementa l'interfaccia `Runnable`
- `DecolloThread` estende `Thread` e sovrascrive il metodo `run()` di `Thread`



QUANDO E' NECESSARIO USARE LA RUNNABLE

In JAVA una classe può estendere una sola altra classe (*eredità singola*)



La classe i cui oggetti devono essere eseguiti come thread non può estendere altre classi.

Questo può risultare svantaggioso in diverse situazioni.

Esempio: *Gestione degli eventi* (es: movimento mouse, tastiera...) la

- classe che gestisce l'evento deve estendere una classe predefinita JAVA
- inoltre può essere necessario eseguire il gestore come un thread separato

GESTIONE DEI THREADS

- **public static native Thread** `currentThread ()`
 - In un ambiente multithreaded, lo stesso metodo può essere eseguito in modo concorrente da più di un thread. Questo metodo restituisce un riferimento al thread che sta eseguendo un segmento di codice
- **public final void** `setName(String newName)`
- **public final String** `getName()`
 - consentono, rispettivamente, di associare un nome ad un thread e di reperire il nome assegnato
- **public static native void** `sleep (long mills)`
 - sospende l'esecuzione del thread che invoca il metodo per mills millisecondi. Durante questo intervallo di tempo il thread non utilizza la CPU. **Non è possibile porre un altro thread in sleep.**

ESERCIZIO 1

- Scrivere un programma JAVA che attivi K thread, chiamati "T1", "T2", ..., "TK". Tutti i thread sono caratterizzati dallo stesso comportamento: ogni thread stampa i primi N numeri naturali, senza andare a capo (K e N sono dati in input dall'utente). Accanto ad ogni numero deve essere visualizzato il nome del thread che lo ha generato, ad esempio usando il formato ": n [Tk] :". Tra la stampa di un numero e quella del numero successivo ogni thread deve sospendersi per un intervallo di tempo la cui durata è scelta in modo casuale tra 0 e 1000 millisecondi.
- Sviluppare due diverse versioni del programma che utilizzino le due tecniche per l'attivazione di threads presentate in questa lezione.

COME INTERRUOMPERE UN THREAD

- Un thread può essere interrotto, durante il suo ciclo di vita, ad esempio mentre sta 'dormendo' in seguito all'esecuzione di una `sleep()`
- L'interruzione di un thread causa una `InterruptedException`

```
public class SleepInterrupt implements Runnable {  
    public void run ( ){  
        try{ System.out.println("vado a dormire per 20 secondi");  
            Thread.sleep(20000);  
            System.out.println ("svegliato"); }  
        catch ( InterruptedException x )  
            { System.out.println ("interrotto"); return;};  
        System.out.println("esco normalmente");}}
```

COME INTERRUOMPERE UN THREAD

```
public class SleepMain {  
    public static void main (String args [ ]) {  
        SleepInterrupt si = new SleepInterrupt();  
        Thread t = new Thread (si);  
        t.start ( );  
        try {  
            Thread.sleep(2000);  
        } catch (InterruptedException x) { };  
        System.out.println("Interrompo l'altro thread");  
        t.interrupt( );  
        System.out.println ("sto terminando...");  
    }  
}
```

COME INTERROMPERE UN THREAD

- Il metodo `interrupt()`:
 - interrompe il thread causando una `InterruptedException` se era sospeso (con `wait()`, `sleep()`, `join()`, I/O)
 - altrimenti imposta a true un **flag** nel descrittore del thread

E' possibile testare il valore del flag mediante:

- `public static boolean interrupted ()` STATIC !!!
restituisce il valore del **flag** (relativo al thread in esecuzione); riporta il valore del **flag** a false
- `public boolean isInterrupted ()`
restituisce il valore del **flag**, relativo al thread su cui è invocato

Nota: se esiste un interrupt pendente al momento dell'esecuzione della `sleep()`, viene sollevata immediatamente una `InterruptedException`.

ESERCIZIO 2: INTERROMPERE UN THREAD

Scrivere un programma che avvia un thread che va in sleep per 10 secondi. Il programma principale interrompe il thread dopo 5 secondi. Il thread deve catturare l'eccezione e stampare il tempo trascorso in sleep.

Per ottenere l'ora corrente usare il metodo

`System.currentTimeMillis()`, consultandone la documentazione on line.

Esercizio 3: CALCOLO DI π

Scrivere un programma che attiva un thread T che effettua il calcolo approssimato di π . Il programma principale riceve in input da linea di comando due argomenti:

- un parametro che indica il grado di accuratezza (accuracy) per il calcolo di π
- il tempo massimo di attesa dopo cui il programma principale interrompe il thread T.

Il thread T effettua un ciclo infinito per il calcolo di π usando la serie di Gregory-Leibniz ($\pi = 4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 \dots$). Il thread esce dal ciclo quando una delle due condizioni seguenti risulta verificata:

- 1) il thread è stato interrotto, oppure
- 2) la differenza tra il valore stimato di π ed il valore Math.PI (della libreria Java) è minore di accuracy

PRIORITA' DEI THREADS

- Ogni thread ha una **priorità** che può essere cambiata durante l'esecuzione. La priorità rappresenta **un suggerimento** allo schedulatore sull'ordine con cui i threads possono essere inviati in esecuzione
- La priorità viene ereditata dal thread **padre**
- Metodi per gestire la priorità
 - **public final void setPriority** (int newPriority)
 - Può essere invocato prima dell'attivazione del thread o durante la sua esecuzione (da un thread che ne ha il diritto)
 - **public final int getPriority** ()
 - Thread.MAX_PRIORITY (= 10)
 - Thread.MIN_PRIORITY (= 1)
 - Thread.NORM_PRIORITY (= 5)

THREAD CON DIVERSE PRIORITA'

Scriviamo un programma dove il thread **main** ha priorità 5 (come da default), e attiva i thread

- Thread A con priorità 8 e poi 3
- Thread B con priorità 2
- Thread D con priorità 7, che crea
 - Thread C con la stessa priorità, 7.

THREAD PRIORITY: Definizione dei task

```
public class Task1 implements Runnable{
    public void run( ) {
        for (int i = 0; i < 4; i++){
            Thread t = Thread.currentThread ( );
            System.out.println(t.getName() +
                " ha priorit  " + t.getPriority());
            try {Thread.sleep (2000);
            } catch(InterruptedExce ion x) {} } }
```

```
public class Task2 implements Runnable{
    public void run( ) {
        Thread tC = new Thread(new Task1(), "thread C");
        tC.start(); new Task1().run(); } }
```

THREAD PRIORITY: il main

```
public class MainPriority{  
    public static void main (String[ ] args) {  
        Thread tA = new Thread(new Task1(),"thread A");  
        Thread tB = new Thread(new Task1(),"thread B");  
        Thread tD = new Thread(new Task2(),"thread D");  
        tA.setPriority(8); tB.setPriority(2); tD.setPriority(7);  
        tA.start(); tB.start(); tD.start();  
        try{Thread.sleep(3000);}  
        catch(InterruptedException x) { }  
        tA.setPriority(3);  
        System.out.println("main ha priorit  " +  
            Thread.currentThread().getPriority()); } }
```

THREAD PRIORITY: Un possibile output

```
thread B ha priorit  2
thread D ha priorit  7
thread C ha priorit  7
thread A ha priorit  8
thread B ha priorit  2
thread D ha priorit  7
thread A ha priorit  8
thread C ha priorit  7
main ha priorit  5
thread B ha priorit  2
thread D ha priorit  7
thread A ha priorit  3
thread C ha priorit  7
thread B ha priorit  2
thread D ha priorit  7
thread A ha priorit  3
thread C ha priorit  7
```