



Lezione n.1
LPR- Informatica Applicata
Thread Programming in JAVA

23/2/2009

Laura Ricci

INFORMAZIONI UTILI

- **Orario del Corso:**

Lunedì 9.00-11.00 - Lezione -

Lunedì 11.00-13.00 - Laboratorio -

Lunedì 14.00-16.00 - Recupero e Correzione Esercizi-

- **Ricevimento:** Giovedì ore 15.00-18.00

- **Laboratorio**

- **Obiettivo:** Verifica degli esercizi assegnati nelle lezioni teoriche
- Ogni esercizio deve essere consegnato via e-mail entro 15 giorni dalla data di assegnazione

- **Modalità di esame:** Progetto finale + Orale

Orale: discussione del progetto + domande sugli argomenti trattati nelle lezioni teoriche (soprattutto quelli non coperti dal progetto)



INFORMAZIONI UTILI

- **Prerequisiti:** conoscenza del linguaggio JAVA, in particolare definizione di packages, gestione delle eccezioni, streams
- Linguaggio di programmazione: JAVA 1.6, librerie JAVA.NET, JAVA.RMI.
- Ambiente di Sviluppo: Linux
 - Uso 'naif' da shell JAVAC + JAVA
 - Uso di IDE Eclipse, Netbeans

INFORMAZIONI UTILI

- **Materiale Didattico:**
 - Lucidi delle lezioni
 - Per la parte relativa ai threads
 - Bruce Eckel - *Thinking in JAVA - Volume 3 - Concorrenza e Interfacce Grafiche*
 - Per la parte relativa alla programmazione di rete
 - Esmond Pitt *Fundamental Networking in JAVA*
- **Materiale di Consultazione:**
 - Harold - *JAVA Network Programming 3rd edition* O'Reilly 2004).
 - K.Calvert, M.Donhoo, *TCP/IP Sockets in JAVA*, Practical Guide for Programmers
 - Per i costrutti di base Cay Horstmann - *Concetti di Informatica e Fondamenti di Java 2*

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

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



PROGRAMMA DEL CORSO

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.

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

Secure Sockets

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

Meccanismi RMI Avanzati: Il meccanismo delle callback.



MULTITHREADING:DEFINIZIONE

Definizioni:

Thread: flusso sequenziale di esecuzione

Multithreading:

- consente di definire più flussi di esecuzione (threads) all' interno dello stesso processo
- 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'utilizzo della CPU quando il programma si blocca (esempio: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 molteplici 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...

ATTIVAZIONE DI THREADS

- Thread "main": viene attivato in ogni applicazione JAVA. Avvia l'esecuzione dell'applicazione
- Altri threads sono attivati automaticamente da JAVA (gestore eventi interfaccia, garbage collector,...)
- Ogni thread durante la sua esecuzione può **attivare** altri threads
- Per attivare un nuovo thread
 - Definire una classe *C* che implementi l'interfaccia **Runnable**, creare un oggetto *O* istanza di *C*, quindi creare **un thread** è assandogli *O*
oppure
 - estendere la classe `java.lang.Thread`

JAVA.LANG.THREAD: GERARCHIA

- Interfaccia Runnable: appartiene al package java.lang
- Contiene solo la segnatura di un metodo `void run()`
- Un classe `C` che implementa l'interfaccia deve fornire un'implementazione del metodo `run()`
- Un'oggetto `O` istanza della classe `C` rappresenta un `task Ta`, cioè un frammento di codice che può essere eseguito in un thread (flusso di controllo) separato
- La creazione di `O` non implica la creazione di un thread per la sua esecuzione. E' necessario istanziare un oggetto Thread `T` e passare a `T` il task `Ta`
- **Esempio:** Implementare un task `Decollo` che implementi un 'conto alla rovescia' e che, alla fine del conto, invii un segnale 'Via!'

IL TASK DECOLLO

```
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



CREAZIONE DI THREADS

- **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 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 dalla `Runnable` 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`

ATTIVAZIONE DI THREAD

```
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!!!),

ATTIVAZIONE DI THREADS

```
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");}}}
```

Output generato:

Aspetto il decollo

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



IL CLASSE JAVA.LANG.THREAD

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

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

Costruttori:

- Sono definiti diversi costruttori che differiscono per i parametri utilizzati (esempio task da eseguire, nome del thread, gruppo a cui appartiene il thread,....vedere le 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

THREADS: ISTRUZIONI PER L'USO

Per definire tasks ed attivare threads che li eseguano

- Definire una classe *C* che implementi l' interfaccia `Runnable`, cioè **implementare** il metodo `run`. In questo modo si definisce un oggetto `Runnable`, cioè un **task** che può essere eseguito
- Allocare un'istanza *R* di *C*
- Per costruire il thread, utilizzare il costruttore
`public Thread (Runnable target)`
passando il task *R* come parametro
- attivare il thread con una **`start()`** come nell'esempio precedente.

ATTIVAZIONE DI THREADS

Il metodo `start()`

- segnala allo schedulatore (tramite la 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" precede quelle effettuate dai threads. Questo significa che il controllo è stato restituito al thread chiamante (il thread associato al main) prima che sia iniziata l'esecuzione dei threads attivati

LA CLASSE JAVA.LANG.THREAD

- 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() implica l'invocazione del metodo run() precedente. A sua volta, viene invocato il metodo RR = run() sull'oggetto che implementa la Runnable (se questo è presente).
- Qualsiasi istruzione eseguita dal thread fa parte di RR o di un metodo invocato da RR. Inoltre il thread termina con l'ultima istruzione di RR.
- 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 contenuto nel metodo `run()` riscritto in *C*
- Invocare il metodo `start()` su *O*. Tale metodo attiva il thread ed invoca il metodo riscritto.

ESTENSIONE DELLA CLASSE THREADS

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

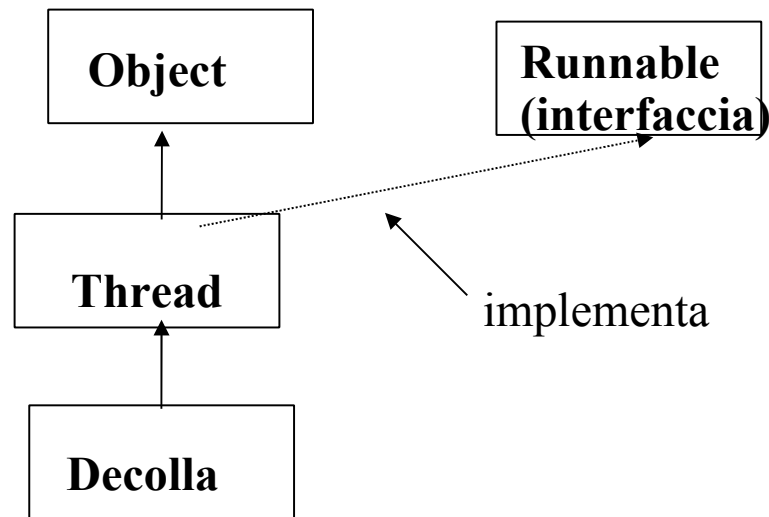
public class MainThread {
    public static void main(String [ ]args)    {
        decollo d = new decollo( );
        d.start();
        System.out.println("Aspetto il Decollo");    }}
```



GERARCHIA DELLE CLASSI

- La classe Thread estende Objects ed implementa l'interfaccia

Runnable



- Decolla** estende threads ed effettua overriding del 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 threads 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.**

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 il sollevamento di 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 INTERROMPERE 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

Implementazione del metodo `interrupt()`: imposta a true un valore booleano nel descrittore del thread

se esistono interrupts pendenti \Rightarrow il flag vale true

E' possibile testare il valore del flag mediante:

- **public static boolean** `Interrupted()` (`Thread.Interrupted()`) restituisce il valore booleano che segnala l'interruzione
il valore booleano si riferisce al thread che ha invocato il metodo
riporta il valore del flag a false
- **public boolean** `isInterrupted()`
- Puo' essere invocato su un oggetto di tipo thread

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

ESERCIZIO 1: 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 un parametro che indica il grado di accuratezza (accuracy) per il calcolo di π ed 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:

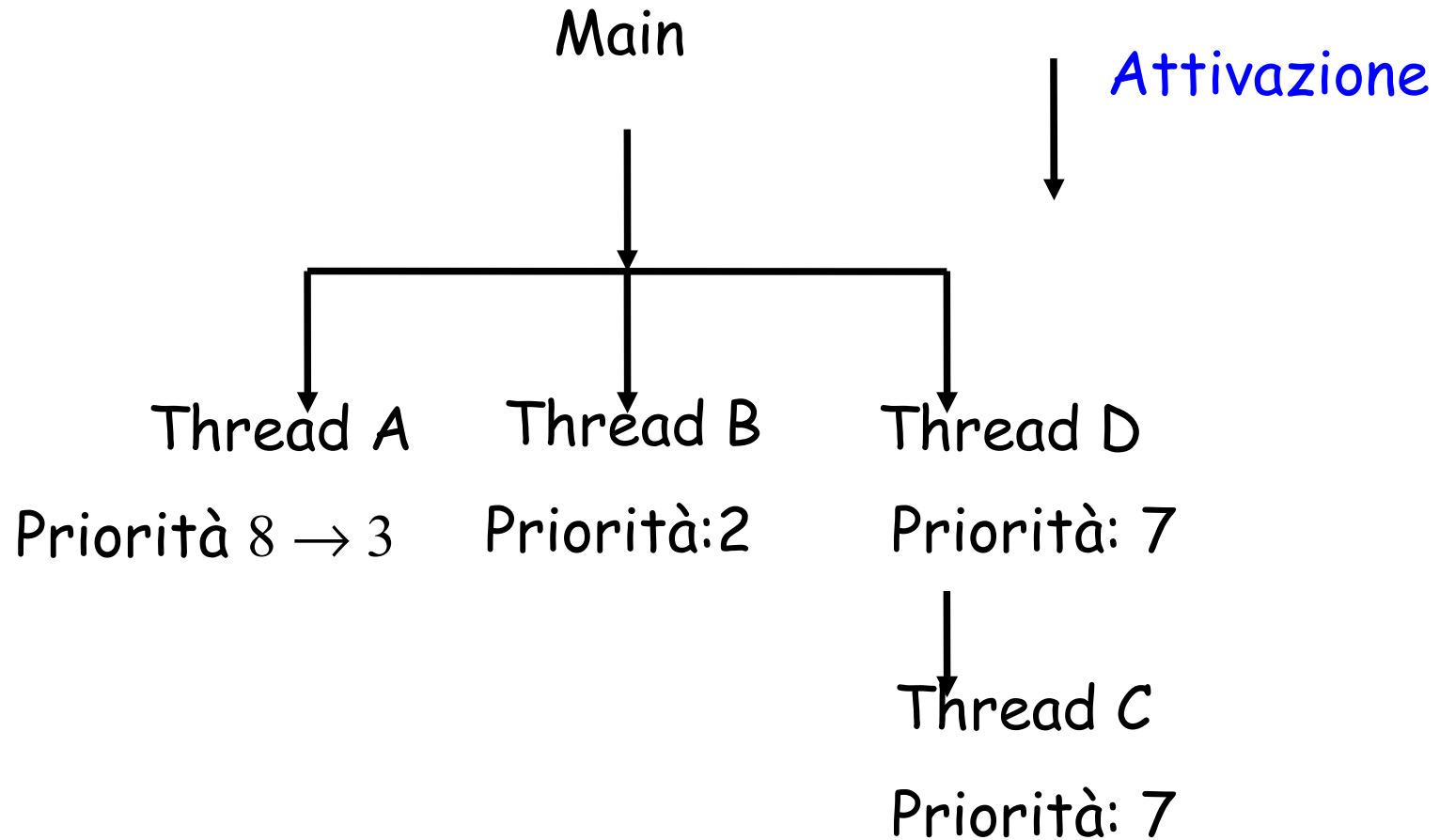
- 1) il thread è stato interrotto
- 2) la differenza tra il valore stimato di π ed il valore `Math.PI` (della libreria Java) è minore di accuracy



ATTIVAZIONE DI THREADS

- E' possibile associare ad ogni thread **una priorità**.
- La priorità consente di 'dare un suggerimento' allo schedulatore sull'ordinamento con cui i threads vengano 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.
 - `public final int getPriority ()`
 - `Thread.MAX_PRIORITY`, `Thread.MIN_PRIORITY`,
`Thread.NORM_Priority`

ATTIVAZIONE DI THREADS



ATTIVAZIONE DI THREADS

Il seguente programma implementa lo schema mostrato nel lucido precedente:

```
public class SetPriority extends Object{
    private static Runnable makeRunnable( ) {
        Runnable r = new Runnable ( ) {
            public void run( ) {
                for ( int i=0; i<5 ; i++)
                    {Thread t = Thread.currentThread ( );
                     System.out.println("run"+t.getPriority( +t.getName( ));
                     try {Thread.sleep (2000);}
                     catch (InterruptedException x) { }
                    } } };
            return r ;}
}
```



ESERCIZIO 3: THREAD PRIORITY

```
public static void main (String[ ] args) {  
    Thread threadA = new Thread(makeRunnable(), "threadA");  
    threadA.setPriority(8);  
    threadA.start();  
    Thread threadB = new Thread(makeRunnable(), "threadB");  
    threadB.setPriority(2);  
    threadB.start();  
    Runnable r = new Runnable() {  
        public void run() {  
            Thread threadC = new Thread(makeRunnable(),  
                                         "threadC");  
            threadC.start();  } };
```

ESERCIZIO 3: THREAD PRIORITY

```
Thread threadD = new Thread(r, "threadD");
threadD.setPriority(7);
threadD.start();
try{Thread.sleep(3000);}
catch (InterruptedException x) { }
threadA.setPriority(3);
System.out.println("main"+threadA.getPriority());
    } }
```

ESERCIZIO 3: THREAD PRIORITY

Priority8 nomethread A

priority7 nomethread C

priority2 nomethread B

priority8 nomethread A

priority2 nomethread B

priority7 nomethread C

main3

priority3 nomethread A

priority2 nomethread B

priority7 nomethread C

priority3 nomethread A

priority2 nomethread B

priority7 nomethread C

priority3 nomethread A

priority2 nomethread B priority7 nomethread C



THREAD POOLING

Concetti fondamentali:

- L'utente struttura l'applicazione mediante un **insieme di tasks**.
- **Task** = segmento di codice che può essere eseguito da un esecutore. Può essere definito come un oggetto di tipo **Runnable**
- **Thread** = **esecutore** in grado di eseguire tasks.
- Uno stesso thread può essere utilizzato per eseguire diversi tasks, durante la sua esecuzione
- **Thread Pool** = Struttura dati la cui dimensione massima è **prefissata**, che contiene riferimenti ad un insieme di threads
- I thread del pool possono essere utilizzati per eseguire i tasks sottomessi per l'esecuzione dall'utente al thread pool

THREAD POOLING

L'utente

- Definisce i tasks della applicazione
- Crea un **thread di pool** e stabilisce una **politica per la gestione dei threads del pool**, la politica stabilisce
 - quando i threads del pool **vengono attivati**: (al momento della creazione del pool, on demand, in corrispondenza dell'arrivo di un nuovo task,...)
 - se e quando è opportuno terminare l'esecuzione di un thread (ad esempio se non c'è un numero sufficiente di tasks da eseguire)
- Sottomette i tasks per l'esecuzione al thread pool.

THREAD POOLING

- L'applicazione sottomette un task T al gestore del thread pool
- Il gestore sceglie un thread dal pool per l'esecuzione di T. Scelte possibili:
 - utilizzare un thread attivato in precedenza, ma inattivo al momento dell'arrivo del nuovo task
 - creare un nuovo thread, purchè non venga superata la dimensione massima del pool
 - memorizzare il task in una struttura dati, in attesa di eseguirlo
 - respingere la richiesta di esecuzione del task
- Il numero di threads attivi nel pool può variare dinamicamente

THREAD POOLING: MOTIVAZIONI

- Tempo stimato per la creazione di un thread: qualche centinaio di microsecondi.
- Creazione di un alto numero di threads può non essere tollerabile per certe applicazioni
- Thread Pooling
 - **diminuisce l'overhead** dovuto alla creazione di un gran numero di threads. Lo stesso thread può essere **riutilizzato per l'esecuzione di più di un tasks**
 - permette una **migliore strutturazione del codice** dell'applicazione. Tutta la gestione dei threads può essere delegata al gestore del thread pool

LIBRERIA JAVA.UTIL.CONCURRENT

- L'implementazione del thread pooling:
 - Fino a J2SE 4 doveva essere realizzata a livello applicazione
 - J2SE 5.0 definisce la libreria `java.util.concurrent` che contiene metodi per
 - Creare un thread pool e il gestore associato
 - Definire la struttura dati utilizzata per la memorizzazione dei tasks in attesa
 - Decidere specifiche politiche per la gestione del pool

CREARE UN THREADPOOL EXECUTOR

JAVA 5 definisce

- Alcune interfacce che definiscono servizi generici di esecuzione.

```
public interface Executor
    {public void execute (Runnable task); }
public interface ExecutorService extends Executor
    {..... }
```

- diversi esecutori che implementano il generico ExecutorService (ThreadPoolExecutor, ScheduledThreadPoolExecutor,..)
- la classe Executors che opera come una Factory in grado di generare oggetti di tipo ExecutorService con **comportamenti predefiniti**.

I tasks devono essere incapsulati in oggetti di tipo Runnable e passati a questi esecutori, mediante invocazione del metodo `execute()`



THREAD POOLING: ESEMPI

```
import java.util.concurrent.*;

public class decollo implements Runnable{

    int countDown = 3; // Predefinito

    public decollo( ) { }

    public String status( ) {

        return "#" + Thread.currentThread() +

            "(" + (countDown > 0 ? countDown: "Via!!!") + ")," ;}

    public void run( ) {

        while (countDown-- > 0) {

            System.out.print(status());

            try{ Thread.sleep(100);}

            catch (InterruptedException e) { } } } }
```

THREAD POOLING: ESEMPIO 1

```
public class esecutori {  
    public static void main(String[] args)  
    {ExecutorService exec = Executors.newCachedThreadPool();  
        for (int i=0; i<3; i++) {  
            exec.execute(new decollo());}}}
```

`newCachedThreadPool()` crea un pool in cui, quando viene sottomesso un task

- viene creato un nuovo thread se tutti i threads del pool sono occupati nell'esecuzione di altri tasks.
- viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente, se disponibile.
- se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina viene rimosso dalla cache.
- adatti nel caso di asincroni tasks di piccola dimensione

ESEMPIO1:OUTPUT

Output del programma:

```
#Thread[pool-1-thread-2,5,main](2)
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-3,5,main](2)
#Thread[pool-1-thread-3,5,main](1),
#Thread[pool-1-thread-2,5,main](1),
#Thread[pool-1-thread-1,5,main](1)
#Thread[pool-1-thread-2,5,main](Via!!!),
#Thread[pool-1-thread-1,5,main](Via!!!)
#Thread[pool-1-thread-3,5,main](Via!!!),
```

ESEMPIO 2: OUTPUT

```
import java.util.concurrent.*;

public class esecutori {

    public static void main(String[] args)

        {ExecutorService exec = Executors.newCachedThreadPool();

        for (int i=0; i<3; i++){

            exec.execute(new decollo( ));

            try{

                Thread.sleep (10000);

            catch (InterruptedException e) { } } } }
```

La sottomissione di tasks al pool viene distanziata di 10 secondi. In questo modo l'esecuzione precedente è terminata ed è possibile riutilizzare un thread attivato precedentemente



THREAD POOLING: ESEMPIO 2

```
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-1,5,main](1),
#Thread[pool-1-thread-1,5,main](Via!!!),
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-1,5,main](1)
#Thread[pool-1-thread-1,5,main](Via!!!)
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-1,5,main](1)
#Thread[pool-1-thread-1,5,main](Via!!!),
```

THREAD POOLING: ESEMPIO 3

```
import java.util.concurrent.*;

public class esecutori {

    public static void main(String[] args)
    {ExecutorService exec = Executors.newFixedThreadPool(2);
    for (int i=0; i<3; i++){
        exec.execute(new liftoff());} } }
```

`newFixedThreadPool (int num)` crea un pool in cui,

- vengono creati `num threads`, al momento della inizializzazione del pool . Questi e solo questi thread verranno utilizzati durante la vita del pool.
- quando viene sottomesso un task T
 - se tutti i threads sono occupati nell'esecuzione di altri tasks, T viene inserito in una coda, gestita dall'ExecutorService
 - se almeno un thread è inattivo, viene utilizzato quel thread

ESEMPIO 3: OUTPUT

```
#Thread[pool-1-thread-1,5,main](2),  
#Thread[pool-1-thread-2,5,main](2),  
#Thread[pool-1-thread-2,5,main](1),  
#Thread[pool-1-thread-1,5,main](1),  
#Thread[pool-1-thread-1,5,main](Via!!!),  
#Thread[pool-1-thread-2,5,main](Via!!!),  
#Thread[pool-1-thread-1,5,main](2),  
#Thread[pool-1-thread-1,5,main](1),  
#Thread[pool-1-thread-1,5,main](Via!!!),
```