



# Lezione n.2

LPR-B-08

## Thread Pooling, Callabe e Future e Indirizzi IP

1/10/2008

Andrea Corradini

Laura Ricci

# ATTENDERE LA TERMINAZIONE DI UN THREAD: METODO `join()`

---

- Un thread `J` può invocare il metodo `join()` su un oggetto `T` di tipo thread
- `J` rimane sospeso sulla `join()` fino alla terminazione di `T`.
- Quando `T` termina, `J` riprende l'esecuzione con l'istruzione successiva alla `join()`.
- Un thread sospeso su una `join()` può essere interrotto da un altro thread che invoca su di esso il metodo `interrupt()`.
- Il metodo può essere utilizzato nel `main` per attendere la terminazione di tutti i threads attivati.

# JOINING A THREAD

```
public class Sleeper extends Thread {  
    private int period;  
    public Sleeper (String name, int sleepPeriod){  
        super(name);  
        period = sleepPeriod;  
        start( ); }  
    public void run( ){  
        try{  
            sleep (period); }  
        catch (InterruptedException e){  
            System.out.println(getName( )+" e' stato interrotto"); return;}  
            System.out.println(getName()+" e' stato svegliato normalmente");}}}
```

# JOINING A THREAD

```
public class Joiner extends Thread {  
    private Sleeper sleeper;  
    public Joiner(String name, Sleeper sleeper){  
        super(name);  
        this.sleeper = sleeper;  
        start( ); }  
    public void run( ){  
        try{  
            sleeper.join( );  
        }catch(InterruptedException e) {  
            System.out.println("Interrotto"); return; }  
        System.out.println(getName( )+" join completed"); }}
```

# JOINING A THREAD

```
public class Joining {  
    public static void main(String[ ] args){  
        Sleeper assonnato = new Sleeper("Assonnato", 1500);  
        Sleeper stanco = new Sleeper("Stanco", 1500);  
        new Joiner("WaitforAssonnato", assonnato);  
        new Joiner("WaitforStanco", stanco);  
        stanco.interrupt(); } }
```

## Output:

Stanco è stato interrotto

WaitforStanco join completed

Assonnato è stato svegliato normalmente

WaitforAssonnato join completed

# 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 vita.
- **Thread Pool** = Struttura dati (normalmente con dimensione massima prefissata), che contiene riferimenti ad un insieme di threads
- I thread del pool vengono utilizzati per eseguire i tasks sottomessi dall'utente.

# THREAD POOLING: MOTIVAZIONI

- Tempo stimato per la creazione di un thread: qualche centinaio di microsecondi.
- La 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 thread: lo stesso thread può essere riutilizzato per l'esecuzione di più di un tasks
  - Permette una semplificazione e una migliore strutturazione del codice dell'applicazione: tutta la gestione dei threads può essere delegata al gestore del pool (che va può essere riutilizzato in altre applicazioni...)

# THREAD POOLING: USO

---

L'utente

- Definisce i tasks dell'applicazione
- Crea un **pool di thread** e stabilisce una **politica per la gestione dei threads all'interno 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 pool di thread.



# THREAD POOLING: IL GESTORE

- L'applicazione sottomette un task T al gestore del pool di thread
- Il gestore sceglie un thread dal pool per l'esecuzione di T.  
Le scelte possibili sono:
  - 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

# LIBRERIA `java.util.concurrent`

- L'implementazione del thread pooling:
  - Fino a **J2SE 1.4** doveva essere realizzata a livello applicazione
  - **J2SE 5.0** introduce la libreria `java.util.concurrent` che contiene metodi per
    - Creare un pool di thread 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

Il package `java.util.concurrent` definisce:

- Alcune interfacce che definiscono servizi generici di esecuzione...

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

```
public interface ExecutorService extends Executor{... }
```

- diverse classi che implementano `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( )`

# ESEMPI: IL TASK

```
public class TakeOff implements Runnable{
    int countDown = 3; // Predefinito
    public String status( ){
        return "#" + Thread.currentThread() +
            "(" + (countDown > 0 ? countDown: "Via!!!") + "),";
    }
    public void run( ){
        while (countDown-- > 0){
            System.out.println(status());
            try{ Thread.sleep(100);}
            catch(InterruptedException e){ }
        }
    }
}
```

# THREAD POOLING: ESEMPIO 1

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

public class Esecutori1 {

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

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

- viene creato un nuovo thread se tutti i thread 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 e viene rimosso dalla cache.

# 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!!!),
```

## THREAD POOLING: ESEMPIO 2

```
import java.util.concurrent.*;
public class Esecutori2 {
    public static void main(String[] args){
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i=0; i<3; i++){
            exec.execute(new TakeOff( ));
            try {Thread.sleep (4000);}
                catch(InterruptedException e) { } } } }
```

La sottomissione di tasks al pool viene distanziata di 4 secondi. In questo modo l'esecuzione precedente è terminata ed è possibile riutilizzare un thread attivato precedentemente

## ESEMPIO 2: OUTPUT

---

```
#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 Esecutori3 {

    public static void main(String[] args){

        ExecutorService exec = Executors.newFixedThreadPool(2);

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

            exec.execute(new TakeOff());} } }
```

`newFixedThreadPool (int i)` crea un pool in cui, quando viene sottomesso un task

- Viene riutilizzato un thread del pool, se inattivo
- Se tutti i thread sono occupati nell'esecuzione di altri tasks, il task viene inserito in una coda, gestita dall'`ExecutorService` e condivisa da tutti i 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!!!),
```

# THREAD POOL EXECUTOR

```
package java.util.concurrent;
```

```
public class ThreadPoolExecutor implements ExecutorService{
```

```
    public ThreadPoolExecutor ( int corePoolSize,
```

```
        int maximumPoolSize,
```

```
        long keepAliveTime,
```

```
        TimeUnit unit,
```

```
        BlockingQueue <Runnable> workqueue);
```

```
    ... }
```

- crea un oggetto di tipo `ExecutorService`
- consente di definire **gestori di thread pool** con una **politica di gestione personalizzata**

# THREAD POOL EXECUTOR

```
public ThreadPoolExecutor ( int corePoolSize,  
                           int maximumPoolSize,  
                           long keepAliveTime,  
                           TimeUnit unit,  
                           BlockingQueue <Runnable> workqueue);  
... }
```

- **corePoolSize**, **maximumPoolSize**, e **keepAliveTime** controllano la gestione dei threads del pool
- **workqueue** è una struttura dati usata per memorizzare gli eventuali tasks in attesa di esecuzione

# THREAD POOL: GESTIONE DINAMICA

---

- **corePoolSize**: dimensione minima del pool
  - È possibile allocare **corePoolSize** thread al momento della creazione del pool mediante il metodo **prestartAllCoreThreads( )**. I thread creati rimangono inattivi in attesa di tasks da eseguire.
  - Oppure i thread possono essere creati "on demand". Quando viene sottomesso un nuovo task, viene creato un nuovo thread, anche se alcuni dei threads già creati sono inattivi. L'obiettivo è di riempire il "core" del pool prima possibile.
- **maximumPoolSize**: dimensione massima del pool

# THREAD POOL: GESTIONE DINAMICA

- Se sono in esecuzione tutti i core thread, un nuovo task sottomesso viene inserito in una coda **Q**.
  - **Q** deve essere una istanza di **BlockingQueue<Runnable>**
  - **Q** viene passata al momento della costruzione del **ThreadPoolExecutor** (ultimo parametro del costruttore)
  - E' possibile scegliere diversi tipi di coda (sottoclassi di **BlockingQueue**). Il tipo di coda scelto **influisce sullo scheduling**.
- I task vengono poi prelevati da **Q** e inviati ai threads che si rendono disponibili
- Solo quando **Q** risulta piena si crea un nuovo thread attivando così **k** threads, **corePoolSize ≤ k ≤ maxPoolSize**

# THREAD POOL: GESTIONE DINAMICA

---

Da questo punto in poi, quando viene sottomesso un nuovo task **T**

- se esiste un thread **th** inattivo, **T** viene assegnato a **th**
- se non esistono threads inattivi, si preferisce sempre accodare un task piuttosto che creare un nuovo thread
- solo se la coda è piena, si attivano nuovi threads
- Se la coda è piena e sono attivi *MaxPoolSize* threads, il task **viene respinto** e viene sollevata un'eccezione

# THREAD POOL: GESTIONE DINAMICA

Supponiamo che un thread **th** termini l'esecuzione di un task, e che il pool contenga **k** threads

- Se **k**  $\leq$  **core**: il thread **si mette in attesa** di nuovi tasks da eseguire. L'attesa è indefinita.
- Se **k**  $>$  **core**, si considera il **timeout** definito al momento della costruzione del thread pool
  - se nessun task viene sottomesso **entro il timeout**, **th** termina la sua esecuzione, riducendo così il numero di threads del pool
- Il **timeout** è determinato dai parametri **long keepAliveTime** e **TimeUnit unit** del costruttore, quindi consiste di:
  - un valore (es: 50000L) e
  - l'unità di misura utilizzata (es: **TimeUnit.MILLISECONDS**)



# THREAD POOL: TIPI DI CODA

- **SynchronousQueue**: dimensione uguale a 0. Ogni nuovo task T
  - viene eseguito immediatamente oppure respinto.
  - T viene eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (numero di threads  $\leq$  **maxPoolSize**)
- **LinkedBlockingQueue**: dimensione illimitata
  - E' sempre possibile accodare un nuovo task, nel caso in cui tutti i tasks attivi nell'esecuzione di altri tasks
  - La dimensione del pool di **non supererà mai core**
- **ArrayBlockingQueue**: dimensione limitata, stabilita dal programmatore

# THREAD POOLING: UN ESEMPIO

---

- Dato un intero **K**, si vuole calcolare, per ogni valore  $n < K$  il valore dell'**n**-esimo numero di Fibonacci
- Si definisce un task T che effettua il calcolo del numero di Fibonacci di **n** (valore passato come parametro)
- Si attiva un **ThreadPoolExecutor**, definendo la politica di gestione del pool di thread mediante i parametri passati al costruttore
- Si passa all'esecutore una istanza di T per ogni  $n < K$ , invocando il metodo **execute()**

# FIBONACCI TASK (I)

```
public class FibTask implements Runnable{
    int n; String id;
    public FibTask(int n, String id){
        this.n = n; this.id = id;
    }
    private int fib(int n){
        if (n == 0 || n == 1) return n;
        if (n==1) return 1;
        return fib(n - 1) + fib(n - 2);
    }
}
```

## FIBONACCI TASK (II)

```
public void run( ){
    try{
        Thread t = Thread.currentThread( );
        System.out.println("Starting task " + id + " su " + n +
            " eseguito dal thread " + t.getName( ));
        System.out.println("Risultato " + fib(n) + " da task " + id +
            " eseguito dal thread " + t.getName( ));
    } catch (Exception e){
        e.printStackTrace();
    }
}
```

# FIBONACCI THREAD POOL

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

public class ThreadPoolTest {

    public static void main (String [] args){
        int nTasks = Integer.parseInt(args[0]); // # di tasks da eseguire
            // dimensione del core pool

        int corePoolSize = Integer.parseInt(args[1]);
            // massima dimensione del pool

        int maxPoolSize = Integer.parseInt(args[2]);

        ThreadPoolExecutor tpe = new ThreadPoolExecutor (corePoolSize,
            maxPoolSize,
            50000L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>( ));
    }
}
```

# FIBONACCI THREAD POOL

```
FibTask [ ] tasks = new FibTask[nTasks];
```

```
for (int i=0; i< nTasks; i++){
```

```
    tasks[i] = new FibTask(i, "Task" + i);
```

```
    tpe.execute(tasks[i]);
```

```
    System.out.println("dimensione del pool " + tpe.getPoolSize());
```

```
}
```

```
tpe.shutdown();
```

```
}
```

```
}
```

# GESTIONE DINAMICA: ESEMPI

**Parametri: tasks= 8, core = 3, MaxPoolSize= 4, SynchronousQueue, timeout=50000msec**

dimensione del pool 1

Starting task Task0 eseguito da pool-1-thread-1

Risultato 0 da task Task0 eseguito da pool-1-thread-1

dimensione del pool 2

dimensione del pool 3

dimensione del pool 3

Starting task Task3 eseguito da pool-1-thread-1

Starting task Task1 eseguito da pool-1-thread-2

Starting task Task2 eseguito da pool-1-thread-3

dimensione del pool 4

Risultato 1 da task Task1 eseguito da pool-1-thread-2

Starting task Task4 eseguito da pool-1-thread-4

Risultato 2 da task Task3 eseguito da pool-1-thread-1

Risultato 1 da task Task2 eseguito da pool-1-thread-3

[java.util.concurrent.RejectedExecutionException](#)

Risultato 3 da task Task4 eseguito da pool-1-thread-4

# GESTIONE DINAMICA: ESEMPI

Tutti I threads attivati inizialmente mediante `tpe.prestartAllCoreThreads( );`

**Parametri: tasks= 8, core = 3, MaxPoolSize= 4, SynchronousQueue**

dimensione del pool 3

Starting task Task0 eseguito da pool-1-thread-3

dimensione del pool3

Risultato 0 da task Task0 eseguito da pool-1-thread-3

dimensione del pool 3

Starting task Task2 eseguito da pool-1-thread-1

Starting task Task1 eseguito da pool-1-thread-2

Risultato 1 da task Task2 eseguito da pool-1-thread-1

Risultato 1 da task Task1 eseguito da pool-1-thread-2

Starting task Task3 eseguito da pool-1-thread-3

dimensione del pool 3

Risultato 2 da task Task3 eseguito da pool-1-thread-3

dimensione del pool 3



# GESTIONE DINAMICA: ESEMPI

*(CONTINUA)*

Starting task Task4 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task5 eseguito da pool-1-thread-3

Risultato 3 da task Task4 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task6 eseguito da pool-1-thread-1

Risultato 5 da task Task5 eseguito da pool-1-thread-3

dimensione del pool 3

Starting task Task7 eseguito da pool-1-thread-2

Risultato 8 da task Task6 eseguito da pool-1-thread-1

Risultato 13 da task Task7 eseguito da pool-1-thread-2

# GESTIONE DINAMICA: ESEMPI

**Parametri: tasks= 10, core = 3, MaxPoolSize= 4, SynchronousQueue,  
timeout=0msec**

dimensione del pool 1

Starting task Task0 eseguito da pool-1-thread-1

Risultato 0 da task Task0 eseguito da pool-1-thread-1

dimensione del pool 2

Starting task Task1 eseguito da pool-1-thread-2

Risultato 1 da task Task1 eseguito da pool-1-thread-2

dimensione del pool 3

dimensione del pool 3

Starting task Task2 eseguito da pool-1-thread-3

Starting task Task3 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task4 eseguito da pool-1-thread-1

Risultato 1 da task Task2 eseguito da pool-1-thread-3

Risultato 2 da task Task3 eseguito da pool-1-thread-2

dimensione del pool 4

# GESTIONE DINAMICA:ESEMPI

*(CONTINUA)*

```
Risultato 3 da task Task4 eseguito dapool-1-thread-1
Starting task Task5 eseguito da pool-1-thread-4
dimensione del pool 3
Starting task Task6 eseguito da pool-1-thread-2
Risultato 5 da task Task5 eseguito dapool-1-thread-4
Starting task Task7 eseguito da pool-1-thread-1
dimensione del pool 3
Risultato 8 da task Task6 eseguito dapool-1-thread-2
dimensione del pool 3
Starting task Task8 eseguito da pool-1-thread-4
dimensione del pool 3
Starting task Task9 eseguito da pool-1-thread-2
Risultato 13 da task Task7 eseguito dapool-1-thread-1
Risultato 21 da task Task8 eseguito dapool-1-thread-4
Risultato 34 da task Task9 eseguito dapool-1-thread-2
```

# GESTIONE DINAMICA: ESEMPI

**Parametri: tasks= 10, core = 3, MaxPoolSize= 4,LinkedBlockingQueue**

dimensione del pool 1

Starting task Task0 eseguito da pool-1-thread-1

Risultato 0 da task Task0 eseguito da pool-1-thread-1

dimensione del pool 2

dimensione del pool 3

Starting task Task1 eseguito da pool-1-thread-2

Risultato 1 da task Task1 eseguito da pool-1-thread-2

Starting task Task3 eseguito da pool-1-thread-2

dimensione del pool 3

Risultato 2 da task Task3 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task2 eseguito da pool-1-thread-3

Starting task Task4 eseguito da pool-1-thread-1

Starting task Task5 eseguito da pool-1-thread-2

dimensione del pool 3

# GESTIONE DINAMICA:ESEMPI

*(CONTINUA)*

```
Risultato 1 da task Task2 eseguito dapool-1-thread-3
Risultato 3 da task Task4 eseguito dapool-1-thread-1
Risultato 5 da task Task5 eseguito dapool-1-thread-2
dimensione del pool 3
Starting task Task6 eseguito da pool-1-thread-3
dimensione del pool 3
Starting task Task7 eseguito da pool-1-thread-1
Risultato 8 da task Task6 eseguito dapool-1-thread-3
dimensione del pool 3
Starting task Task8 eseguito da pool-1-thread-2
Risultato 13 da task Task7 eseguito dapool-1-thread-1
dimensione del pool 3
Starting task Task9 eseguito da pool-1-thread-3
Risultato 21 da task Task8 eseguito dapool-1-thread-2
Risultato 34 da task Task9 eseguito dapool-1-thread-3
```

# TERMINAZIONE DI THREADS

- La JVM termina la sua esecuzione quando **tutti i thread (non demoni) terminano la loro esecuzione**
- Poiché un **ExecutorService** esegue i tasks in modo asincrono rispetto alla loro sottomissione, è necessario ridefinire il concetto di terminazione, nel caso si utilizzi un **ExecutorService**
- Un **ExecutorService** mette a disposizione del programmatore diversi metodi per effettuare lo 'shutdown' dei thread del pool
- La terminazione può avvenire
  - in **modo graduale**. Si termina l'esecuzione dei tasks già sottomessi, ma non si inizia l'esecuzione di nuovi tasks
  - in **modo istantaneo**. Terminazione immediata

# TERMINAZIONE DI EXECUTORS

---

Alcuni metodi definiti dalla interfaccia **ExecutorService**

- **void shutdown( )**
- **List<Runnable> shutdownNow( )**
- **boolean isShutdown( )**
- **boolean isTerminated( )**
- **boolean awaitTermination(long timeout, TimeUnit unit)**

# TERMINAZIONE DI EXECUTORS

- **void shutdown( )**: graceful termination.
  - non accetta ulteriori task
  - i tasks sottomessi in precedenza vengono eseguiti, compresi quelli la cui esecuzione non è ancora iniziata (quelli accodati).
  - tutti i threads del pool terminano la loro esecuzione
- **List<Runnable> shutdownNow( )**: immediate termination
  - non accetta ulteriori tasks,
  - elimina dalla coda i tasks la cui esecuzione non è ancora iniziata, e li restituisce in una lista
  - **tenta di terminare** l'esecuzione dei thread che stanno eseguendo i tasks (tipicamente con **interrupt()**): quindi non può garantire la terminazione dei thread.



# CALLABLE E FUTURE

- Un oggetto di tipo **Runnable** incapsula un'attività che viene eseguita in modo asincrono
- Una **Runnable** si può considerare un **metodo asincrono**, senza **parametri** e che **non restituisce un valore di ritorno**
- Per definire un task che restituisca un valore di ritorno occorre utilizzare le seguenti interfacce:
  - **Callable**: per definire un task che **può restituire un risultato** e **sollevare eccezioni**
  - **Future**: per rappresentare il **risultato di una computazione asincrona**. Definisce metodi per controllare se la computazione è terminata, per attendere la terminazione di una computazione (eventualmente per un tempo limitato), per cancellare una computazione, .....
- La classe **FutureTask** fornisce una implementazione della interfaccia **Future**.

# L'INTERFACCIA CALLABLE

```
public interface Callable<V>
    { V call() throws Exception; }
```

L'interfaccia generica **Callable<V>**

- contiene il solo **metodo call()**, analogo al metodo `run()` della interfaccia `Runnable`, ma che può restituire un valore e sollevare eccezioni controllate
- per definire il codice `deltask`, **occorre implementare il metodo call()**
- il parametro di tipo **<V>** indica **il tipo del valore restituito**
- Esempio: **Callable<Integer>** rappresenta una elaborazione asincrona che restituisce un valore di tipo `Integer`

## CALLABLE: UN ESEMPIO

Definire un task T che calcoli una approssimazione di  $\pi$ , mediante la serie di Gregory-Leibniz (vedi lezione precedente). T restituisce il valore calcolato quando la differenza tra l'approssimazione ottenuta ed il valore di Math.PI risulta inferiore ad una soglia *precision*. T deve essere eseguito in un thread indipendente.

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

public class Pigreco implements Callable<Double>{
    private Double precision;

    public Pigreco (Double precision) {this.precision = precision;};

    public Double call ( ){
        Double result = <calcolo dell'approssimazione di  $\pi$ >
        return result;
    }
}
```

# L'INTERFACCIA FUTURE

- Per poter accedere al valore restituito dalla *Callable*, occorre costruire un oggetto di tipo **Future<V>**, che rappresenta il risultato della computazione
- Per costruire un oggetto di tipo **Future** se si gestiscono esplicitamente i threads:
  - si costruisce un oggetto della classe **FutureTask** (che implementa **Future** e **Runnable**) passando un oggetto di tipo **Callable** al costruttore
  - si passa l'oggetto **FutureTask** al costruttore del thread
- Se si usano i thread pools, si sottomette direttamente l'oggetto di tipo **Callable** al pool (con **submit()**) e si ottiene un oggetto di tipo **Future**

# L'INTERFACCIA FUTURE

```
public interface Future <V>{
```

```
    V get( ) throws ...;
```

```
    V get (long timeout, TimeUnit) throws ...;
```

```
    void cancel (boolean mayInterrupt);
```

```
    boolean isCancelled( );
```

```
    boolean isDone( ); }
```

- **get( )** si blocca fino alla terminazione del task e restituisce il valore calcolato
- È possibile definire un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una **TimeoutException**
- E' possibile cancellare il task e verificare se la computazione è terminata oppure è stata cancellata

# CALLABLE E FUTURE: UN ESEMPIO

```
import java.util.*;
import java.util.concurrent.*;
public class FutureCallable {
public static void main(String args[])
    double precision = .....;
    Pigreco pg = new Pigreco(precision);
    FutureTask <Double> task= new FutureTask <Double>(pg);
    Thread t = new Thread(task);
    t.start();
```

# CALLABLE E FUTURE: UN ESEMPIO

```
try{
    double ris = task.get(1000L, TimeUnit.MILLISECONDS);
    System.out.println("valore di isdone" + task.isDone());
    System.out.println(ris + "valore di pigreco");
}
catch(ExecutionException e) { e.printStackTrace();}
catch(TimeoutException e)
    { e.printStackTrace();
    System.out.println("tempo scaduto");
    System.out.println("valore di isdone" + task.isDone());}
catch(InterruptedException e){  } }
```

# THREAD POOLING CON CALLABLE

---

- E' possibile sottomettere un oggetto di tipo `Callable<V>` ad un thread pool mediante il metodo `submit( )`
- Il metodo restituisce direttamente un oggetto `O` di tipo `Future<V>`, per cui non è necessario costruire oggetti di tipo `FutureTask`
- E' possibile applicare all'oggetto `O` tutti i metodi visti nei lucidi precedenti



# THREAD POOLING CON CALLABLE

```
import java.util.*;
import java.util.concurrent.*;
public class futurepools {
public static void main(String args[])
    ExecutorService pool = Executors.newCachedThreadPool ( );
    double precision = .....;
    pigreco pg = new pigreco(precision);
    Future <Double> result = pool.submit(pg);
    try{ double ris = result.get(1000L, TimeUnit.MILLISECONDS);
    System.out.println(ris+"valore di pigreco");}
    catch(.....){ }.....}}
```

# PROGRAMMAZIONE DI RETE: INTRODUZIONE

## Programmazione di rete:

sviluppare applicazioni definite mediante due o più **processi** in esecuzione su **hosts diversi**, distribuiti sulla rete. I processi **cooperano** per realizzare una certa funzionalità

- **Cooperazione**: richiede lo scambio di informazioni (**comunicazione**) tra i processi
- **Comunicazione** = Utilizza **protocolli** (cioè insieme di regole che i partners della comunicazione devono seguire per poter comunicare)
- Alcuni protocolli utilizzati in INTERNET:
  - **IP (Interned Protocol)**
  - **TCP (Trasmission Control Protocol)** un protocollo connection-oriented
  - **UDP (User Datagram Protocol)** protocollo connectionless

# PROGRAMMAZIONE DI RETE: INTRODUZIONE

---

Per identificare un **processo** con cui si vuole comunicare occorre conoscere:

- la **rete** in cui si trova l'host su cui e' in esecuzione il processo
  - l'**host** all'interno della rete
  - il **processo** in esecuzione sull'host
- 
- **Identificazione della rete e dell'host**
    - definita dal protocollo **IP (Internet Protocol)**
  - **Identificazione del processo**
    - utilizza il concetto di **porta**
  - **Porta**
    - Intero da 0 a 65535

# IL PROTOCOLLO IP

---

Il Protocollo **IP (Internet Protocol)** definisce

- un **sistema di indirizzamento** per gli hosts
- la definizione della **struttura del pacchetto IP**
- un **insieme di regole** per la spedizione/ricezione dei pacchetti

Due versioni del protocollo IP sono attualmente utilizzate in Internet:

- **IPV4 (IP Versione 4)**
- **IPV6 (IP versione 6)**
  - **IPV6** introduce uno spazio di indirizzi di dimensione maggiore rispetto a **IPV4** (i cui indirizzi cominciano a scarseggiare...)
  - Di uso ancora limitato

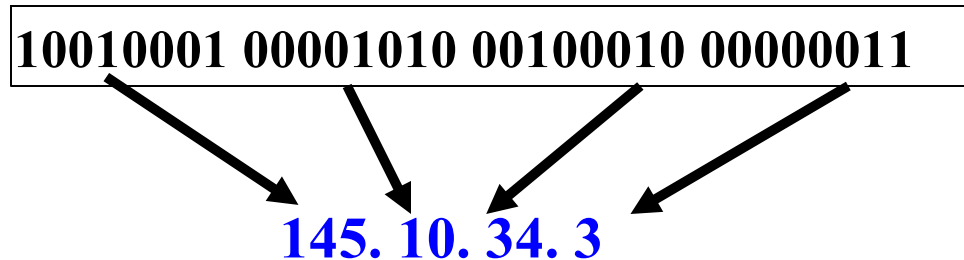
# INDIRIZZAMENTO DEGLI HOSTS

## Materiale di riferimento: Pitt, capitolo 2

- Ogni host di una rete IPV4 o IPV6 è connesso alla rete mediante **una o più interfacce**
- Ogni interfaccia è caratterizzata da un **indirizzo IP**
- **Indirizzo IP**
  - IPV4: numero rappresentato su **32 bits (4 bytes)**
  - IPV6: numero rappresentato su **128 bits (16 bytes)**
- **Multi-homed host**: un host che possiede **un insieme di interfacce** verso la rete, e quindi **un insieme di indirizzi IP** (uno per ogni interfaccia)
  - **gateway** tra sottoreti IP
  - **routers**

# INDIRIZZI IP

## Un indirizzo IPV4



- 32 bits
- Ognuno dei 4 bytes, viene interpretato come un numero decimale senza segno
- Rappresentato come sequenza di 4 valori da 0 a 255 (massimo valore su 8 bits) separati da "."

## Un indirizzo IPV6

- 128 bits
- sequenza di 16 valori da 0 a 255 separati da ":", per esempio

3:6:0:0:0:0:6:45:0:0:9:56:67:0:0:1

# INDIRIZZI IP E NOMI DI DOMINIO

- Gli **indirizzi IP** sono indispensabili per la funzionalità di instradamento dei pacchetti effettuata dai routers, ma sono poco leggibili per gli utenti della rete
- **Soluzione:** assegnare un **nome simbolico unico** ad ogni host
  - si utilizza uno spazio **di nomi gerarchico**, per esempio:  
**fujih1.cli.di.unipi.it** (host **fujih1** nel dominio **cli.di.unipi.it** )
  - livelli della gerarchia separati dal punto.
  - nomi interpretati da destra a sinistra
- **Risoluzione di indirizzi IP:** corrispondenza tra nomi ed indirizzi IP
- La **risoluzione** viene gestita da un servizio di nomi ("servizio **DNS**")  
**DNS = Domain Name System**

# INDIRIZZAMENTO A LIVELLO DI PROCESSI

- Su ogni host possono essere attivi contemporaneamente più **servizi** (es: e-mail, ftp, http,...)
  - Ogni **servizio** viene incapsulato in un diverso **processo**
  - L'indirizzamento di un processo avviene mediante una **porta**
  - Porta = intero compreso tra **0 e 65535**. Non è un **dispositivo fisico**, ma un'**astrazione** per individuare i singoli servizi (processi).
  - Porte comprese tra **1 e 1023** riservati per particolari servizi.
  - In Linux: solo i programmi in esecuzione con diritti di **root** possono ricevere dati da queste porte. Chiunque può inviare dati a queste porte.
- Esempi:**
- |          |             |          |               |
|----------|-------------|----------|---------------|
| porta 7  | <b>echo</b> | porta 21 | <b>ftp</b>    |
| porta 22 | <b>ssh</b>  | porta 32 | <b>telnet</b> |
| porta 80 | <b>HTTP</b> |          |               |
- In LINUX: controllare il file **/etc/services**



# SOCKETS

---

- Socket: astrae il concetto di 'communication endpoint'
- Individuato da un indirizzo IP e da un numero di porta
- Socket in JAVA: istanza di una di queste classi
  - Socket
  - ServerSocket
  - DatagramSocket
  - MulticastSocket

# JAVA: LA CLASSE INETADDRESS

**Materiale didattico (studiare!!!): Pritt, capitolo 2**

Classe `java.net.InetAddress` (importare `java.net`).

Gli oggetti di questa classe sono strutture con due campi

- **hostname** = una stringa che rappresenta il nome simbolico di un host
- **indirizzo IP** = un intero che rappresenta l'indirizzo IP dell'host

La classe `InetAddress`:

- non definisce costruttori
- fornisce tre *metodi statici* per costruire oggetti di tipo `InetAddress`: tutti possono lanciare una `UnknownHostException`
  - **public static** `InetAddress.getByName (String hostname)`
  - **public static** `InetAddress.getAllByName (String hostname)`
  - **public static** `InetAddress getLocalHost ( )`

# JAVA: LA CLASSE INETADDRESS

```
public static InetAddress getByName (String hostname)  
    throws UnknownHostException
```

- cerca l'indirizzo **IP** corrispondente all'host di nome **hostname** e restituisce un oggetto di tipo  
InetAddress = nome simbolico dell'host + l'indirizzo IP corrispondente
- Se **hostname** è un indirizzo **IP**, controlla solo che sia ben formato.
- in generale richiede **una interrogazione del DNS per risolvere il nome dell'host (resolution)** ⇒ il computer su cui è in esecuzione l'applicazione deve essere connesso in rete
- Può sollevare una eccezione se non riesce a risolvere il nome dell'host (ricordarsi di gestire la eccezione!)

# JAVA: LA CLASSE INETADDRESS

---

Esempio:

```
try { InetAddress address = InetAddress.getByName  
                                     ("fujim0.cli.di.unipi.it");  
    System.out.println (address);  
}  
catch (UnknownHostException e)  
{System.out.println ("Non riesco a risolvere il nome"); }
```

# JAVA: LA CLASSE INETADDRESS

**public static** InetAddress [ ] *getAllByName* (String hostname)  
**throws** *UnknownHostException*

utilizzata nel caso di hosts che posseggano piu indirizzi (es: web servers)

**public static** InetAddress *getLocalHost* ()  
**throws** *UnknownHostException*

per reperire nome simbolico ed indirizzo IP del computer locale

**Getter Methods** = Per reperire i campi di un oggetto di tipo InetAddress

**public String** *getHostName* () // può fare una **reverse resolution**

**public byte** [ ] *getAddress* ()

**public String** *getHostAddress* ()

# JAVA: LA CLASSE INETADDRESS

```
public static InetAddress getByName (String hostname)  
    throws UnKnownHostException
```

se il valore di hostname è l'indirizzo IP (una stringa che codifica la dotted form dell'indirizzo IP)

- la *getByName* *non contatta* il DNS
- il nome dell'host non viene impostato nell'oggetto *InetAddress*
- il DNS viene contattato solo quando viene *richiesto esplicitamente* il nome dell'host tramite il metodo getter *getHostName( )*
- la *getHostName( )* non solleva eccezione, se non riesce a risolvere l'indirizzo IP.

# JAVA: LA CLASSE INETADDRESS

- accesso al DNS: operazione potenzialmente molto costosa
- i metodi descritti **effettuano caching** dei nomi/indirizzi risolti.
- nella cache vengono memorizzati anche i tentativi di risoluzione non andati a buon fine (di default: per un certo numero di secondi)
- se creo un `InetAddress` per lo stesso host, il nome viene risolto con i dati nella cache (di default: per sempre)
- permanenza dati nella cache: **per default** alcuni secondi se la risoluzione non ha avuto successo, illimitato altrimenti. Problemi: indirizzi dinamici..
- **`java.security.Security.setProperty`** consente di impostare il numero di secondi in cui una entry nella cache rimane valida, tramite le proprietà

**`networkaddress.cache.ttl`** e **`networkaddress.cache.negative.ttl`**

esempio:

```
java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
```

# LA CLASSE INETADDRESS: ESEMPIO DI UTILIZZO

---

- implementazione in JAVA della utility UNIX *nslookup*
- *nslookup*
  - consente di tradurre nomi di hosts in indirizzi IP e viceversa
  - i valori da tradurre possono essere forniti in modo interattivo oppure da linea di comando
  - si entra in modalità interattiva se non si forniscono parametri da linea di comando
  - consente anche funzioni più complesse (vedere LINUX)



# LA CLASSE INETADDRESS: ESEMPIO DI UTILIZZO

```
import java.net.*;
import java.io.*;

public class HostLookUp {
    public static void main (String [ ] args) {
        if (args.length > 0) {
            for (int i=0; i<args.length; i++) {
                System.out.println (lookup(args[i]));
            }
        }
        else { /* modalita' interattiva*/ }
    }
}
```

# LA CLASSE INETADDRESS: ESEMPIO DI UTILIZZO

---

```
private static boolean isHostName (String host)
```

```
    {char[ ] ca = host.toCharArray();  
    for (int i = 0; i < ca.length; i++)  
        { if(!Character.isDigit(ca[i])) {  
            if (ca[i] != '.') return true; }  
        }  
    return false;  
    }
```

# LA CLASSE INETADDRESS: ESEMPIO DI UTILIZZO

```
private static String lookup(String host) {  
    InetAddress node;  
    try { node = InetAddress.getByName(host);  
        System.out.println(node);  
        if (isHostName(host))  
    {return node.getHostAddress( );}  
        else{  
            return node.getHostName ( );}  
        }  
    catch (UnknownHostException e)  
        {return "non ho trovato l'host";}  
}
```

# Esercizio 1

- Scrivere un programma Java `Resolve` che traduca una sequenza di nomi simbolici di host nei corrispondenti indirizzi IP.
- `Resolve` legge i nomi simbolici da un file, il cui nome è passato da linea di comando oppure richiesto all'utente.
- Si deve definire un task che estenda l'interfaccia `Callable`, e che, ricevuto come parametro un nome simbolico, provvede a tradurre il nome ritornando un `InetAddress`.
- Per ottimizzare la ricerca, si deve attivare un pool di thread che esegua i task in modo concorrente. Ogni volta che si sottomette al pool di thread un task, si ottiene un oggetto `Future<InetAddress>`, che deve essere aggiunto ad un `ArrayList`.
- Infine, si scorre l'`ArrayList`, stampando a video gli `InetAddress`.

## Esercizio 2

---

- Scrivere un programma che enumeri e stampi a video tutte le interfacce di rete del computer, usando i metodi della classe `java.net.NetworkInterface`.
- Usare il metodo statico `getNetworkInterfaces()` per ottenere una `Enumeration` di `NetworkInterface`.
- Per ogni `NetworkInterface`, stampare gli indirizzi IP associati ad essa (IPv4 e IPv6) e il nome dell'interfaccia.

## Esercizio 3

- Scrivere un programma che ricerca una parola chiave (*key*) nei file contenuti in una directory (fornita dall'utente) e nelle sue sottodirectory. Per ogni file che contiene *key*, si deve visualizzare il nome del file e il contenuto della prima riga trovata che contiene *key*.
- Creare una classe *FindKeyword* che implementa *Callable*, alla quale si può passare una directory come parametro del costruttore, e che ritorna un array di stringhe del formato  

<nome file> : <contenuto riga che contiene *key*>
- Il metodo *search* della classe *FindKeyword* implementa la ricerca di *key* all'interno di un singolo file, e ritorna una stringa formattata come sopra oppure null se *key* non compare.
- Creare un pool di thread a cui vengono sottomessi un task *FindKeyword* per ogni directory/sottodirectory, e usare gli oggetti *Future* restituiti per stampare a video i risultati ottenuti.