Lezione n.2
LPR A- Informatica Applicata
Thread Pooling
Indirizzi IP: InetAddress

02/03/2009

Laura Ricci

THREAD POOL EXECUTORS

```
import java.util.concurrent.*;
public class ThreadPoolExecutor implements ExecutorService
    {public ThreadPoolExecutor
                (int core PoolSize,
                 int maximum  PoolSize,
                 long keepAliveTime,
                 TimeUnit unit,
                 BlockingQueue <Runnable> workqueue);
                          ..........}
```

- crea un oggetto di tipo ExecutorService
- consente di personalizzare la politica di gestione del pool

THREAD POOL EXCUTORS

```
public ThreadPoolExecutor
          ( int core PoolSize,
            int maximunPoolSize,
            long keepAliveTime,
            TimeUnit unit,
            BlockingQueue<Runnable> workqueue);
```

- CorePoolSize, MaximumPoolSize, keepAliveTime controllano la gestione dei threads del pool
- Workqueue è una struttura dati necessaria per memorizzare gli eventuali tasks in attesa di esecuzione

THREAD POOLING: UN ESEMPIO

- Data una sequenza di valori interi, si vuole calcolare, per ogni valore x, il valore dell'x-esimo numero di Fibonacci
- Si definisce un oggetto che implementa l'interfaccia Runnable e che definisce il task T che effettua il calcolo
- Si attiva un ThreadPoolExecutor, definendo, al momento dell'attivazione, mediante i parametri passati al costruttore, la politica di gestione del ThreadPool.
- Si passa T all'esecutore, invocando il metodo execute

FIBONACCI TASK

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

FIBONACCI TASK

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

FIBONACCI TASK POOL

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

public class ThreadPoolTest {

        public static void main (String [] args)

        { int nTasks = Integer.parseInt(args[0]);

        //numero di tasks da eseguire

        int core       = Integer.parseInt(args[1]);

        // dimensione del "nucleo"  pool

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

        // massima dimensione del pool
```

FIBONACCI TASK POOL

```
ThreadPoolExecutor tpe =
 new ThreadPoolExecutor  (core,  maxPoolSize, 50000L,
   TimeUnit.MILLISECONDS, new LinkedBlockingQueue <Runnable>( ) );
 Task [  ] tasks = new Task[nTasks];
 for (int i=0; i< nTasks; i++){
  tasks[i]= new Task(i, "Task"+i);
  tpe.execute(tasks[i]);
  System.out.println("dimensione del pool     "+  tpe.getPoolSize(
  };
  tpe.shutdown();    }    }
```

THREAD POOL: GESTIONE DINAMICA

- Core: dimensione minima del pool: il supporto crea un pool di dimensione Core.
  - È possibile allocare core threads al momento della creazione del pool mediante il metodo prestartAllCoreThreads( ). I threads creati rimangono inattivi in attesa di tasks da eseguire
  - oppure i threadspossono 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 pool prima possibile
- MaxPoolSize:dimensione massima del pool

THREAD POOL: GESTIONE DINAMICA

- Se sono in esecuzione tutti i core threads, un nuovo task sottomesso viene inserito in una coda *C*.
  - *C* deve essere una istanza di BlockingQueue
  - *C* viene passata al momento della costruzione del threadpool (ultimo parametro del costruttore)
  - E' possibile scegliere diversi tipi di coda (tipi derivati da BlockingQueue). Il tipo di coda scelto influisce sullo scheduling.
- I task vengono poi prelevati da *C* e inviati ai threads che si rendono disponibili
- Solo quando *C* risulta piena si crea un nuovo thread attivando così k threads, core ≤ 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 thread 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 <= core: il thread si mette in attesa di nuovi tasks da eseguire. L'attesa è indefinita.
- Se k > core, si considera il *timeout T* definito al momento della costruzione del thread pool
  - se nessun thread viene sottomesso entro T, TH termina la sua esecuzione, riducendo così il numero di threads del pool
  - Timeout: occorre definire
    - un valore (es: 50000) 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 è se è possibile creare un nuovo thread (numero di threads ≤ 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 può superare core
- ArrayBlockingQueue: dimensione limitata, stabilita dal programmatore

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 dapool-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 dapool-1-thread-2
Starting task   Task4   eseguito da    pool-1-thread-4
Risultato   2    da task     Task3   eseguito dapool-1-thread-1
Risultato   1    da task     Task2   eseguito dapool-1-thread-3
java.util.concurrent.RejectedExecutionException
Risultato   3    da task     Task4   eseguito dapool-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 pool    3
Risultato   0    da task     Task0   eseguito dapool-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 dapool-1-thread-1
Risultato   1    da task     Task1   eseguito dapool-1-thread-2
Starting task   Task3   eseguito da    pool-1-thread-3
dimensione del pool    3
Risultato   2    da task     Task3   eseguito dapool-1-thread-3
dimensione del pool    3
```

GESTIONE DINAMICA: ESEMPI

```
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 dapool-1-thread-2
dimensione del pool    3
Starting task   Task6   eseguito da    pool-1-thread-1
Risultato   5    da task     Task5   eseguito dapool-1-thread-3
dimensione del pool    3
Starting task   Task7   eseguito da    pool-1-thread-2
Risultato   8    da task     Task6   eseguito dapool-1-thread-1
Risultato   13    da task     Task7   eseguito dapool-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 dapool-1-thread-1
dimensione del pool    2
Starting task   Task1   eseguito da    pool-1-thread-2
Risultato   1    da task     Task1   eseguito dapool-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 dapool-1-thread-3
Risultato   2    da task     Task3   eseguito dapool-1-thread-2
dimensione del pool    4
```

GESTIONE DINAMICA:ESEMPI

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

```
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 dapool-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 dapool-1-thread-2
Starting task   Task3   eseguito da    pool-1-thread-2
dimensione del pool    3
Risultato   2    da task     Task3   eseguito dapool-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

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

```
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 Executor Service
- Un Executor Service mette a disposizione del programmatore diversi metodi per effettuare lo 'shutdown' dei thrads 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

- `shutdown( )` graceful termination.
  - nessun task viene accettato dopo che la shutdown() è stata invocata.
  - tutti 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
- `shutdowNow( )` immediate termination
  - non accetta ulteriori tasks,
  - elimina i tasks la cui esecuzione non è ancora iniziata
  - restituisce una lista dei tasks che sono stati eliminati dalla coda
  - tenta di terminare l'esecuzione dei thread che stanno eseguendo i tasks.

TERMINAZIONE DI EXECUTORS

ShutdownNow( )

- implementazion best effort
- non garantisce la terminazione immediata dei threads del pool
- implementazione generalmente utilizzata: invio di una interruzione ai thread in esecuzione nel pool
- se un thread non risponde all'interruzione non termina
- infatti, se sottometto il seguente task al pool

```
public class ThreadLoop implements Runnable {
    public ThreadLoop(){};
    public void run( ){while (true)  { }  }}
```

e poi effettuo la shutdown(), posso osservare che il programma non termina

ATTENDERE LA TERMINAZIONE DI UN THREAD: METODO JOIN

- Un thread J può invocare il motodo 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");}
      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);
 Joiner waitassonnato = new Joiner("WaitforAssonnato",assonnato);
 Joiner waitstanco = new Joiner("WaitforStanco",stanco);
 stanco.interrupt( );} }
```

Output: Stanco è stato interrotto
WaitforStanco join completed
Assonnato è stato svegliato normalmente
WaitforAssonnato join completed

THREAD POOL: POLITICHE DI SATURAZIONE

Politiche di saturazione: quando viene sottomesso un task T e la coda è piena, possono essere adottate le seguenti politiche

- abort T viene scartato e viene sollevata un'eccezione
- discard policy rifiuta T, ma non solleva alcun tipo di eccezione
- discard oldest scarta il primo task della coda (quello che avrebbe dovuto essere eseguito successivamente) e inserisce T in coda
- caller-runs
  - non scarta il task e non solleva eccezioni.
  - cerca di rallentare (throttling, to throttle=strozzare) il flusso dei tasks restituendo alcuni task al chiamante per l'esecuzione. Il task non viene eseguito in un thread del pool, ma nel thread che ha invocato la execute

THREAD POOL: POLITICHE DI SATURAZIONE

```
import java.util.concurrent.*;
public class prova {
public static void main (String args[ ])
{ ThreadPoolExecutor executor=
      new ThreadPoolExecutor(10,11,0L,TimeUnit.MILLISECONDS,
                        new LinkedBlockingQueue<Runnable>(100));
  executor.setRejectedExecutionHandler
         (new ThreadPoolExecutor.CallerRunsPolicy());}}
```

THREAD POOL: POLITICHE DI SATURAZIONE

- Politica di saturazione:
  - utilizzata nel caso si utilizzino thread pool con code di dimensione limitata
  - indica le azioni che devono essere effettuate nel caso in cui il pool è saturo, cioè la coda è piena ed i threads del pool sono tutti attivi
- Strategia di default: abort il task viene rifiutato e viene sollevata una RejectedExecutionException
- E' possibile definire una politica 'ad hoc' mediante un Rejected Execution Handler
- JAVA mette a disposizione diversi Rejected Execution Handler, ognuno dei quali implementa una diversa politica di saturazione
- Selezione della politica: setRejectedExecutionHandler

TASK ASINCRONI

- Un oggetto di tipo Runnable incapsula un'attività che viene eseguita in modo asincrono
- Il metodo run() di una Runnable ha un comportamento diverso rispetto ai metodi classici perchè
  - Non è possibile passare parametri al metodo run()
  - Non è possibile ricevere un valore di ritorno. A differenza dei metodi classic, non è possibileprevedere quando l'attività svolta dal metodo termina, poichè tale attività è asincrona
  - Il metodo non può propagare eccezioni alchiamante. Poichè l'attività svolta dal metodo è asincrona, è complesso definire un meccanismo di propagazione delle eccezioni al chiamante
- Queste limitazioni sono state superate a partire da JAVA 5 con l'introduzione degli oggetti di tipo Callable e Future

CALLABLE E FUTURE

- 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 Callable

- contiene il solo metodo call, analogo al metodo run( ) della interfaccia Runnable
- per definire il codice del task, occorre implementare il metodo call
- a differenza del metodo run( ), il metodo call( ) può restituire un valore e sollevare eccezioni
- 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 Tche calcoli una approssimazione di π, 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 precisione. T deve essere eseguito in un thread.

```
import java.util.concurrent.*;
public class pigreco implements Callable <Double>
     {private Double precision;
      public pigreco (Double precision)
        {this.precision=precision;};
      public Double call ( ) throws Exception
        {Double result = <calcolo dell'approssimazione di π>
         return result }}
```

L'INTERFACCIA FUTURE

- Per poter accedere al valore restituito dalla Callable, occorre costruire un oggetto di tipo <Future>, che rappresenta il risultato della computazione
- Per costruire un oggetto di tipo <Future>
  - se i threads sono gestiti esplicitamente:
    - si costruisce un oggetto Future a partire da un oggetto di tipo <Callable> mediante i costruttori della classe FutureTask
    - si passa l'oggetto Future al costruttore del thread
  - se si usano i thread pools, si sottomette direttamente l'oggetto di tipo Callable al pool 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( );
```

pg è un oggetto di tipo callable( ) su cui si costruisce l'oggetto di tipo Future

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 is done"+task.isDone());}
catch(CancellationException e){e.printStackTrace(); }    }
      }
```

THREAD POOLING CON CALLABLE

- E' possibile sottomettere un oggetto di tipo Callable ad un thread pool mediante il metodo submit
- Il metodo restituisce direttamente un oggetto O di tipo Future, 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 (Exception e) {....}
```

UN TASK CHE RESTITUISCE UN RISULTATO

```
import java.util.concurrent.*;
public class wordlength implements Callable <Integer> {

    private String word;

    public wordlength(String word) {
                this.word = word;    }

   public Integer call() {
            return Integer.valueOf(word.length());
                        }
                                                    }
```

UN TASK CHE RESTITUISCE UN RISULTATO

```
import java.util.concurrent.*;  import java.util.*;
public class computesumlenghts {
   public static void main(String args[]) throws Exception {
     ExecutorService pool = Executors.newFixedThreadPool(3);
     Set<Future<Integer>> set = new HsshSet<Future<Integer>>();
     for (String word: args) {
        Callable<Integer> callable = new wordlength(word);
        Future<Integer> future = pool.submit(callable);
        set.add(future);}
   int sum = 0;
   for (Future<Integer> future : set) { sum += future.get(); }
   System.out.printf("The sum of lengths is", sum);} }
```

CANCELLAZIONE DI TASKS

- Un task sottomesso al pool può essere cancellato, eliminando il rispettivo oggetto Future
- se si tenta di reperire in seguito il risultato dell'elaborazione di quel task viene sollevata una CancellationException

```
public class maincallable {
public static void main(String args[]) throws Exception {
ExecutorService pool = Executors.newFixedThreadPool(3);
Set<Future<Integer>> set = new HashSet<Future<Integer>>();
for (String word: args) {
    Callable<Integer> callable = new futurelength(word);
    Future<Integer> future = pool.submit(callable);
    set.add(future);
    if (word.equals("Juve")){future.cancel(true);}; }
    catch(.....) {}}}
```

CANCELLAZIONE DI TASKS

(prosegue dalla pagina precedente)....

```
int sum = 0;
for (Future<Integer> future : set) {
try{  sum += future.get();
   }catch(CancellationException e)
   {System.out.println("task Cancellato"+future.isCancelled();}
System.out.print(sum);
pool.shutdown();
                                          }}}
```

se viene inserita la stringa 'Juve', viene sollevata un'eccezione, nel momento in cui si tenta di accedere all'oggetto future relativo.

CALLABLE: PROPAGAZIONE ECCEZIONI

- Il metodo call può sollevare eccezioni
- Le eccezioni sollevate possono essere propagate al chiamante e poi gestite da esso

```
import java.util.concurrent.*;
public class wordlength implements Callable <Integer>
{ private String word;
  public wordlength(String word) { this.word =word;    }
  public Integer call() throws Exception {
    if (word.equals("Juve")){throw new RuntimeException("...");}
   return Integer.valueOf(word.length());
   }}
```

CALLABLE:PROPAGAZIONE DI ECCEZIONI

```
.............

int sum = 0;

for (Future<Integer> future : set) {

    try{

        sum += future.get();}

    catch(ExecutionException e)

          {System.out.println("Sollevata Eccezione"+e);}

.............
```

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 tra i processi
- Comunicazione = Utilizza protocolli (= insieme di regole che i partners della comunicazione devono seguire per poter comunicare)
- Alcuni protocolli utilizzati in INTERNET:
  - 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

- la rete all'interno della quale 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

Versioni del protocollo IP sono attualmente utilizzate in Internet

- IPV4 (IP Versione 4)
- IPV6 (IP versione 6)
  - Introduce uno spazio di indirizzi di dimensione maggiore rispetto ad IPV4
  - Di uso ancora limitato

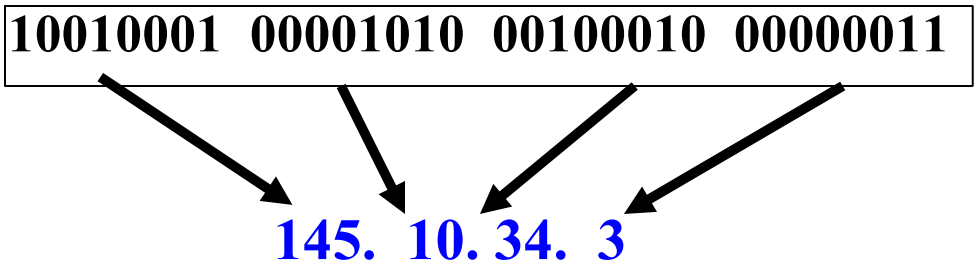
INDIRIZZAMENTO DEGLI HOSTS

Materiale da studiare: 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)
- se un host, ad esempio un router, presenta più di una interfaccia sulla rete ⇒ più indirizzi IP per lo stesso host, uno per ogni interfaccia
- Multi-homed hosts: un host che possiede un insieme di interfacce verso la rete, e quindi da un insime di indirizzi IP
  - gateway tra sottoreti IP
  - routers

INDIRIZZI IP

Un indirizzo IPV4



- Ognuno dei 4 bytes, viene interpretato come un numero decimale senza segno
- Ogni valore varia da 0 a 255 (massimo valore su 8 bits)

Un indirizzo IPV6

- 128 bits
- rappresentato come una sequenza di 16 cifre separate da due punti

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 semplificano l'eleborazione effettuata dai routers, ma sono poco leggibili per gli utenti della rete

- Soluzione: assegnare un nome simbolico unico ad ogni host della rete
  - si utilizza uno spazio di nomi gerarchico

    esempio: fujih0.cli.di.unipi.it (host fuji presente nell'aula H alla postazione 0, nel dominio cli.di.unipi.it)
  - livelli della gerarchia separati dal punto.
  - nomi interpretati da destra a sinistra (diverso dalle gerarchia di LINUX)
- Corrispondenza tra nomi ed indirizzi IP = Risoluzione di indirizzi IP
- La risoluzione viene gestita da un servizio di nomi

  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 1 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.
- Linux :solo i programmi in esecuzione su root possono ricevere dati da queste porte. Chiunque può inviare dati a queste porte.

  Esempio: porta 7 echo
  porta 22 ssh
  porta 80 HTTP
- 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
  - **public static** InetAddress.*getByName* (*String* hostnme) *throws UnKnownHostException*
  - **public static** InetAddress.*getAllByName*(*String* hostname) *throws UnKnownHostException*
  - **public static** InetAddress *getLocalHost*( ) *throws UnKnownHostException*

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

  (reverse resolution: può essere utilizzata anche per tradurre un indirizzo IP nel nome simbolico corrispondente)
- in generale richiede una interrogazione del DNS per risolvere il nome dell'host ⇒ 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 su cui è in esecuzione l'applicazione

Getter Methods = Per reperire i campi di un oggetto di tipo InetAddress (non effettuano collegamenti con il DNS ⇒ non sollevano eccezioni)

```
public String getHostName ( )
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, tempo illimitato altrimenti.
- problemi: indirizzi dinamici.....
- soluzione: java.security.Security.setProperty consente di impostare il numero di secondi in cui una entrata nella cache rimane valida, tramite le proprietà

```
networkaddress.cache.ttl
networkaddress.cache.negative.ttl
```

```
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

```
public class nslookup {
    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";}
                                                    }
```

WEBLOOKUP: ELABORAZIONE DI INDIRIZZI IP

Scrivere un programma JAVA, WebLookUp che traduca una sequenza di nomi simbolici di hosts nei corrispondenti indirizzi IP. WebLookUp legge nomi simbolici da un file , LogFile. Si deve definire un TasK, che, ricevuto come parametro un nome simbolico, provvede ad interrogare il DNS per la traduzione del nome.

Poichè l'esecuzione sequenziale di queste interrogazioni al DNS può provocare una notevole degradazione delle prestazioni del programma, si deve attivare un pool di thread che esegua i tasks in modo concorrente.

Si utilizzino i threadpool di JAVA implementando diverse politiche