ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

9: Threads

THREADS

Approfondiamo la struttura del modello dei Thread in Java e quali operazioni si possono fare su di essi.

AVVIO E ISPEZIONE

```java
/**
* Allocates a new Thread object so that it has target as
* its run object, has the specified name as its name, and
* belongs to the thread group referred to by group, and
* has the specified stack size.
*
*/
public Thread(ThreadGroup group,
  Runnable target,
  String name,
  long stackSize)
```

```
/**
* Causes this thread to begin execution; the Java Virtual
* Machine calls the run method of this thread.
*
*/
void start()
```

```java
/**
* Returns this thread's name.
*/
public String getName()
```

```java
/**
* Tests if this thread is alive.
*/
public boolean isAlive()
```

```java
/**
* If this thread was constructed using a separate Runnable
* run object, then that Runnable object's run method is
* called; otherwise, this method does nothing and returns.
*/
public void run()
```

```java
/**
* Returns a reference to the currently executing thread
* object.
*
*/
public static Thread currentThread()
```

```java
/**
* Causes the currently executing thread to sleep
* (temporarily cease execution) for the specified number of
* milliseconds, subject to the precision and accuracy of
* system timers and schedulers
*
* @param millis the length of time to sleep in milliseconds
*
*/
public static void sleep(long millis)
  throws InterruptedException
```

ESEMPI

it.unipd.app2020.threads.ThreadObserver

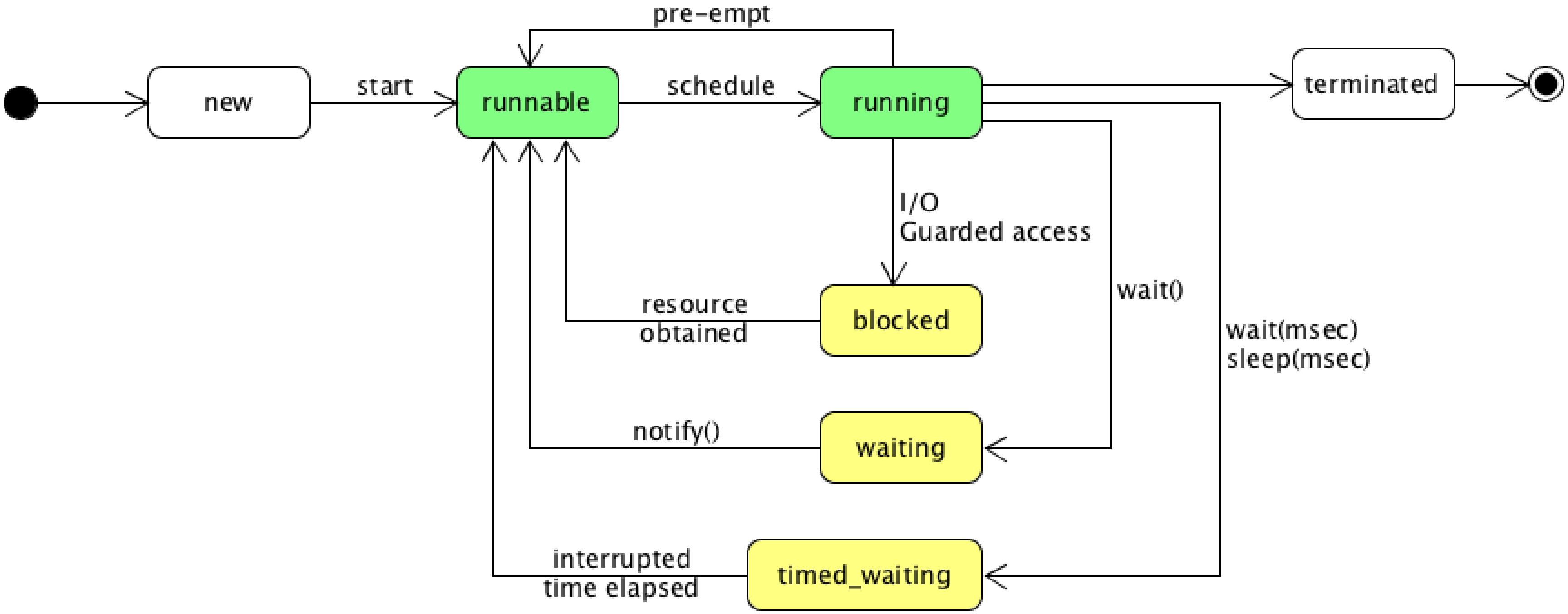
```java
final Thread observer = new Thread(() -> {
  out.println("(Start) Target live: " + tgt.isAlive());
  for (int i = 0; i < 10; i++) {
    try {
      Thread.sleep(100L);
      out.println("Target live: " + tgt.isAlive());
    } catch (InterruptedException e) {
      out.println(" Observer Interrupted");
      e.printStackTrace();
    }
  }
  out.println("(End) Target live: " + tgt.isAlive());
});
```

```
public static void main(String[] args) {
  final Thread tgt = new ThreadSupplier(800L).get();

  // ...observer...

  observer.start();
  tgt.start();
}
```

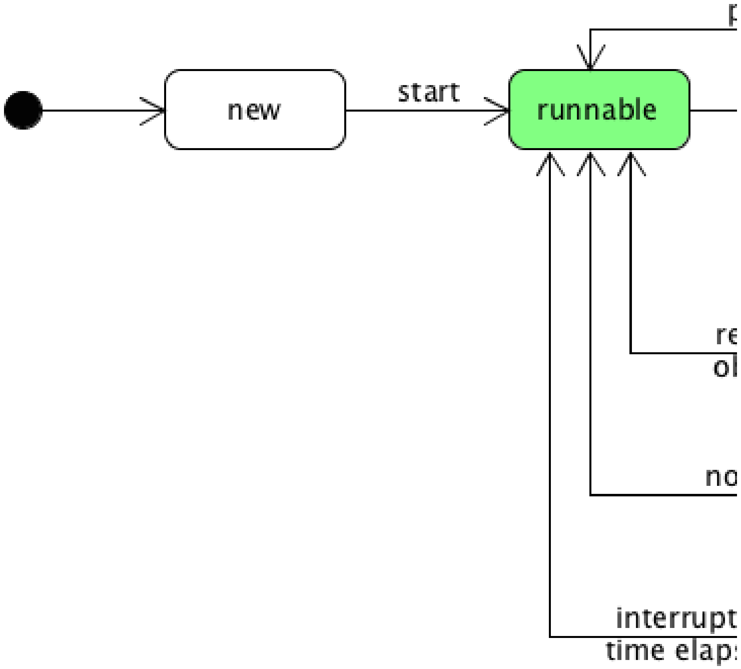
STATO DEL THREAD



java.lang.Thread.State

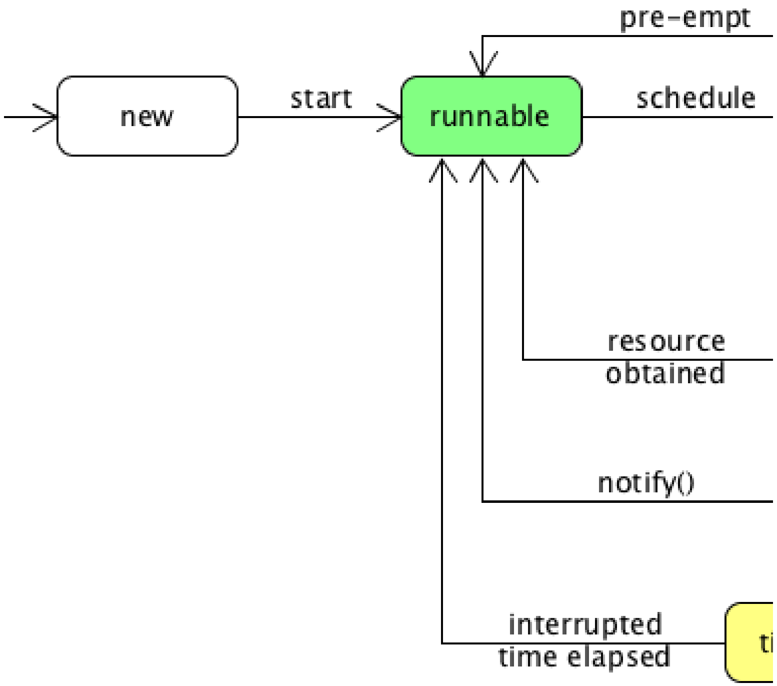
NEW

Il Thread è stato creato.



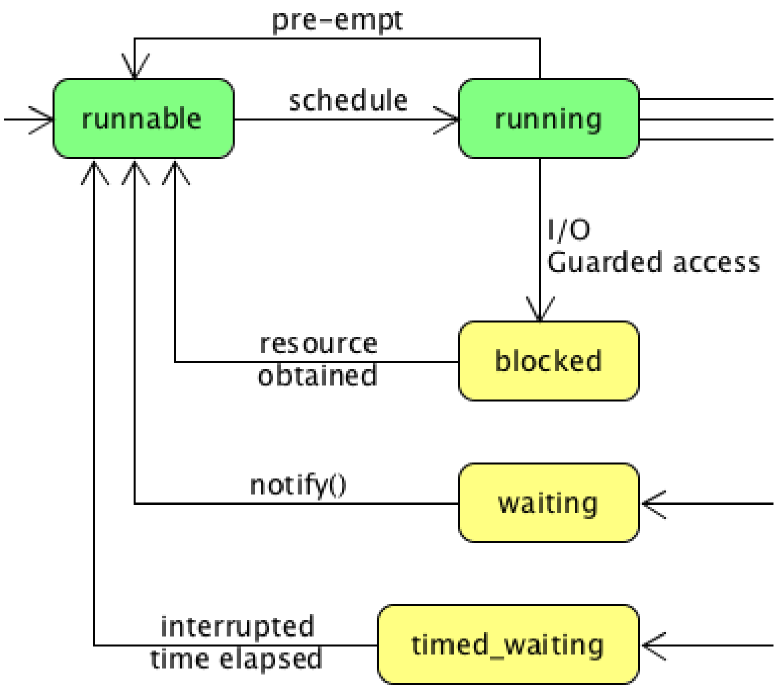
RUNNABLE

E' stato richiamato start(). Il metodo run() del Thread o del Runnable contenuto può essere messo in esecuzione.

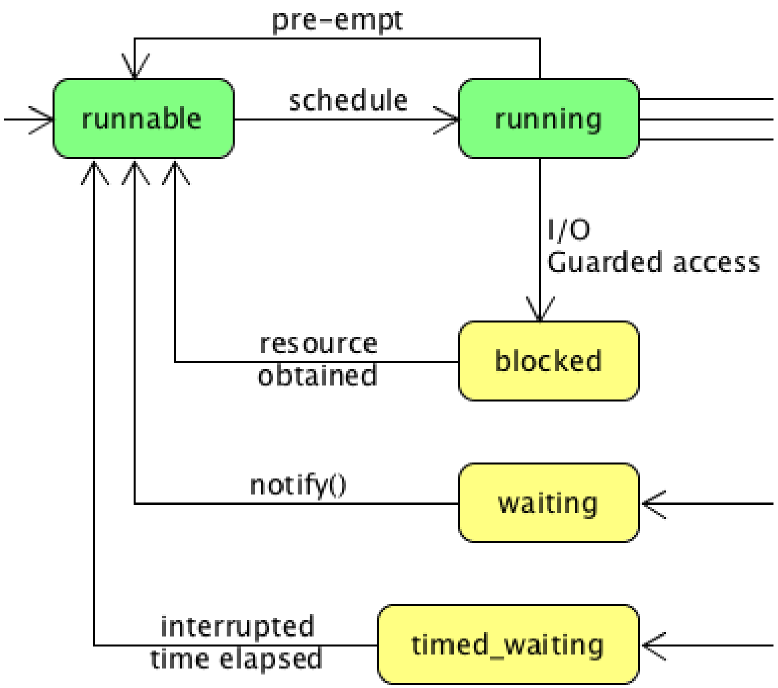


RUNNING

Il Thread è effettivamente in esecuzione, ha a disposizione la CPU finché non gli viene sottratta o passa ad altro stato.

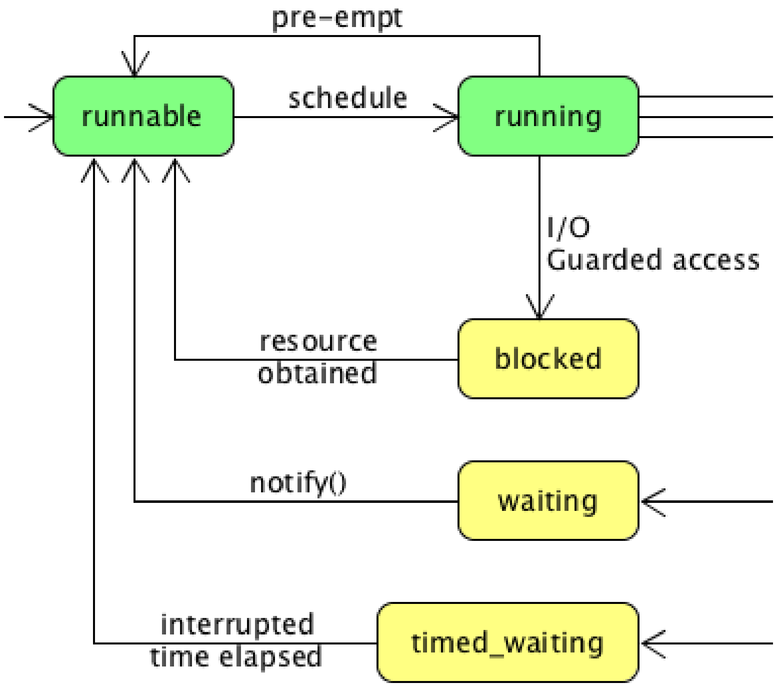


BLOCKED



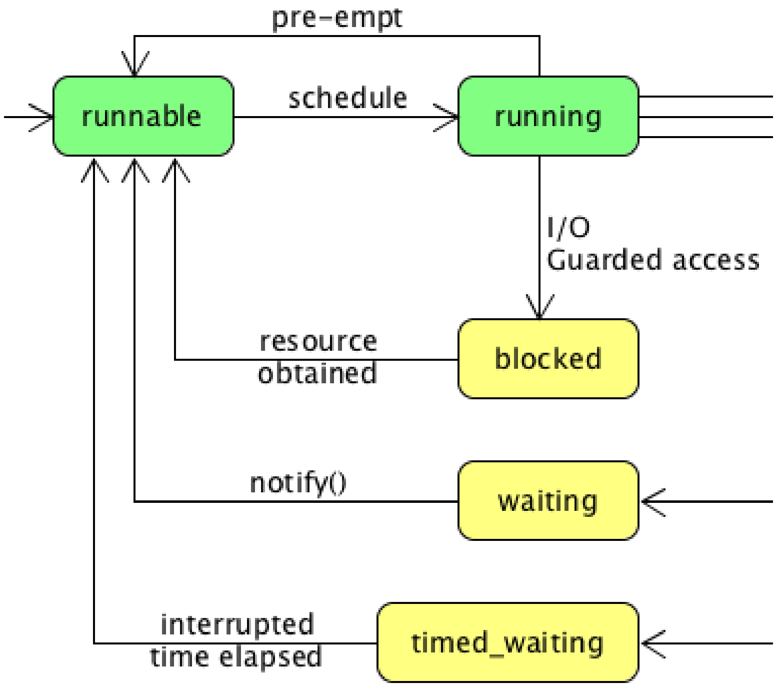
Il Thread ha richiesto accesso ad una risorsa monitorata (per es. un canale di I/O) e sta aspettando la disponibilità di dati.

WAITING



Il Thread si è posto in attesa di una risorsa protetta da un lock chiamando wait(object) e sta aspettando il suo turno.

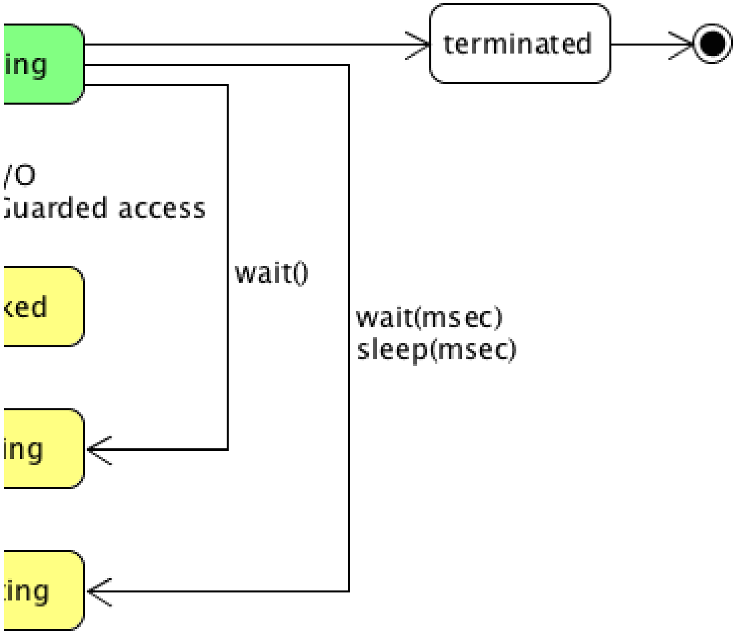
TIMED_WAITING



Il Thread si è posto in attesa di un determinato periodo di tempo (per es. con sleep(millis)) scaduto il quale ritornerà RUNNABLE.

TERMINATED

Il metodo run() è completato (correttamente o meno) ed il Thread ha concluso il lavoro.



INTERRUZIONI ED ECCEZIONI

```
/**
 * Interrupts this thread.
 *
 */
public void interrupt()
```

it.unipd.app2020.threads.ThreadInterrup

```java
@Override public void run() {
  out.println("Target Thread alive: " + tgt.isAlive());
  for (int i = 0; i < 4; i++) {
    try {
      Thread.sleep(1000L);
      tgt.interrupt();
      out.println("Target interrupted.");
    } catch (InterruptedException e) {
      out.println("Interrupter Interrupted");
      e.printStackTrace();
    }
  }
  out.println("Target Thread alive: " + tgt.isAlive());
}
```

```java
public static void main(String[] args) {
  final Thread tgt = new ThreadSupplier(2000L).get();
  final Thread interrupter = new Thread(new Interrupter(tgt));

  interrupter.start();
  tgt.start();
}
```

```java
/**
 * Set the handler invoked when this thread abruptly
 * terminates due to an uncaught exception.
 */
public void setUncaughtExceptionHandler(
    Thread.UncaughtExceptionHandler eh)
```

it.unipd.app2020.threads.RethrowingThre

```java
@Override
public Thread get() {
  return new Thread(() -> {
    String s = Thread.currentThread().getName();
    long t = waitTime.get();
    out.println(s + " will wait for " + t + " ms.");
    try {
      Thread.sleep(t);
      out.println(s + " is done wating for " + t + " ms." );
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
  });
}
```

```java
final Thread tgt=new RethrowingThreadBuilder(2000L).get();
tgt.setUncaughtExceptionHandler((Thread t, Throwable e) -> {
  out.println("Thread " + t.getName() +
    " has thrown:\n" + e.getClass() + ": " + e.getMessage());
  });

final Thread interrupter = new Thread(new Interrupter(tgt));
interrupter.start();
tgt.start();
```

EXECUTORS

Creare un nuovo Thread per ogni operazione da fare può velocemente diventare costoso.

L'amministrazione dei Thread impegnati, allo stesso modo, si complica al crescere del numero degli oggetti.

La soluzione è cedere parte del controllo al sistema, in cambio di maggiore semplicità ed efficienza.

```java
/**
 * An object that executes submitted Runnable tasks.
 * This interface provides a way of decoupling task submission
 * from the mechanics of how each task will be run, including
 * details of thread use, scheduling, etc.
 *
 */
public interface Executor
```

```
/**
 * Executes the given command at some time in the future.
 * The command may execute in a new thread, in a pooled
 * thread, or in the calling thread, at the discretion
 * of the Executor implementation.
 *
 * @param command the runnable task
 *
 */
void execute(Runnable command)
```

it.unipd.app2020.threads.FixedThreadPoo

```java
Executor executor = Executors.newFixedThreadPool(4);

var threads = Stream.generate(new ThreadSupplier());
out.println("Scheduling runnables");
threads.limit(10).forEach((r) -> executor.execute(r));
out.println("Done scheduling.");
```

it.unipd.app2020.threads.SingleThreadPo

```
Executor executor = Executors.newSingleThreadExecutor();

var threads = Stream.generate(new ThreadSupplier());
out.println("Scheduling runnables");
threads.limit(10).forEach((r) -> executor.execute(r));
out.println("Done scheduling.");
```

Esempi di esecutori:

Tipo	Funzionamento
CachedThreadPool	Riusa thread già creati, ne crea nuovi se necessario
FixedThreadPool	Riusa un insieme di thread di dimensione fissa

Esempi di esecutori:

Tipo	Funzionamento
ScheduledThreadPool	Esegue i compiti con una temporizzazione
SingleThreadExecutor	Usa un solo thread per tutti i compiti

Esempi di esecutori:

Tipo	Funzionamento
ForkJoinPool	Punta ad usare tutti i processori disponibili. Specializzato per il framework di fork/join

CALLABLES

Finora abbiamo usato come lavoro da eseguire dei `Runnable`, cioè dei blocchi privi di risultato.

L'interfaccia `Callable` ci permette di definire dei compiti che producono un risultato.

```java
/**
 * A task that returns a result and may throw an exception.
 */
@FunctionalInterface
public interface Callable < V > {
  /**
   * Computes a result, or throws an exception if unable
   * to do so.
   *
   * @return computed result
   * @throws Exception - if unable to compute a result
   */
  V call() throws Exception;
}
```

Un semplice `Executor` non esegue `Callable`: è necessario scegliere un `ExecutorService`, che espone i metodi necessari.

```
/**
 * An Executor that provides methods to manage termination
 * and methods that can produce a Future for tracking
 * progress of one or more asynchronous tasks.
 *
 */
public interface ExecutorService
  extends Executor
```

```java
/**
 * Submits a value-returning task for execution and
 * returns a Future representing the pending results
 * of the task.
 *
 * @param T - the type of the task's result
 * @param task - the task to submit
 * @return a Future representing pending completion
 * of the task
 *
 */
 < T > Future< T > submit(Callable< T > task)
```

Un `Future` è rappresenta un calcolo che prima o poi ritornerà un valore. E' possibile verificare se il calcolo è stato completato, ottenere il valore risultante, o controllare se sia ancora in corso.

```
/**
* A Future represents the result of an asynchronous
* computation. Methods are provided to check if the
* computation is complete, to wait for its completion,
* and to retrieve the result of the computation.
*
*/
public interface Future< V >
```

```
/**
 * Waits if necessary for the computation to complete,
 * and then retrieves its result.
 *
*/
T get()
```

```java
/**
 * Returns true if this task completed.
 *
 */
boolean isDone()
```

it.unipd.app2020.ScheduledFuture

```
ThreadPoolExecutor executor =
  (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
Supplier< Callable< Integer > > supplier =
  new FactorialBuilder();
List< Future< Integer > > futures =
  new ArrayList< Future< Integer > >();
```

```
for (int i = 0; i < 10; i++)
  futures.add(executor.submit(supplier.get()));
while (executor.getCompletedTaskCount() < futures.size()) {
  out.printf("Completed Tasks: %d: %s\n",
    executor.getCompletedTaskCount(),
    format(futures));
  TimeUnit.MILLISECONDS.sleep(50);
}
```

Con a disposizione una lista di `Callables`, un `ExecutorService` ci permette di:

- ottenere un risultato di un `Future` che ha terminato (non necessariamente il primo, ma probabilmente uno dei primi)
- ottenere una lista di `Future` nel momento in cui sono tutti completati

```java
/**
 * Executes the given tasks, returning the
 * result of one that has completed successfully
 * (i.e. without throwing an exception), if any do.
 *
 */
< T > T invokeAny(
  Collection < ? extends Callable< T > > tasks)
```

```
/**
 * Executes the given tasks, returning a list of Futures
 * holding their status and results when all complete.
 * Future.isDone() is true for each element of the
 * returned list.
 */
< T > List< Future< T > >
  invokeAll(Collection< ? extends Callable< T > > tasks)
```

it.unipd.app2020.AllFutures

```java
ThreadPoolExecutor executor =
  (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
var supplier = new FactorialBuilder();
var callables = new ArrayList< Callable< Integer > >();
for (int i = 0; i < 10; i++)
  callables.add(supplier.get());

out.println("Scheduling computations");
var futures = executor.invokeAll(callables);
out.println("Done scheduling.");
```

it.unipd.app2020.AnyFutures

```
ThreadPoolExecutor executor =
  (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
var supplier = new FactorialBuilder();
var callables = new ArrayList< Callable< Integer > >();
for (int i = 0; i < 10; i++)
  callables.add(supplier.get());

out.println("Scheduling computations");
var result = executor.invokeAny(callables);
out.println("Done invoking: " + result);
```

Un `ExecutorService` rimane sempre in attesa di nuovi compiti da eseguire, impedendo alla JVM di terminare.

Per permettere alla JVM di fermarsi bisogna esplicitamente fermare il servizio.

```
/**
 * Initiates an orderly shutdown in which previously
 * submitted tasks are executed, but no new tasks will
 * be accepted.
 *
 */
void shutdown()
```

```java
/**
 * Blocks until all tasks have completed execution after a
 * shutdown request, or the timeout occurs, or the current
 * thread is interrupted, whichever happens first.
 *
 */
boolean awaitTermination(long timeout, TimeUnit unit)
```

```
/**
 * Returns true if all tasks have completed following
 * shut down.
 *
 */
boolean isTerminated()
```

```java
/**
 * Attempts to stop all actively executing tasks, halts
 * the processing of waiting tasks, and returns a list
 * of the tasks that were awaiting execution.
 *
 */
List< Runnable > shutdownNow()
```