# PCD2018 - 14

# THREADS

Metodi e applicazioni

# 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)
```

Speaker notes

Questo è la forma più completa del costruttore di un Thread. Il parametro `group` può essere soggetto a restrizioni di sicurezza, perché è usato da alcuni meccanismi di abilitazione di capabilities. Il supporto del parametro `stackSize` è a discrezione della specifica implementazione della JVM.

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

Speaker notes

come abbiamo già visto, questo metodo ritorna immediatamente al chiamante, e contemporaneamente il metodo `run()` dell'oggetto (o il `Runnable` passato alla costruzione) viene avviato su di una linea di esecuzione separata.

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

Speaker notes

per gestire una popolazione di thread, e per rendere più facili da leggere i log, ogni thread ha un nome, che viene impostato o dal costruttore o automaticamente.

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

Speaker notes

Come vedremo, "alive" è una semplificazione: un thread può trovarsi in diversi stati a seconda del momento in cui si trova la sua esecuzione.

```
/**
 * 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()
```

Speaker notes

al contrario di `start()`, questo metodo non ritorna: esegue il contenuto dell'oggetto thread all'interno del chiamante. Di solito, non è questo quello che si desidera.

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

Speaker notes

questo metodo consente di ottenere il riferimento al thread di esecuzione per effettuare operazioni introspettive.

```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
```

Speaker notes

negli esempi, è un modo molto comodo per rendere più visibili (non instantanee) le interazioni fra i thread.

# ESEMPI

# pcd2019.threads.ThreadObserver

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

Speaker notes

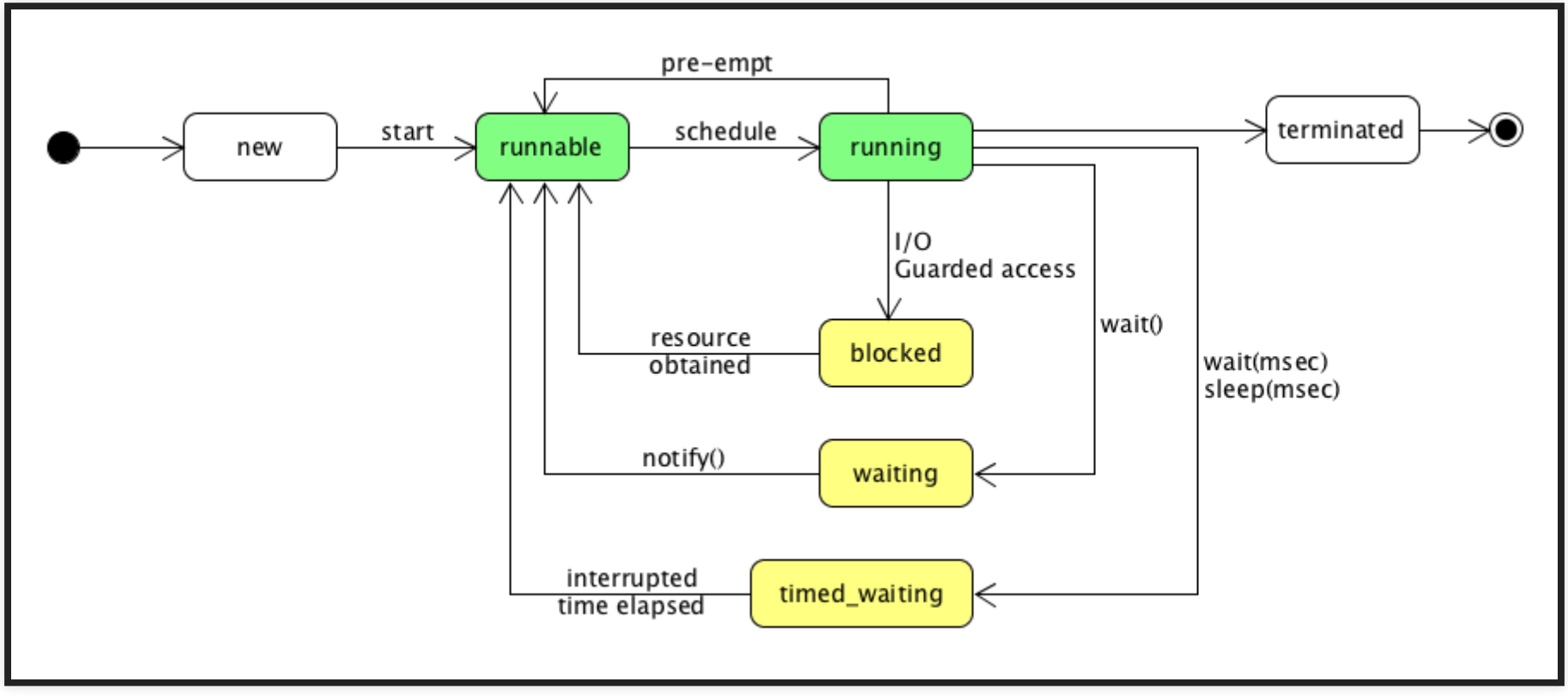
costruiamo un thread che osserva lo stato di un altro thread. Lo fa per 10 volte, ogni 100ms.

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

  // ...observer...

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

Speaker notes

Il thread osservato attende 800ms prima di uscire, quindi dovremmo notare il campbio di stato. Ancora osserviamo che la JVM non termina dopo l'avvio dei Thread.

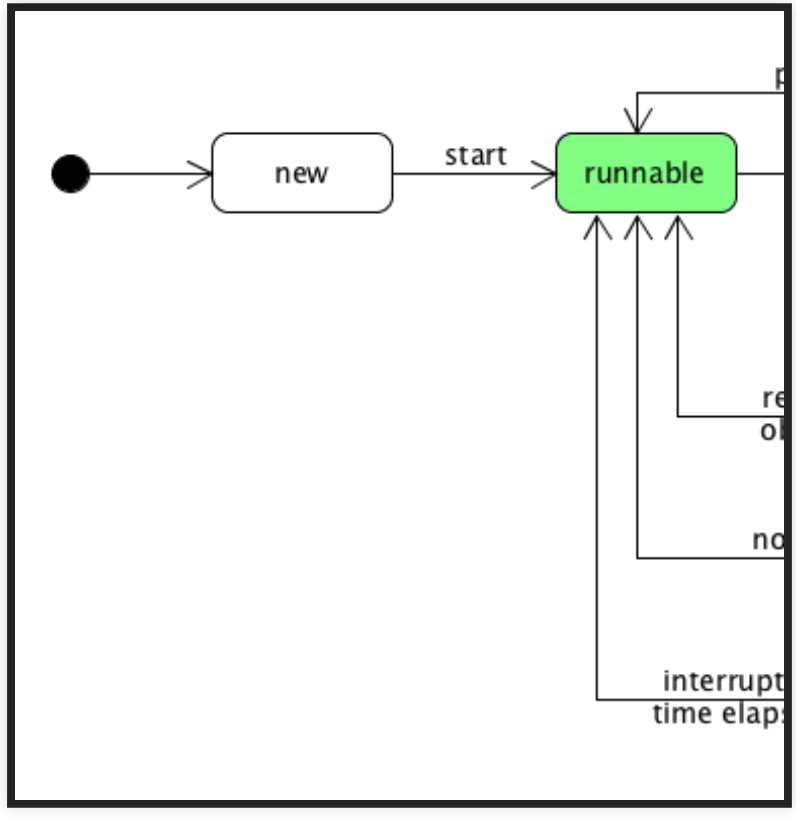
# STATO DEL THREAD




# java.lang.Thread.State

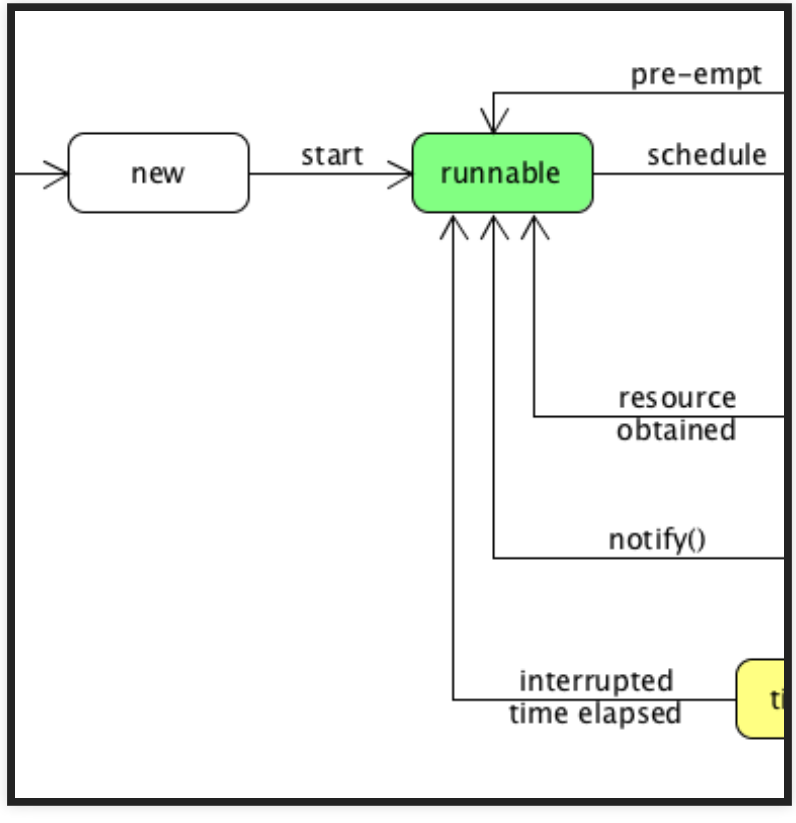
# NEW

Il Thread è stato creato.



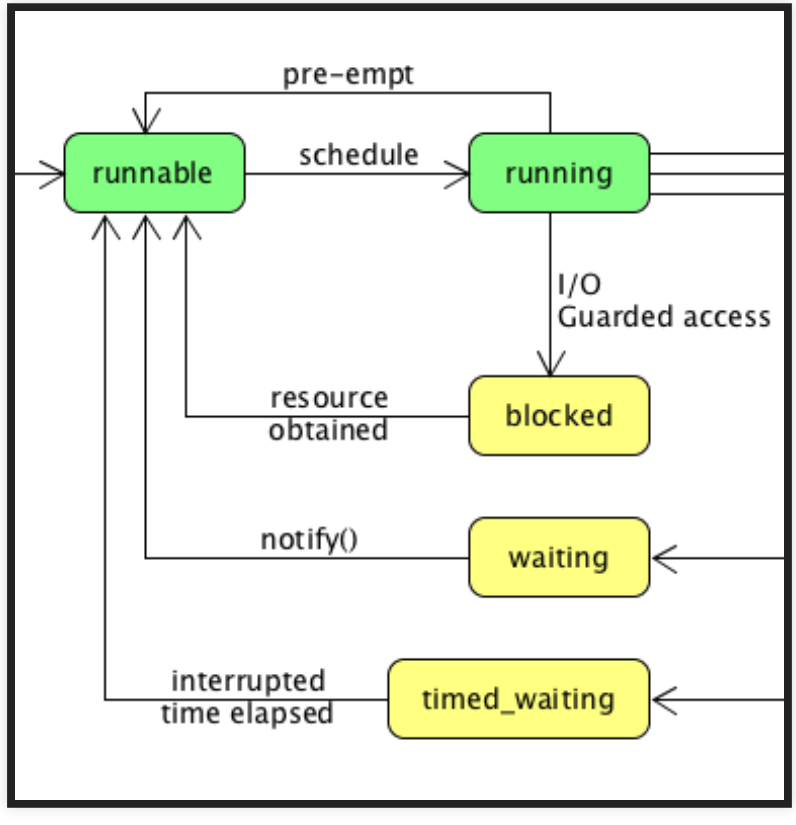
# RUNNABLE

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

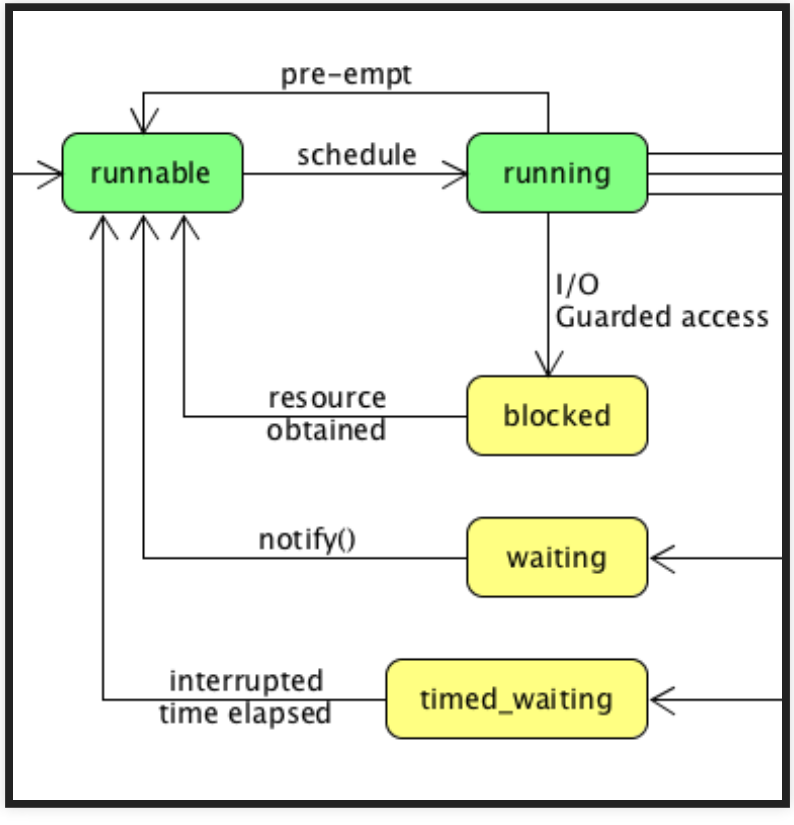


# RUNNING

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

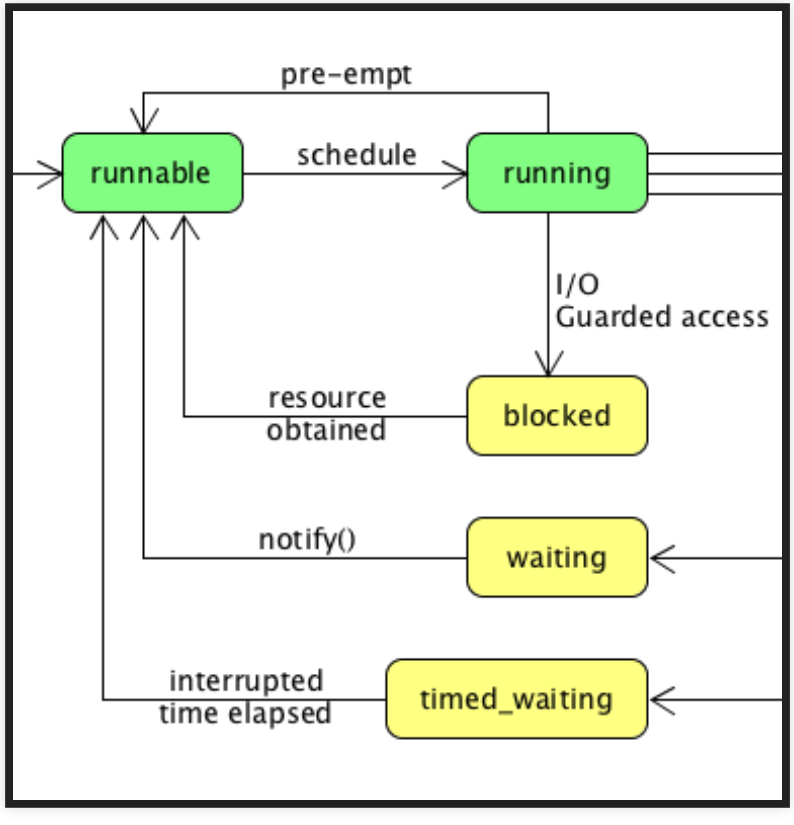


# BLOCKED



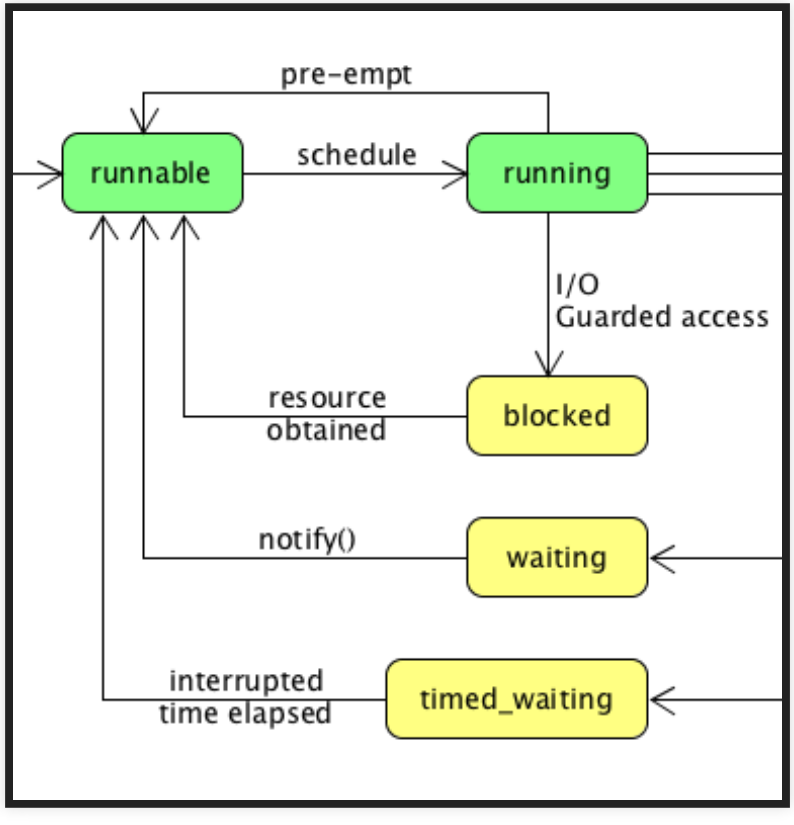

Il Thread ha richiesto accesso ad una risorsa monitorata (per es. un canale di I/O) e sta aspettando la disponibilità di dati.

# WAITING




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

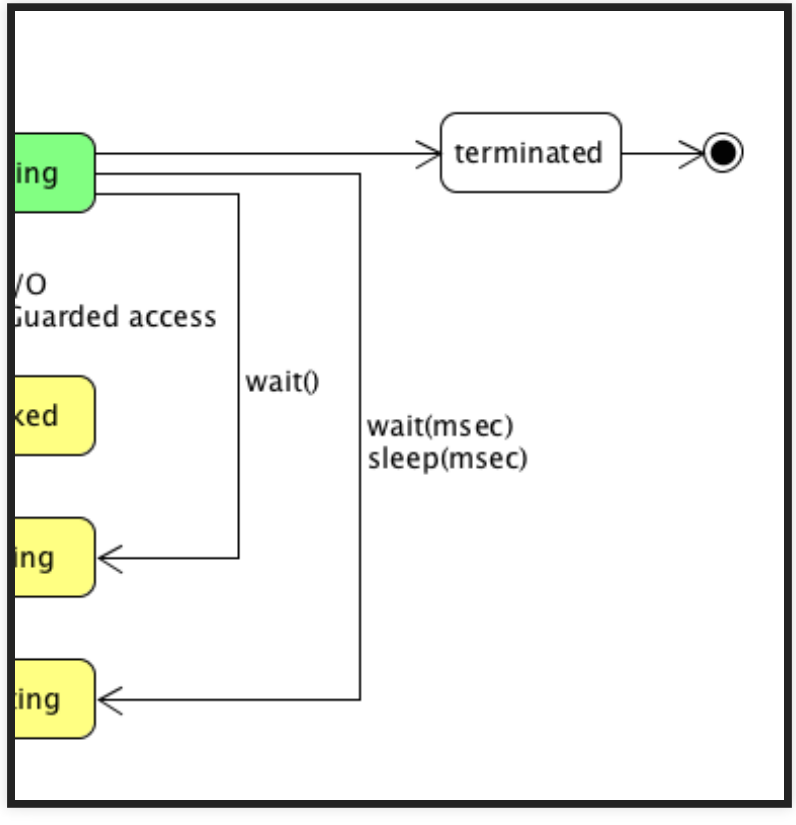
# TIMED_WAITING




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()
```

Speaker notes

Come abbiamo appena detto, un thread può concludere la sua esecuzione correttamente o meno. Fra i modi non corretti abbiamo il lancio di una eccezione all'interno della sua esecuzione, oppure il metodo interrupt che può essere invocato anche da un altro thread.

# pcd2019.threads.ThreadInterrupter

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

Speaker notes

creiamo una classe `Interrupter` che implementa `Runnable` in questo modo.

```
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();
}
```

Speaker notes

interrompere un thread che non è vivo non porta a nessun risultato.

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

Speaker notes

possiamo impostare, per uno specifico thread, un gestore delle eccezioni che riceve le eccezioni non intercettate e può quindi modificare il modo in cui un thread termina in modo non previsto.

# pcd2019.threads.RethrowingThread

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

Speaker notes

creiamo uno specifico supplier per farci fornire dei threads che rilanciano l'eccezione di interruzione invece di gestirla. A tutti gli effetti, se interrotti questi thread lanciano una eccezione non gestita.

```
final Thread tgt = new RethrowingThreadBuilder(2000L).get();

tgt.setUncaughtExceptionHandler((Thread t, Throwable e) -> {
  System.out.println("Thread " + t.getName() +
    " has thrown:\n" + e.getClass() + ": " + e.getMessage());
});

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

Speaker notes

Impostiamo l'exception handler sul thread bersaglio: vedremo che l'handler viene richiamato e gestisce l'eccezione.

# 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.

```
/**
 * 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)
```

# pcd2019.threads.FixedThreadPool

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

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

Speaker notes

Notate come in questo caso la JVM sia rimasta attiva: l'ExecutorService rimane in attesa di nuovi compiti da eseguire, anche se il metodo `main` è concluso.

# pcd2019.threads.SingleThreadPool

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

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

Speaker notes

Dal nome e dal comportamento possiamo osservare come i compiti accodati siano eseguiti da un solo thread.

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 |

Speaker notes

Usa un algoritmo detto di work stealing per gestire il caso in cui le attività eseguite avviino ulteriori sotto-attività.
Interessante l'annotazione: `This implementation restricts the maximum number of running threads to 32767`

# 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
```

Speaker notes

Diversi `Executor` comunque implementano anche questa interfaccia.

```
/**
 * 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()
```

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

# pcd2018.threads.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()) {
  System.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

```
/**
 * 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)
```

Nota: quando questa chiamata ritorna, non tutti i `Callable` hanno completato l'esecuzione.

```
/**
 * 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)
```

Speaker notes

questa chiamata ritorna solo dopo che almeno uno dei `Future` ha completato.

# pcd2018.threads.AllFutures

```
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());

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

# pcd2018.threads.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());

System.out.println("Scheduling computations");
var result = executor.invokeAny(callables);
System.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()
```

```
/**
 * 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()
```

```
/**
 * 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()
```

# PUBBLICITÀ

# The Future of Java is Today

https://www.infoq.com/news/2018/10/codeone-java-keynote

Speaker notes

questo articolo è un breve riassunto della presentazione data da Oracle all'ultimo CodeOne, per spiegare la prospettiva ed la direzione futura della governance dell'ecosistema Java.

Packt Free Learning

https://www.packtpub.com/packt/offers/free-learning

Nota: registrandosi, ogni giorno si può trovare un diverso libro (a volte non recentissimo, ma spesso utile) gratuitamente.