

Example05

Table of contents

- [Pool of resources](#)
- [`it.unipr.informatica.concurrent`](#)
 1. [Thread pools](#)
 1. [Executors](#)
 1. [`Executor.java`](#)
 2. [`RejectedExecutionException.java`](#)
 3. [`ExecutorService.java`](#)
 4. [`Executors.java`](#)
 5. [`SimpleThreadPoolExecutorService.java`](#)
- [Example05](#)
 1. [`DownloadManager.java`](#)
 2. [`Example05.java`](#)

Pool of resources

Se abbiamo per esempio 10 risorse e 20 flussi di esecuzione, fintanto che i flussi di esecuzione richiedenti la risorsa non superano il numero di risorse disponibili, allora non ci saranno problemi.

Un gruppo di risorse e' chiamato *pool of resources*:

- alla creazione/eliminazione, il pool viene acquisito/rilasciato;
- le risorse sono assegnate alla richiesta;
- il controllo dell'accesso e' garantito dal pool.

`it.unipr.informatica.concurrent`

Thread pools

Sono oggetti usati dal flusso di esecuzione nel momento del bisogno: nel momento in cui ci serve un thread per computare, uno qualsiasi va bene.

Nelle configurazioni vengono sempre specificati 1024 thread per ogni processo, quindi nella JVM. Sono risorse da usare con parsimonia ma tipicamente non ce ne accorgiamo, tuttavia noi discutiamo il numero giusto per le nostre operazioni.

Executors

Per eseguire in concorrenza dei *task*:

- associato ad un thread pool;
- mette in coda i task che non possono essere eseguiti subito;
- provvede modo di restituire il risultato di un task;
- provvede modo per ritornare eccezioni che hanno causato la terminazione;
- permette d'interrompere task e thread associati.

L'esecutore viene usato dall'utente per gestire il pool di thread.

La classe `Executor` di JAVA si presenta come punto di accesso principale per un sottosistema (degli esecutori), la chiamiamo *façade* (termine francese).

Executor.java

Prendiamo dalla documentazione e vediamo che riceve un `Runnable` che mette in coda e che prima o poi verra' eseguito. Lancia una eccezione, non dichiarata nella signature siccome runtime exception.

```
package it.unipr.informatica.concurrent;

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

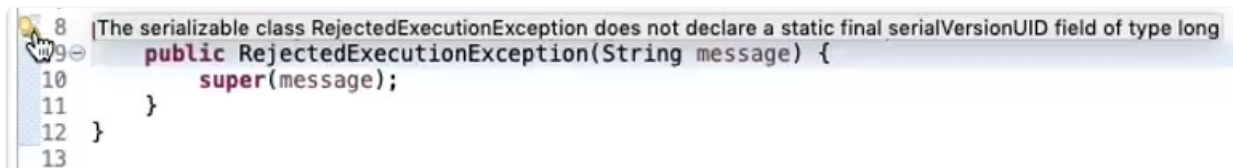
RejectedExecutionException.java

Estende `RuntimeException`, quindi puo' essere lanciata ma non dichiarata e catturata. Di solito queste eccezioni hanno costruttori:

- uno per output del messaggio d'errore;
- uno per catturare un throwable (l'eccezione permette di lanciare con tipo diverso);
- uno con messaggio + oggetto throwable embedded;
- costruttore senza argomenti.

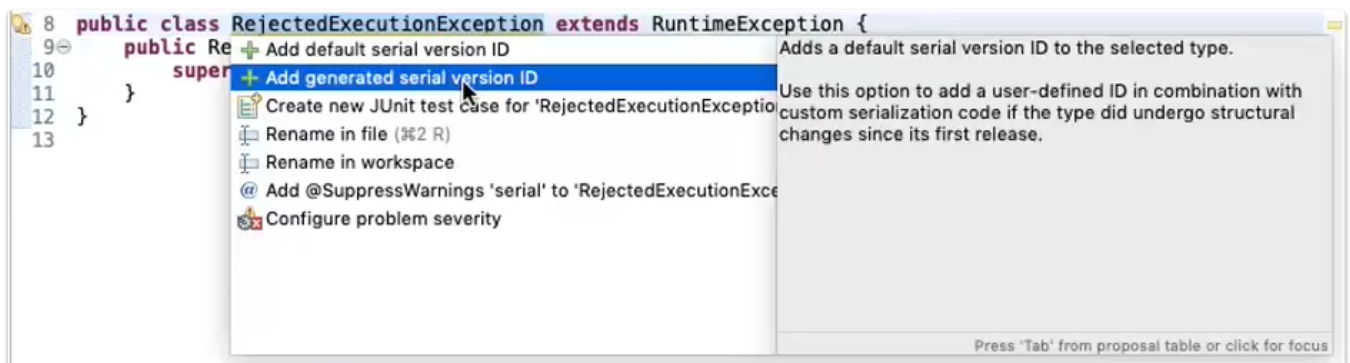
Noi ne mettiamo uno solo, siccome usiamo solo quello per il nostro scopo.

Alcune classi possono essere trasmesse come array di byte sulla rete, passanti attraverso la JVM sia in uscita che in ingresso le possiamo trasformare. Queste classi prendono il nome di *serializzabili*: una classe e' serializzabile se implementa `Serializable` e la nostra e' un caso.



Dobbiamo dare, tramite la JVM, un numero di versione univoco alla classe per far capire a chi rilegge l'oggetto, la sua versione, per garantire che le operazioni vengano fatte su quello che vogliamo e non sulla classe di un'altra versione.

Clicchiamo sul **warning** che ci viene segnalato e in Eclipse generiamo il nostro UID (seconda voce del menu).



```
package it.unipr.informatica.concurrent;

public class RejectedExecutionException extends RuntimeException {
    // hash del contenuto "quasi" testuale della classe
    private static final long serialVersionUID =
        -6748556787095111509L;

    public RejectedExecutionException(String message) {
        super(message);
    }
}
```

```
}  
}
```

ExecutorService.java

```
package it.unipr.informatica.concurrent;  
  
public interface ExecutorService extends Executor {  
    public void shutdown();  
}
```

Executors.java

Una classe che istanzia un insieme piu' piccolo si chiama *factory*.

Contiene solo metodi statici, il costruttore e' private e quindi la classe non puo' essere istanziata se non unicamente dalla classe stessa, solo i suoi metodi statici possono istanziarla.

Ci serve un solo metodo statico, per creare thread pool di dimensione fissata, `newFixedThreadPool(int count)` che istanzia `SimpleThreadPoolExecutorService`.

```
package it.unipr.informatica.concurrent;  
  
public class Executors {  
    public static ExecutorService newFixedThreadPool(int count) {  
        return new SimpleThreadPoolExecutorService(count);  
    }  
  
    private Executors() {  
        // blank  
    }  
}
```

SimpleThreadPoolExecutorService.java

Implementazione.

Costruiamo un numero di thread pari all'argomento che ci viene passato e aggiungiamo coda bloccante per i thread che vogliamo siano bloccati. Man mano che arrivano runnable da eseguire, i thread fanno il loro lavoro. Quando viene attivato `shutdown()`, le richieste di accodamento vengono bloccate e si chiede di completare la lettura dalla coda e andare a terminare i thread.

```
package it.unipr.informatica.concurrent;  
  
class SimpleThreadPoolExecutorService implements ExecutorService {  
    private Worker[] workers;  
    private BlockingQueue<Runnable> tasks;  
    private boolean shutdown;  
    // ...  
}
```

1. costruiamo una coda concatenata di `tasks` da eseguire di `Runnable`;
2. facciamo classi interne pari all'argomento passato di `Worker`;
3. costruiamo i `Worker` col ciclo.

Chiamiamo `start()` per fare partire i thread che cominciano a leggere dalla coda, chiamano `run()` e cosi' via.

```
// ...  
SimpleThreadPoolExecutorService(int count) {  
    if(count<1)  
        throw new IllegalArgumentException("count < 1");  
}
```

```

        this.shutdown = false;
        this.tasks = new LinkedBlockingQueue<>();
        this.workers = new Worker[count];
        for(int i=0; i<count; ++i) {
            Worker worker = new Worker();
            workers[i] = worker;
            worker.start();
        }
    }
    // ...

```

Verifica se possiamo o meno accettare il nuovo `task`, se `shutdown` e' impostato a `true` lanciamo `RejectedExecutionException`. Viceversa facciamo `put()` sulla coda e questa non si bloccherà mai siccome la nostra e' una lista concatenata.

```

    // ...
    @Override
    public void execute(Runnable command) {
        if(command == null)
            throw new NullPointerException("command == null");
        synchronized(tasks) {
            if(shutdown)
                throw new
                    RejectedExecutionException("shutdown == true");
            try {
                tasks.put(command);
            } catch(InterruptedException interruptedException) {
                // blank
                // non avverrà mai ma dobbiamo implementarla
            }
        }
    }
    // ...

```

```

    // ...
    @Override
    public void shutdown() {
        synchronized(tasks) {
            shutdown = true;
            int length = workers.length;
            for(int i=0; i<length; ++i)
                workers[i].shutdown();
        }
    }
    // ...

```

Il `Worker` ha bisogno di accesso alla coda e quindi lo mettiamo come inner class (oppure come argomento). Mantenuto privato, il thread, per evitare problemi d'interferenza dall'esterno (magari un interrupt).

```

    // ...
    private class Worker implements Runnable {
        private boolean shutdown;
        private Thread thread;
        private Worker() {
            this.shutdown = false;
            this.thread = new Thread(this);
        }
    }
    // ...

```

Ciclo infinito:

1. siamo in `shutdown`? se no vado avanti, altrimenti mi blocco;
2. `take()` va a leggere gli oggetti `Runnable` e li esegue con `run()`;
3. il ciclo infinito prende un'altra `take()` e così' via.

```
// ...
@Override
public void run() {
    for(;;) {
        synchronized(thread) {
            if(shutdown)
                return;
        }
        try {
            Runnable runnable = tasks.take();
            runnable.run();
        } catch(InterruptedException interruptedException) {
            return;
        } catch(Throwable throwable) {
            // stampa stack-trace
            // non vogliamo che si blocchi
            throwable.printStackTrace();
        }
    }
}
// ...
```

```
// ...
private void start() {
    thread.start();
}
// ...
```

```
// ...
private void shutdown() {
    synchronized(thread) {
        shutdown = true;
        thread.interrupt();
    }
}
}
```

Example05

DownloadManager.java

Riceve una serie di richieste di download e piano piano scarica dati dalla rete usando un URL, li consegna ai client e una volta terminati i dati, termina.

Il numero di dati scaricati vengono stampati.

```
package it.unipr.informatica.examples;
import java.io.InputStream;

public final class DownloadManager {
    private static final int BUFFER_SIZE = 1024;
    private ExecutorService executorService;

    public DownloadManager(int connections) {
```

```

        if(connections < 1)
            throw IllegalArgumentException("connections < 1");
        this.executorService =
            Executors.newFixedThreadPool(connections);
    }

    public void shutdown() {
        executorService.shutdown();
    }

    public void download(String url) {
        if(url==null)
            throw new IllegalArgumentException("url==null");
        executorService.execute(() -> downloadAndPrint(url));
    }
    // ...

```

Riceve una URL non necessariamente http e con new costruisce l'oggetto URL.

Possiamo aprire lo stream su cui andare a leggere, senza lettura diretta, costruendo un buffer di input stream. Mettiamo nello stesso i dati che vengono scaricati e lo usiamo anche per leggere (funziona come coda a capienza finita sui dati).

La cosa migliore, siccome non sappiamo quanti dati arrivano, lo scriviamo su array di byte

`ByteArrayOutputStream` su cui possiamo scrivere dati e che man mano si allunga senza problemi.

Stampiamo i dati scaricati e in qualsiasi momento se c'e' qualche problema, viene lanciata un'eccezione per quel URL.

```

    // ...
    private void downloadAndPrint(String url) {
        try (InputStream inputStream = new URL(url).openStream();
            BufferedInputStream bufferedInputStream =
                new BufferedInputStream(inputStream);
            ByteArrayOutputStream outputStream =
                new ByteArrayOutputStream()) {
            byte[] buffer = new byte[BUFFER_SIZE];
            int read = bufferedInputStream.read(buffer);
            while(read >= 0) {
                outputStream.write(buffer, 0, read);
                read = bufferedInputStream.read(buffer);
            }
            byte[] data = outputStream.toByteArray();
            System.out.println("Downloaded" + data.length +
                " bytes from " + url);
        } catch (Throwable throwable) {
            System.err.println("Cannot download with error: " +
                throwable.getMessage());
        }
    }
}

```

Example05.java

Scarichiamo da un po di siti web le pagine HTML.

Usa sempre https, alcuni siti ci sono tranne che per missingwebsite.com, cosi' vediamo cosa succede con URL errata. Il `DownloadManager` fara' 4 scaricamenti insieme.

Una volta terminati i 10 secondi di attesa, termina il programma.

Notiamo che l'ordine non e' lo stesso rispetto a quello scritto nel codice; ci sarebbe da fare un'analisi della rete in dettaglio per capire il motivo.

```

package it.unipr.informatica.examples;

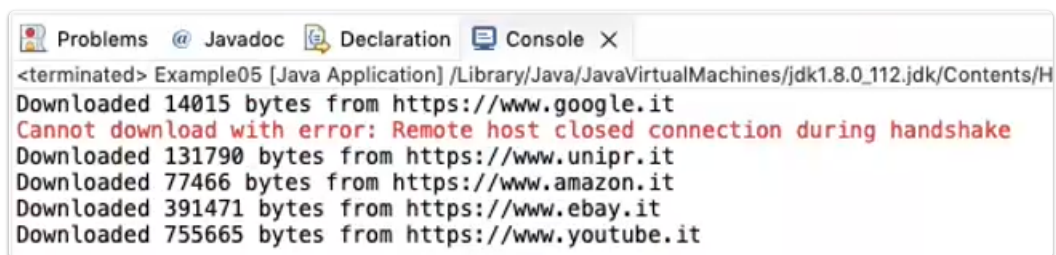
public class Example05 {
    private void go() {
        DownloadManager downloadManager = new DownloadManager(4);

        downloadManager.download("https://www.google.it");
        downloadManager.download("https://www.youtube.it");
        downloadManager.download("https://www.amazon.it");
        // \ 'host esiste ma non ha porta aperta(?)
        downloadManager.download("https://www.missingwebsite.com");
        downloadManager.download("https://www.ebay.it");
        downloadManager.download("https://maruko.it");

        try {
            Thread.sleep(10000);
            downloadManager.shutdown();
        } catch (InterruptedException interruptedException) {
            return;
        }
    }

    public static void main(String[] args) {
        new Example05().go();
    }
}

```



```

<terminated> Example05 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/H
Downloaded 14015 bytes from https://www.google.it
Cannot download with error: Remote host closed connection during handshake
Downloaded 131790 bytes from https://www.unipr.it
Downloaded 77466 bytes from https://www.amazon.it
Downloaded 391471 bytes from https://www.ebay.it
Downloaded 755665 bytes from https://www.youtube.it

```