#Appunti_24-11-22

Per far un passo avanti dobbiamo permettere ai tasks di produrre un risultato inteso come un valore di ritorno. quindi andiamo a vedere di realizzare l'esecuzione di task che diano dei risultati, che si dividono in:

- risultato effettivo
- eccezione

Interfaccia Callable <"V">

Metodo *call()*: non ha argomenti ma ha un valore di ritorno; computa un risultato o se no lancia un'eccezione, due tipi di eccezioni:

1. Quelle che estendono RunTimeException che possono essere ignorate.
2. Quelle che estendono Exception e devono essere gestite in due modi: le dichiariamo nell'intestazione del metodo oppure la catturiamo.

Una volta che abbiamo fornito l'oggetto Call che andremo ad utilizzare per eseguire del codice che produce un risultato di tipo T (parametro generico), come lo ritorniamo questo risultato al chiamante? Nel momento che abbiamo un Pool di Thread questo call(non sappiamo quando sarà eseguito e da chi) ad questo punto come facciamo ad metterci in attesa di questo risultato? Per farlo utilizziamo la interfaccia *Future*.

Future è un oggetto che ci viene ritornato subito, nel momento in cui abbiamo in mano il nostro future, continuiamo la nostra computazione lasciamo il tempo al task di arrivare a compimento, facciamo una chiamata get su future, se il risultato è presente ci viene rilasciato, se c'è un eccezione ci dara l'eccezione se no si mette in attesa, è un modo per fare randevue sul risultato/eccezione e tipicamente quello che succede è che il future aspetta.

Callable.java (INTERFACCIA)

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

public interface Callable<T> {
        public T call() thrwos Exception;
}
```

ExecutionException.java (CLASSE)

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

public class ExecutionException extends Exception {
	private static final long serialVersionUID = -1644619538323674773L;

		public ExecutionException(Throwable cause) {
			super(cause);
		}

}
```

La *ExecutionException* estende Exception, quindi non possiamo trascurarla di conseguenza bisogna gestirla come detto nella parte precedente, abbiamo un throwable che usiamo come eccezione di base.
Perchè stiamo lanciando questa ExecutionException? Perchè c'è stato un problema, che è stato descritto dal throwable che chiamiamo cause, lo passiamo nel costruttore, chiamiamo il costruttore della classe base che è Exception, passiamo il throwable e viene messo dentro alla nostra eccezione.

Future.Java (INTERFACCIA)

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

public interface Future<T> {
	public T get() throws InterruptedException, ExecutionException;

	public boolean isDone();

}
```

Future è un interfaccia di java.util.concurrent è tipato con un tipo generico T che è il tipo del risultato, ha un metodo *get()* che ci permette di leggere il contenuto del future, quindi metterci

in attesa se non c'è ancora oppure leggerlo; inoltre ha un metodo *isDone()* che verifica se effettivamente il risultato o l'eccezione sono stati prodotti, in pratica ci permette di sapere se la prossima get() sarà bloccante o no. Se isDone torna true la prossima get() non si bloccherà se invece torna false siccome non è fatto in modo atomico la get() potrebbe avere il risultato e non si blocca oppure la prossima get() non ha il risultato e si blocca.

ExecutorService.java (INTEFACCIA)

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

public interface ExecutorService extends Executor {
        public void shutdown();

        public Future<?> submit(Runnable task);

        public <T> Future<T> submit(Callable <T> task);
}
```

Aggiunti due metodi: il *primo* è un corner case cioè facciamo submit otteniamo un future ma la Runnable è void, e cosa ci mettiamo dentro al valore del future, convenzionalmente nullo; questa submit è un pò strana, in pratica ci permette di metterci in attesa che il Runnable sia terminato, nel momento che facciamo get() dal future che viene ritornato viene sicuramente ritornato null e questo è possibile, poi serve perchè nel momento che la get() non si blocca o si sblocca il task è finito, quindi serve sostanzialmente come via di mezzo per avere la possibilità di mettersi in attesa di un task che però non producendo valori non ha un valore di ritorno (neppure eccezioni), in poche parole è un modo per sapere quando è finito il task e se è andato bene. Il punto interrogativo è utilizzabile quando non sappiamo dire che tipo è al compilatore. Il *secondo* è la submit con un Callable, e siccome ha un Callable ha un tipo di ritorno T, e a questo punto ci viene ritornato un future di tipo T (i due tipi devono essere uguali).

SimpleFuture.java

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

class SimpleFuture<T> implements Future<T> {
                private Object mutex;
```

```
        private T value;
        private Throwable exception;
        private boolean done;

        SimpleFuture(){
                this.mutex = new Object();

                this.done = false;

                this.value = null;

                this.exception = null;
        }
        @Override
        public T get() throws InterruptedException, ExecutionException{
                synchronized (mutex) {
                        while(!done)
                                mutex.wait();

                        if(exception != null)
                                throw new ExecutionException(exception);

                        return value;
                }

        }

        @Override
        public boolean isDone() {
                synchronized (mutex) {

                        return done;
                }
        }
        void setValue(T object) {
                synchronized (mutex) {
                        if(done)
                                throw new IllegalStateException("done ==
true");

                        value = object;

                        done = true;

                        mutex.notifyAll();
                }
```

```
                }
                void setException(Throwable throwable) {
                        if(throwable == null)
                                throw new IllegalArgumentException("throwable ==
null");

                        synchronized (mutex) {
                                if(done)
                                        throw new IllegalStateException("done ==
true");
                                exception = throwable;

                                done = true;

                                mutex.notifyAll();
                        }
        }
}
```

Il nostro utente non dovrà mai fare new SimpleFuture perchè non voglia che gestista niente all'infuori dell'interfaccia e quindi lo facciamo package scope.
Il nostro simpleFuture ha un costruttore package scope (quindi non c'è scritto niente ne pubblico ne privato) e contiene il suo stato formato da quattro attributi:

1. *mutex* per identificare la sezione critica
2. *un booleano* done che dice se effettivamente è stato prodotto o no il risultato o l'eccezione
3. un valore di tipo T (value) perchè se ci viene prodotto un risultato dobbiamo immagazzinarlo
4. un valore che è l'eccezione e bisogna memorizzarla

Nel punto 3-4 non possiamo avere un solo valore che gestisce entrambi perchè non necessariamente hanno un'interfaccia in comune.

- *public T get() thrwos InterruptedException, ExecutionException*: una sezione critica che ci permette di verificare se c'è il valore di risultato o l'eccezione, in particolare questa verifica viene fatta su done se è false ci mettiamo in attesa, al contrario se è presente il valore o l'eccezione vengono ritornare, nel caso il mutex è in wait e si sveglia possono succedere due cose : 1) effettivamente si è sbloccata perchè è stata fatta una notify a questo punto ritorniamo nel while e verifichiamo il done. 2) abbiamo la necessità di capire se è stata lanciata l'eccezione, l'idea è che il task genera l'eccezione e nel momento che lo genera, questa viene catturata e messa dentro al future, e vuol dire che non è null e la mettiamo dentro alla ExecutionExceptione viene lanciata, se è nulla in realtà non è nullo allora limitiamo a ritornarlo.

- *public boolean isDone()*: che si limita in modo atomico a ritornare done, quindi in un blocco synchronized con la guardia mutex ritorna done, se è false non sono state prodotte ne eccezioni ne valor di risultato, se invece è true la prossima get non si blocca perchè l'eccezione o il valore di risultato sono presenti.

Qualcuno dovrà fare il set del valore e il set dell'eccezione, queste non vengono messe nell'interfaccia perchè le faremo fare al nostro executor.

- *void setValue(T object)*: viene utilizzato per scrivere il risultato del task all'interno del future, blocco synchronized per fare le cose in mutua esclusione se done è true vuol dire che è già stato prodotto un risultato e non può essere (bug), se done è false impostiamo value, il future conosce un riferimento all'oggetto risultato, impostiamo done a true e svegliamo tutti quelli in attesa del future.
- *void setException(Throwable throwable)*: stessa logica del setValue ma con un controllo aggiuntivo che è quello di controllare se throwable non sia null.

Aggiunti due metodi a SimpleThreadPoolExecutorService.java

```
@Override
public Future<?> submit(Runnable task){
        SimpleFuture<?> future = new SimpleFuture<>();

        execute(() -> {
                try {
                        task.run();

                        future.setValue(null);
                }catch (Throwable throwable) {
                        future.setException(throwable);
                        }

        });
        return future;
}

@Override
public <T> Future<?> submit(Callable<T> task){
        SimpleFuture<?> future = new SimpleFuture<>();

        execute(() -> {
                try {
                        T result = task.call();
```

```
                future.setValue(result);
            }catch (Throwable throwable) {
                future.setException(throwable);
            }
        });
        return future;
    }
```

- *public Future < ? > submit(Runnable task)* : avrà l'effetto di ritornare un future e di impostare il valore di questo a nullo; quindi costruiamo un future, poi chiediamo un'esecuzione di un task, e quali vengono eseguiti? Quelli che implementano Runnable, per evitare una inner class facciamo un lambda expression che non è nient'altro che l'implementazione dell'interfaccia runnable, cosa fa il try catch? esegue la run del task che ha ricevuto come argomento, una volta eseguita abbiamo due casi possibili:
- 1)Se tutto è andato a buon fine impostiamo il valore del future a nullo ed eventuali get in attesa ritorneranno nullo, cosi come tutte quelle che verranno.
  2)Se è stata lanciata un'eccezione la catturiamo, e la impostiamo come eccezione nel future.