Conception de logiciels intranet : intergiciels à objets répartis (NSY102)

Calculs asynchrones

Interface `Future<T>`

L'interface `Future<T>` de la JDK représente un calcul qui doit ultimement (à un moment dans le futur) renvoyer un résultat de type `T`. L'interface `RunnableFuture<T>` de la JDK étend en plus l'interface `Runnable`, et la classe `FutureTask<V>` implante cette interface.

Si `e` est une expression de type `T`, on peut créer un tel calcul en utilisant simplement le constructeur ainsi `new FutureTask(() -> e)`. Ce calcul peut alors être exécuté de manière concurrente.

On peut aussi stocker ce calcul en cours d'execution, par exemple dans une variable `x`, et il est possible ensuite de récupérer le résultat avec l'expression `x.get()`, en précisant éventuellement le temps que l'on est prêt à attendre pour obtenir le résultat. La méthode `get()` qui est une opération bloquante.

Questions

Ecrire une classe `FutureTask<T>` qui implante l'interface `RunnableFuture<T>` en utilisant les threads et les `BlockingQueue<E>` pour réaliser la communication entre une instance de `FutureTask<T>` et le serveur qui réalise effectivement de calcul (le calcul sera lancé lorsque `run` est invoqué).

Les méthodes à implanter sont les suivantes :

`boolean`	`cancel(boolean mayInterruptIfRunning)`
	Attempts to cancel execution of this task.
`V`	`get()`
	Waits if necessary for the computation to complete, and then retrieves its result.
`V`	`get(long timeout, TimeUnit unit)`
	Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
`boolean`	`isCancelled()`
	Returns `true` if this task was cancelled before it completed normally.
`boolean`	`isDone()`
	Returns `true` if this task completed.

Calculs asynchrones

1. Pour faire les tests avec de “vrais” calculs, nous donnons l'implantation naïve suivante de la suite de Fibonacci :

   ```
   static Integer fib(Integer n) {
       if (n <= 1) return 1;
       else return fib(n - 1) + fib(n - 2);
   }
   ```

   Le plus grand terme de la suite calculable avec des entiers 32 bits signés est `fib(45)` et le temps de calcul est de quelques secondes.

2. Implanter une méthode de fabrique statique qui crée une `FutureTask<V>` puis lance son calcul de manière asynchrone en utilisant un nouveau `Thread` par tâche :

   ```
   static <V> Future<V> futureTask(Callable<V> callable)
   ```

   Généraliser `futureTask` en lui passant un `Executor` en paramètre, et tester avec les nouveaux threads virtuels de Java 19 (la méthode `newVirtualThreadPerTaskExecutor` de la classe `Executors` sera permet de créer un tel `Executor`).

3. Faire les tests suivants :

   ```
   var x = futureTask(() -> fib(45));
   ```

   ```
   var r = x.get();
   ```

   ```
   var l = List.of(futureTask(() -> fib(43)),
                   futureTask(() -> fib(44)),
                   futureTask(() -> fib(45)));
   ```

   ```
   for (var f : l) System.out.println(f.get());
   ```

4. Afin de permettre l'enchaînement de calculs asynchrones, et d'éviter le plus possible des barrières de synchronisation (dû à l'utilisation de la méthode bloquante `get`), nous ajoutons à la classe `FutureTask<V>` les méthodes suivantes :

   ```
   <T> FutureTask<T> map(Function<V,T> f)
   ```

   ```
   <T> FutureTask<T> flatMap(Function<V,FutureTask<T>> f)
   ```

   Remarque. Dans l'interface `CompletableStage<T>` de la JDK, `map` s'appelle `thenApply` et `flatMap` s'appelle `thenCompose`.