Java Futures

Short introduction

© Natallia Müller, ING Germany

Java Future

- Java `Future` is an interface used to represent the result of asynchronous computation
- Exists since Java 5
- Comes from `java.util.concurrent` package
- You need an `ExecutorService` to use the Future, it provides the threads pool and assign the tasks to the threads
- The task being executed should implement `Callable` interface

Java Future

```
PredictionGenerator infoService = new PredictionGenerator();
ExecutorService service = Executors.newSingleThreadExecutor();
Future<String> futureResult = service.submit(infoService::generatePrediction);

String result = "no result";

try {
    result = futureResult.get();
} catch (InterruptedException | ExecutionException e) {
    //do nothing
}
System.out.println(result);

service.shutdown();
```

Executor / Callable

- `Callable` interface exposes a single method, returning a single value
- Similar to `Runnable`, but returns a value in form of `Future`
- `Executor` is an object, that executes submitted runnable tasks
- `ExecutorService` provides a pool of threads for execution of the tasks
- Tasks for execution are submitted via `submit()`

Pull solution

- To get the result of the `Future`, the method get() can be called.
- This blocks the thread until the result can be asynchronously calculated
- In failed case, an `ExecutionException` will be thrown.

Limits:

By lots of tasks lots of thirds are needed, which are waiting for execution -> very resource intensive!!! -> limited for scaling

Poll solution

- Use overloaded `get()`-method

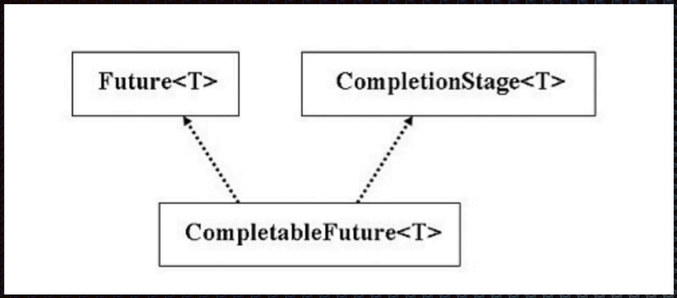
```
V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,
TimeoutException;
```

- After timeout has been reached, the method throws the `TimeoutException` and informs, that the result is still not available

```
try {
    result = futureResult.get( timeout: 2, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException | TimeoutException e) {
    //do nothing
}
System.out.println(result);
```

Limits: CPU resources for polling

CompletableFuture



- Since Java 8
- Asynchronous, non-blocking tool
- Allows to write the code in more responsive and efficient way

CompletableFuture

- The method `thenAccept()` contains the future result, which can be used for further operations (in example print the prediction)
- The lambda-expression will be triggered as soon as the asynchron result is delivered
- The threads is not blocked and runs independently on the availability of the result

```
CompletableFuture<String> futurePrediction = CompletableFuture.supplyAsync(generator::generatePrediction);

futurePrediction.thenAccept(result -> {
    System.out.println(result);
});
```

- In error case nothing will happen (no exception is thrown)

CompletableFuture: `thenAccept()`

`CompletableFuture<Void>`	**`thenAccept`**`(Consumer<? super T> action)` Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied action.
`CompletableFuture<Void>`	**`thenAcceptAsync`**`(Consumer<? super T> action)` Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage's result as the argument to the supplied action.
`CompletableFuture<Void>`	**`thenAcceptAsync`**`(Consumer<? super T> action, Executor executor)` Returns a new CompletionStage that, when this stage completes normally, is executed using the supplied Executor, with this stage's result as the argument to the supplied action.

- In case of asynchronous execution we can pass through the thread pool of type `Executor`.
- If no `Executor` is defined, the `ForkJoinPool` will be used by default in Java, in case this one has more than 2 threads. Otherwise the new thread will be created explicitly for task execution.

CompletableFuture: basic functions

```
<U>  CompletableFuture<U>       thenApply    (Function<T,U>   task)

     CompletableFuture<Void>    thenRun          (Runnable        task)

     CompletableFuture<Void>    thenAccept       (Consumer<T>   task)
```

- `thenApply()` gets the result T of the previous step and produces a new output of type U
- `thenRun()` gets a `Runnable` task for execution (no input, no output)
- `thenAccept()` has input, but doesn't produce any output
- Every of these methods has 3 modifications (e.g. `thenRun()` and 2 `thenRunAsynch()`)

CompletableFuture: basic functions

`<U> CompletableFuture<U>`	**`thenCompose`**`(Function<? super T,? extends CompletionStage<U>> fn)` Returns a new CompletionStage that, when this stage completes normally, is executed with this stage as the argument to the supplied function.

- `thenCompose()` works like `FlatMap`

`<U> CompletableFuture<U>`	**`applyToEither`**`(CompletionStage<? extends T> other, Function<? super T,U> fn)` Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed with the corresponding result as argument to the supplied function.

- `applyToEither()` works with the first result

`<U,V> CompletableFuture<V>`	**`thenCombine`**`(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)` Returns a new CompletionStage that, when this and the other given stage both complete normally, is executed with the two results as arguments to the supplied function.

- `thenCombine()` waits for both results

CompletableFuture: exception handling

- No checked exceptions are accepted
- If a checked exception occurs in one phase, the chain execution will be terminated and the exception gets lost. The termination will be not announced
- Two possibilities:

1. Handle exception inside of the task
2. Use error-handle support for CompletableFuture

CompletableFuture: exception handling

`CompletableFuture<T>`	**`exceptionally`**`(Function<Throwable,? extends T> fn)` Returns a new CompletableFuture that is completed when this CompletableFuture completes, with the result of the given function of the exception triggering this CompletableFuture's completion when it completes exceptionally; otherwise, if this CompletableFuture completes normally, then the returned CompletableFuture also completes normally with the same value.

```
service.findPredictionCompletableFutureWithError()
        .exceptionally(throwable -> {
            System.out.println("Exception occured: " + throwable);
            return "Sorry, no prediction today";
        })
        .thenAccept(System.out::println);
```

CompletableFuture: exception handling

<U> CompletableFuture<U>	handle(BiFunction<? super T,Throwable,? extends U> fn) Returns a new CompletionStage that, when this stage completes either normally or exceptionally, is executed with this stage's result and exception as arguments to the supplied function.

```
service.findPredictionCompletableFutureWithError()
        .handle((result, error) -> {
            if (error != null) {
                System.out.println("Exception occured: " + error);
                result = "Sorry, no prediction today";
            }
            System.out.println(result);
            return (Void) null;
        }).join();
```

CompletableFuture: exception handling

`CompletableFuture<T>`	**`whenComplete`**`(BiConsumer<? super T,? super Throwable> action)` Returns a new CompletionStage with the same result or exception as this stage, that executes the given action when this stage completes.

```
service.findPredictionCompletableFutureWithError() CompletableFuture<String>
        .thenAccept(System.out::println) CompletableFuture<Void>
        .whenComplete((v,e) -> {
            if (e != null) {
                System.out.println("Exception occured: " + e);
            }
        });
```

For more information, check the documentation.