JAVA BACKEND DEVELOPMENT PROGRAM

Asynchronous

OUTLINE

- Asynchronous
  - Introduction
  - @Async
    - Future
    - CompletableFuture

SCENARIO

- In enterprise application, it is very common to talk about the performance of the application. Think about the following scenario:
  - In order to prepare the home page, we have to make 4 independent calls
    - UserInfo — it takes 2 seconds to get the result
    - AccountInfo — it takes 4 seconds to get the result
    - BillingInfo — it takes 1 seconds to get the result
    - UsageInfo — it takes 5 seconds to get the result
  - Now we are calling the four services in sequence, and ignore the time costing for combine all results, how long will it take for UI to the response for home page?
  - How can we improve the performance in this scenario?

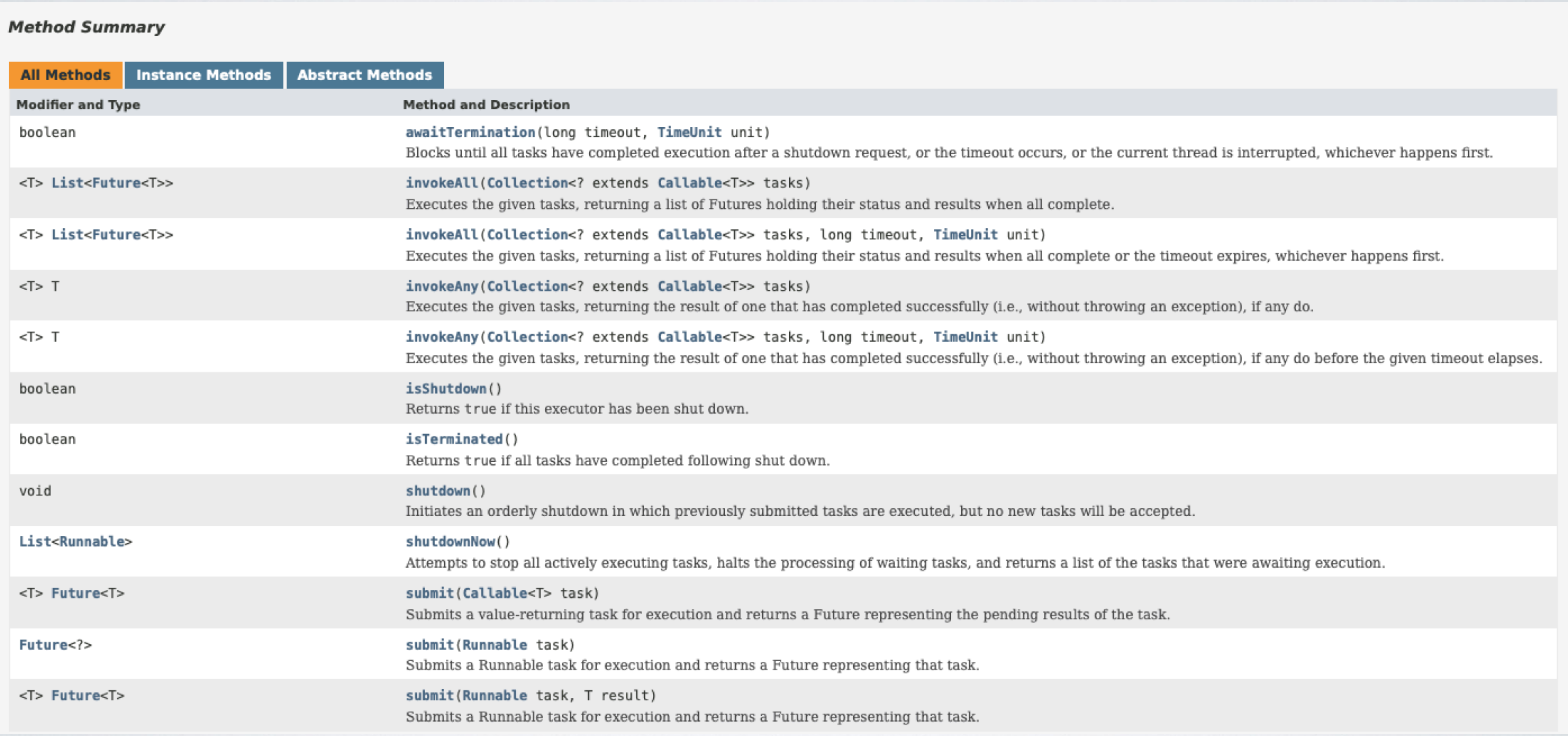
ASYNCHRONOUS

- For independent tasks, we can run them in different threads and combine the result until all threads finish.
- Two ways to implement the asynchronous process
  - @Async
  - Messaging

MULTI-THREADING IN JAVA

- We already knew that we can use Runnable or Thread to create thread manually in Java.
- But what if I want to limit the number of thread that can be created in one application?
- The Concurrency API introduces the concept of an **ExecutorService** as a higher level replacement for working with thread directly.
    - Executors are capable of running asynchronous tasks and typically manage a **pool** of threads; so we don't have to create are thread manually.
    - All threads of the internal pool will be reused under the hood of revenant tasks, so we can run as many as concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

EXECUTOR SERVICE



Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
boolean	`awaitTermination(long timeout, TimeUnit unit)` Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
`<T> List<Future<T>>`	`invokeAll(Collection<? extends Callable<T>> tasks)` Executes the given tasks, returning a list of Futures holding their status and results when all complete.
`<T> List<Future<T>>`	`invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.
`<T> T`	`invokeAny(Collection<? extends Callable<T>> tasks)` 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, long timeout, TimeUnit unit)` Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.
boolean	`isShutdown()` Returns `true` if this executor has been shut down.
boolean	`isTerminated()` Returns `true` if all tasks have completed following shut down.
void	`shutdown()` Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
`List<Runnable>`	`shutdownNow()` Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
`<T> Future<T>`	`submit(Callable<T> task)` Submits a value-returning task for execution and returns a Future representing the pending results of the task.
`Future<?>`	`submit(Runnable task)` Submits a Runnable task for execution and returns a Future representing that task.
`<T> Future<T>`	`submit(Runnable task, T result)` Submits a Runnable task for execution and returns a Future representing that task.

EXECUTOR SERVICE

- submit - Submits a task to the ExecutorService, take both Runnable and Callable interface.
    - Runnable won't return anything.
    - Callable will return a Future object which can be used to retrieve the actual result at the later point in time.
- shutDown - Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
- awaitTermination - Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
- shutDownNow - Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

EXECUTOR SERVICE

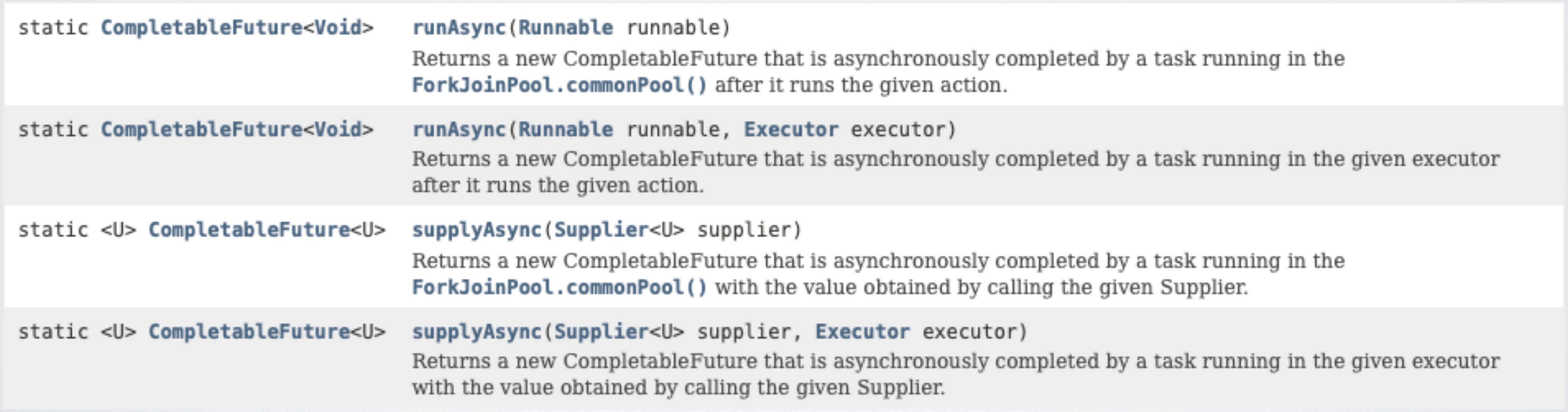
- invokeAll - Executes the given tasks, returning a list of Futures holding their status and results when all complete.
- invokeAny - Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.

COMPLETABLE FUTURE

- CompletableFuture is used for asynchronous programming in Java
- Asynchronous programming is a means of writing *non-blocking* code by running a task on a separate thread than the main application thread and notifying the main thread about its progress, completion or failure
- We can use completable future instead of implementing Runnable interface to create new thread, but under the hook, it still uses the Runnable to implement the multi-threading.
- Here I will introduce some functions that we need to use to implement asynchronous process in web application.

COMPLETABLE FUTURE

- How to get CompletableFuture from ExecutorService.



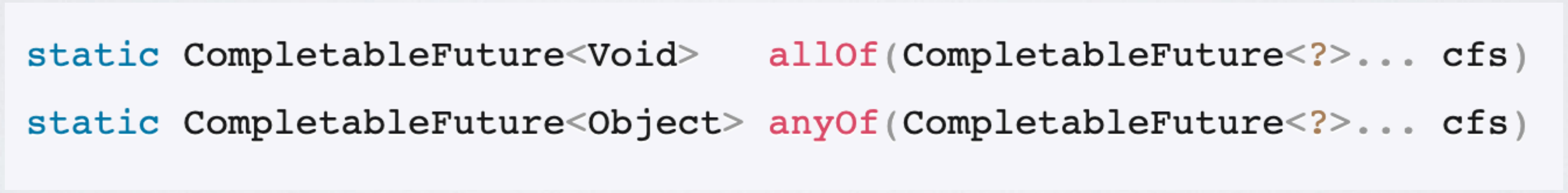
`static CompletableFuture<Void>`	`runAsync(Runnable runnable)` Returns a new CompletableFuture that is asynchronously completed by a task running in the `ForkJoinPool.commonPool()` after it runs the given action.
`static CompletableFuture<Void>`	`runAsync(Runnable runnable, Executor executor)` Returns a new CompletableFuture that is asynchronously completed by a task running in the given executor after it runs the given action.
`static <U> CompletableFuture<U>`	`supplyAsync(Supplier<U> supplier)` Returns a new CompletableFuture that is asynchronously completed by a task running in the `ForkJoinPool.commonPool()` with the value obtained by calling the given Supplier.
`static <U> CompletableFuture<U>`	`supplyAsync(Supplier<U> supplier, Executor executor)` Returns a new CompletableFuture that is asynchronously completed by a task running in the given executor with the value obtained by calling the given Supplier.

COMPLETABLE FUTURE

- All the caller who want to wait the task execution and get the result of this CompletableFuture can call
  - completableFuture.get()
  - completableFuture.join()
  - The main difference between get() and join() is that get() will throw InterruptedException and ExecutionException as checked exception, while join() only throw unchecked exception. Also get() allows to specify max wait time.

COMPLETABLE FUTURE

- Combining multiple completable future



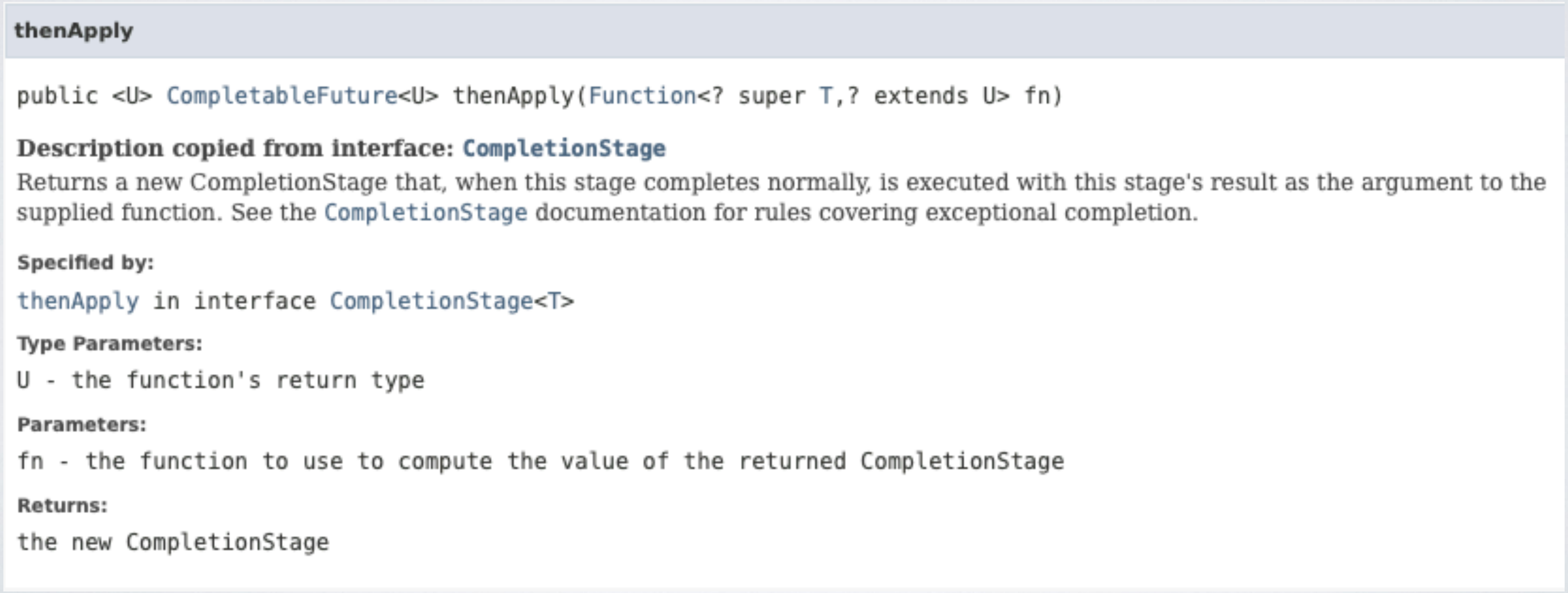
```
static CompletableFuture<Void>   allOf(CompletableFuture<?>... cfs)
static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
```

- CompletableFuture.allOf is used in scenarios when you have a List of independent futures that you want to run in parallel and do something after all of them are complete

- CompletableFuture.anyOf() as the name suggests, returns a new CompletableFuture which is completed when any of the given CompletableFutures complete, with the same result

COMPLETABLE FUTURE

- Processing the result of CompletableFuture



thenApply

```
public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)
```

Description copied from interface: CompletionStage
Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied function. See the CompletionStage documentation for rules covering exceptional completion.

Specified by:
thenApply in interface CompletionStage<T>

Type Parameters:
U - the function's return type

Parameters:
fn - the function to use to compute the value of the returned CompletionStage

Returns:
the new CompletionStage

@ASYNC

- Annotating a method of a bean with @Async will make it execute in a separate thread
    - It must be applied to public methods only
        - The method needs to be public so that it can be proxied
    - Self-invocation – calling the async method from within the same class – won't work
        - It bypasses the proxy and calls the underlying method directly
- To get started, we need to add @EnableAsync to either the Starter or Configuration classes.
- We can use Completable Future as the return type of the method annotated with *@Async*

@ASYNC

- Just as we can configure the connection pool for Database Connections, there is a way for us to configure the thread pool.
- Executors — The *Executors* helper class contains several methods for creation of pre-configured thread pool instances for you
    - Those classes are a good place to start with – use it if you don't need to apply any custom fine-tuning.
- By default, Spring uses a *SimpleAsyncTaskExecutor* to actually run these methods asynchronously
- We can override the the default at
    - Application level — rarely used since we may have different executor for different usage
    - Method level

QUESTIONS