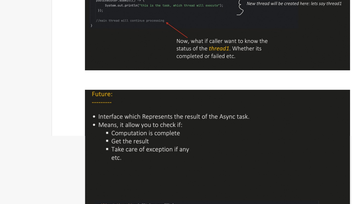
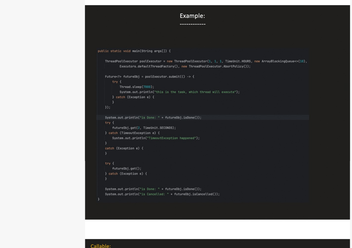
Future, CompletableFuture and Callable

"Concept && Coding" YT Video Notes



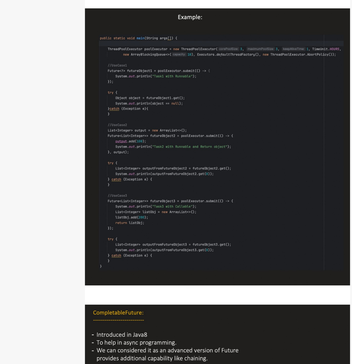
Future:

- Interface which Represents the result of the Async task.
- Means, it allow you to check if:
  - Computation is complete
    - Get the result
    - Take care of exception if any etc.



S.No.	Method Available in Future Interface	Purpose
1.	boolean cancel(boolean mayInterruptIfRunning)	• Attempts to cancel the execution of the task. • Returns false, if task can not be cancelled (typically bcos task already completed); returns true otherwise.
2.	boolean isCancelled()	• Returns true, if task was cancelled before it get completed.
3.	boolean isDone()	• Returns true if this task completed. Completion may be due to normal termination, an exception, or cancellation -- in all of these cases, this method will return true.
4.	V get()	• Wait if required, for the completion of the task. • After task completed, retrieve the result if available.
5.	V get(long timeout, TimeUnit unit)	• Wait if required, for at most the given timeout period. • Throws 'TimeoutException' if timeout period finished and task is not yet completed.

Example:



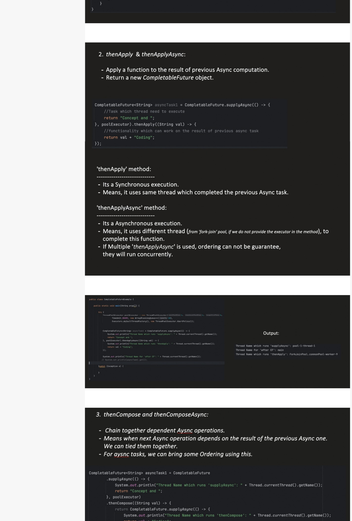
Callable:

- Callable represents the task which need to be executed just like Runnable.
- But difference is:
  - Runnable do not have any Return type.
  - Callable has the capability to return the value.

Runnable Interface:	Callable Interface:

@FunctionalInterface
public interface Runnable {
  public abstract void run();
}
	@FunctionalInterface
	public interface Callable<V> {
	  V call() throws Exception;
	}

Example:



CompletableFuture:

- Introduced in Java8
- To help in async programming.
- We can considered it as an advanced version of Future provides additional capability like chaining.

How to use this:

1. CompletableFuture.supplyAsync:

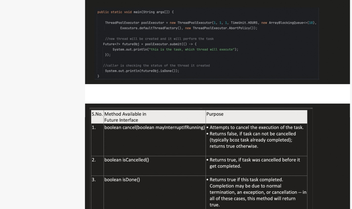
public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier)

public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier, Executor executor)

- *supplyAsync* method initiates an Async operation.
- 'supplier' is executed asynchronously in a separate thread.
- If we want more control on Threads, we can pass Executor in the method.
- By default its uses, shared **Fork-join Pool** executor. it dynamically adjust its pool size based on processors.

2. *thenApply* & *thenApplyAsync*:

- Apply a function to the result of previous Async computation.
- Return a new *CompletableFuture* object.



'thenApply' method:

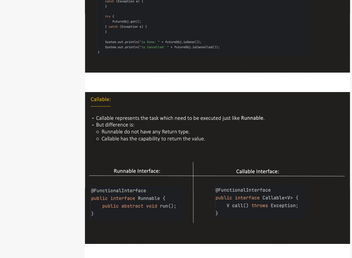
- Its a Synchronous execution.
- Means, it uses same thread which completed the previous Async task.

'thenApplyAsync' method:

- Its a Asynchronous execution.
- Means, it uses different thread (*from 'fork-join' pool, if we do not provide the executor in the method*), to complete this function.
- If Multiple '*thenApplyAsync*' is used, ordering can not be guarantee, they will run concurrently.

3. *thenCompose and thenComposeAsync:*

- *Chain together dependent Async operations.*
- *Means when next Async operation depends on the result of the previous Async one. We can tied them together.*
- *For async tasks, we can bring some Ordering using this.*



4. *thenAccept and thenAcceptAsync:*

- *Generally end stage, in the chain of Async operations*
- *It does not return anything.*

5. *thenCombine and thenCombineAsync:*

- *Used to combine the result of 2 Comparable Future.*