# Future, CompletableFuture and Callable

```
public static void main(String args[]) {

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1, 1, 1, TimeUnit.HOURS, new ArrayBlockingQueue<>(10),
            Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

    //new thread will be created and it will perform the task
   poolExecutor.submit(() -> {
        System.out.println("this is the task, which thread will execute");
    });

   //main thread will continue processing
}
```

*New thread will be created here: lets say thread1*

Now, what if caller want to know the status of the ***thread1***. Whether its completed or failed etc.

## 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.

```
public static void main(String args[]) {

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1, 1, 1, TimeUnit.HOURS, new ArrayBlockingQueue<>(10),
            Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

    //new thread will be created and it will perform the task
   Future<?> futureObj = poolExecutor.submit(() -> {
        System.out.println("this is the task, which thread will execute");
    });

   //caller is checking the status of the thread it created
    System.out.println(futureObj.isDone());
}
```

| S.No. | Method Available in Future Interface | Purpose |
|---|---|---|
| 1. | boolean cancel(boolean mayInterruptIfRunning) | • Attempts to cancel the execution of the task.<br>• Returns false, if task can not be cancelled (typically bcoz 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.<br>• 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.<br>• Throws 'TimeoutException' if timeout period finished and task is not yet completed. |

## Example:

```
public static void main(String args[]) {

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1, 1, 1, TimeUnit.HOURS, new ArrayBlockingQueue<>(10),
            Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

    Future<?> futureObj = poolExecutor.submit(() -> {
        try {
            Thread.sleep(7000);
            System.out.println("this is the task, which thread will execute");
        } catch (Exception e) {
        }
    });

    System.out.println("is Done: " + futureObj.isDone());
    try {
        futureObj.get(2, TimeUnit.SECONDS);
    } catch (TimeoutException e) {
        System.out.println("TimeoutException happened");
    }
    catch (Exception e) {
    }

    try {
        futureObj.get();
    } catch (Exception e) {
    }

    System.out.println("is Done: " + futureObj.isDone());
    System.out.println("is Cancelled: " + futureObj.isCancelled());
}
```

## 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:**

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

**Callable Interface:**

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

## Example:

```
public static void main(String args[]) {

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor( corePoolSize: 3, maximumPoolSize: 3, keepAliveTime: 1, TimeUnit.HOURS,
            new ArrayBlockingQueue<>( capacity: 10), Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

    //UseCase1
    Future<?> futureObject1 = poolExecutor.submit(() -> {
        System.out.println("Task1 with Runnable");
    });

    try {
        Object object = futureObject1.get();
        System.out.println(object == null);
    }catch (Exception e){
    }

    //UseCase2
    List<Integer> output = new ArrayList<>();
    Future<List<Integer>> futureObject2 = poolExecutor.submit(() -> {
        output.add(100);
        System.out.println("Task2 with Runnable and Return object");
    }, output);

    try {
        List<Integer> outputFromFutureObject2 = futureObject2.get();
        System.out.println(outputFromFutureObject2.get(0));
    } catch (Exception e) {
    }

    //UseCase3
    Future<List<Integer>> futureObject3 = poolExecutor.submit(() -> {
        System.out.println("Task3 with Callable");
        List<Integer> listObj = new ArrayList<>();
        listObj.add(200);
        return listObj;
    });

    try {
        List<Integer> outputFromFutureObject3 = futureObject3.get();
        System.out.println(outputFromFutureObject3.get(0));
    } catch (Exception e) {
    }
}
```

## 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 Aync 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.

```java
public class CompletableFutureExample {

    public static void main(String args[]) {

        try {
            ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor( corePoolSize: 1,  maximumPoolSize: 1,  keepAliveTime: 1,
                    TimeUnit.HOURS, new ArrayBlockingQueue<>( capacity: 10),
                    Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());


            CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(() -> {
                //"this is the task which need to be completed by thread";
                return "task completed";
            }, poolExecutor);

            System.out.println(asyncTask1.get());

        }catch (Exception e) {

        }
    }
}
```

2. *thenApply* & *thenApplyAsync*:

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

```
CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(() -> {
    //Task which thread need to execute
    return "Concept and ";
}, poolExecutor).thenApply((String val) -> {
    //functionality which can work on the result of previous async task
    return val + "Coding";
});
```

'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.

```
public class CompletableFutureExample {

    public static void main(String args[]) {

        try {
            ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor( corePoolSize: 1,  maximumPoolSize: 1,  keepAliveTime: 1,
                    TimeUnit.HOURS, new ArrayBlockingQueue<>( capacity: 10),
                    Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());


            CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(() -> {
                System.out.println("Thread Name which runs 'supplyAsync': " + Thread.currentThread().getName());
                return "Concept and ";
            }, poolExecutor).thenApplyAsync((String val) -> {
                System.out.println("Thread Name which runs 'thenApply': " + Thread.currentThread().getName());
                return val + "Coding";
            });

            System.out.println("Thread Name for 'after CF': " + Thread.currentThread().getName());
            // System.out.println(asyncTask1.get());

        }catch (Exception e) {

        }
    }
}
```

Output:

```
Thread Name which runs 'supplyAsync': pool-1-thread-1
Thread Name for 'after CF': main
Thread Name which runs 'thenApply': ForkJoinPool.commonPool-worker-9
```

3. *thenCompose and thenComposeAsync:*

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

```
CompletableFuture<String> asyncTask1 = CompletableFuture
        .supplyAsync(() -> {
            System.out.println("Thread Name which runs 'supplyAsync': " + Thread.currentThread().getName());
            return "Concept and ";
        }, poolExecutor)
        .thenCompose((String val) -> {
            return CompletableFuture.supplyAsync(() -> {
                System.out.println("Thread Name which runs 'thenCompose': " + Thread.currentThread().getName());
                return val + "Coding";
            });
        });
```

4. *thenAccept and thenAcceptAsync:*

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

```
CompletableFuture<Void> asyncTask1 = CompletableFuture
        .supplyAsync(() -> {
            System.out.println("Thread Name which runs 'supplyAsync': " + Thread.currentThread().getName());
            return "Concept and ";
        }, poolExecutor)
        .thenAccept((String val) -> System.out.println("All stages completed"));
```

*5. thenCombine and thenCombineAsync:*

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

```
CompletableFuture<Integer> asyncTask1 = CompletableFuture
        .supplyAsync(() -> {
            return 10;
        }, poolExecutor);

CompletableFuture<String> asyncTask2 = CompletableFuture
        .supplyAsync(() -> {
            return "k ";
        }, poolExecutor);


CompletableFuture<String> combinedFutureObj = asyncTask1.thenCombine(asyncTask2, (Integer val1, String val2) -> val1 + val2);
```