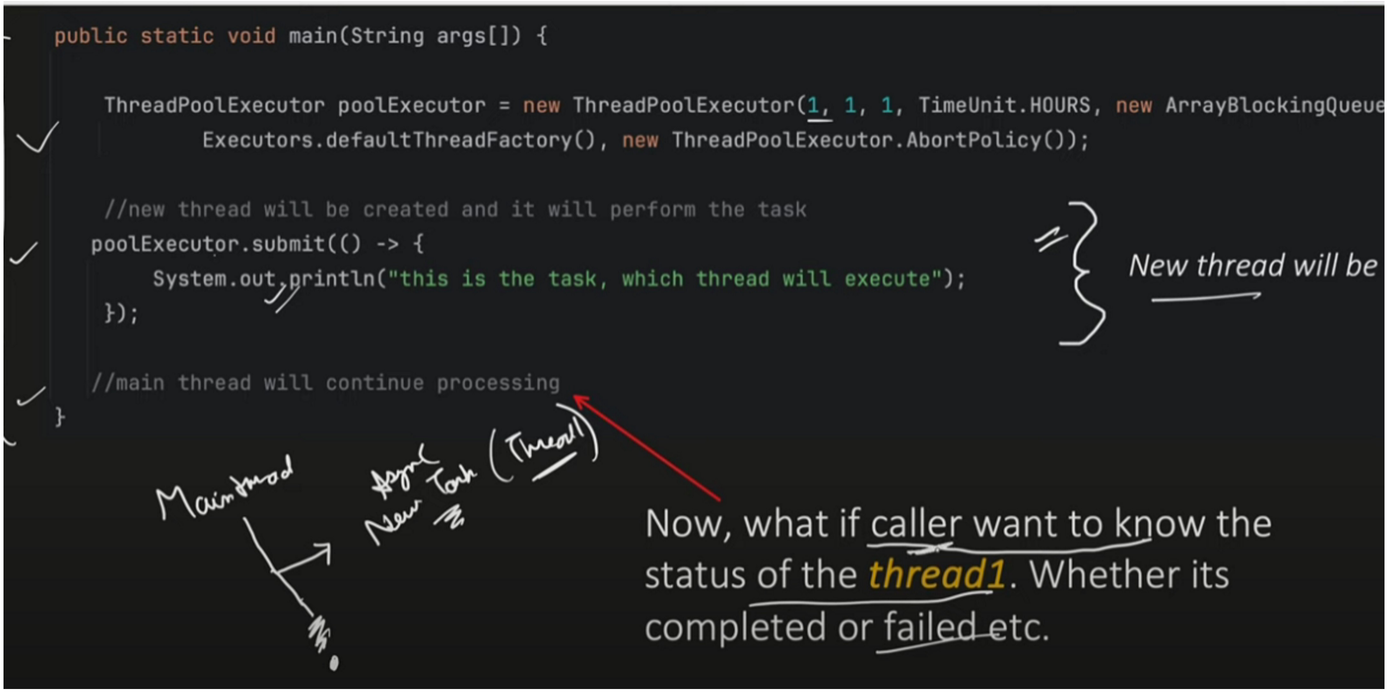
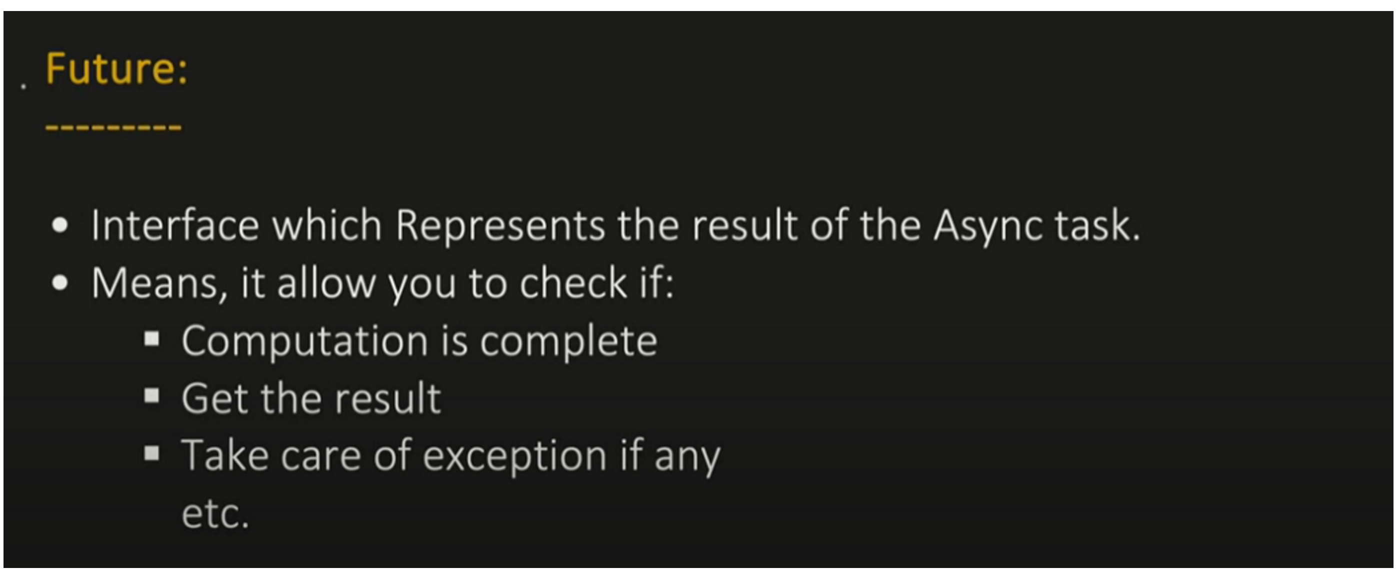
Lesson 7 : CompletableFuture Future Callable

Monday, May 6, 2024 10:05 PM



We know that threadPoolExecutor will do its job and main thread will continue to run. But what if we want to know the status of thread. As we have not taken any reference variable of poolExecutor.submit() thus we can't know the details if thread failed or completed.

Whenever we call submit, the return type is future.



```java
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());
}
```

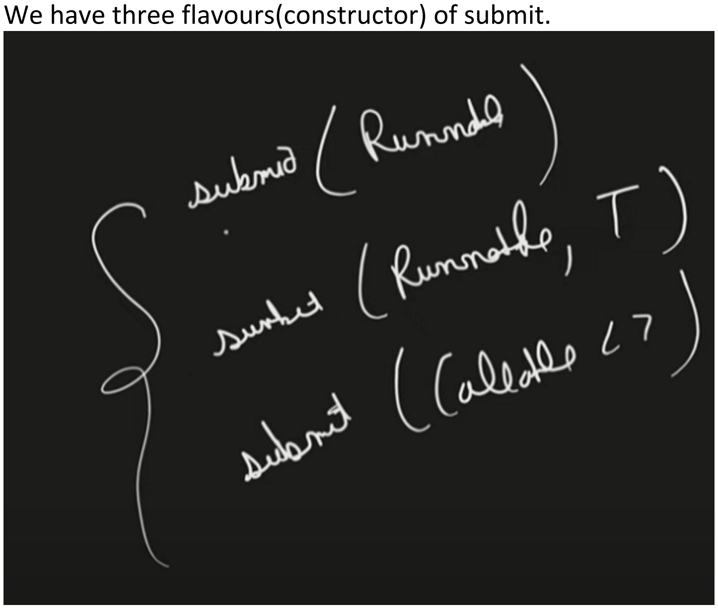
Methods available in Future Interface

1. **boolean cancel(boolean mayInterruptIfRunning)** - Attempts to cancel the execution of the task. 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 got completed.
3. **boolean isDone()** - Returns true if task is completed. Completion may be due to normal termination, an exception, or cancellation, all of these cases, this method will return true.
4. **get()** -- blocks the caller thread till completion of thread, after completion retreive the result if available.
5. **get(long timeout, TimeUnit unit) -** Waits for at most given time. Throws TimeoutException if timeout period finished and task is not completed yet.

What is <?> in Future ??

Callable

We have three flavours(constructor) of submit.



- Callable represents the task which need to be executed just like Runnable. But difference is :

1. Runnable do not have any return type.
2. Callable has the capability to return the value.

In executor.submit() we pass runnable/callable, if there is return then it is callable.

When we do .get() on a runnable future the output is always null. Eg :
Object obj = future.get(); here obj will always be null if future is of type runnable.

We have seen two flavours of submit where it accepts runnable and callable. Now look at third type submit(Runnable task, T result) :
Here we pass the reference to object that we wish to perform operation on and the same object is returned back to us.

Example we want to a list of integers using the executor service
Creating a custom runnable class

```java
import java.util.List;

public class MyRunnable implements Runnable{

    List<Integer> list;

    MyRunnable(List<Integer> list){
        this.list = list;
    }

    @Override
    public void run() {

        list.add(300);
        //It has to do some work
    }
}
```

List<Integer> output = new ArrayList<>();
Future<List<Integer>> futureObject = executor.submit(new MyRunnable(output),output);
Now we can get the result

int number = futureObject.get().get(0);

COMPLETABLE FUTURE aka big brother of Future

CompletableFuture:

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

There are 5 important methods in Completable Future

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

Just like submit() here we have supplyAsync. This also creates a new thread accepted as supplier. If executor is not provieded it uses the default forkjoin pool

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 processors.

Code example :

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

    try {
        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor( corePoolSize: 1,  maximumPoolSize: 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) {

    }
}
```

Other methods :

2. *thenApply* & *thenApplyAsync*:

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

Usage is below

```
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";
});
```

Point to be noted is that no new thread is created in thenApply(), the same thread used in supplyAsync once finished gets used to execute the thenApply() statements. Meanwhile main threads remains unblocked.

What if we don't want to hold the thread for thenApply() but want another thread to pick up then chaining. For that we use thenApplyAsync() method.

```
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";
});
```

3. *thenCompose and thenComposeAsync:*

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

thenCompose and thenComposeAsync are used to define ordering. It accepts result of previous chaining method and returns Completablefuture. Just that we use multiple chains then they are executed one after the other irrespective if we use thenCompose or thenComposeAsync

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.currentThr
            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.*

```java
CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {
    // Simulate a long-running task
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 10;
});

CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {
    // Simulate another long-running task
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 20;
});
```

The above two completableFuture can be combined as :
CompletableFuture<Integer> combinedFuture = future1.thenCombine(future2, (result1, result2) -> result1 + result2);