Multithreading 4 : CompletableFuture

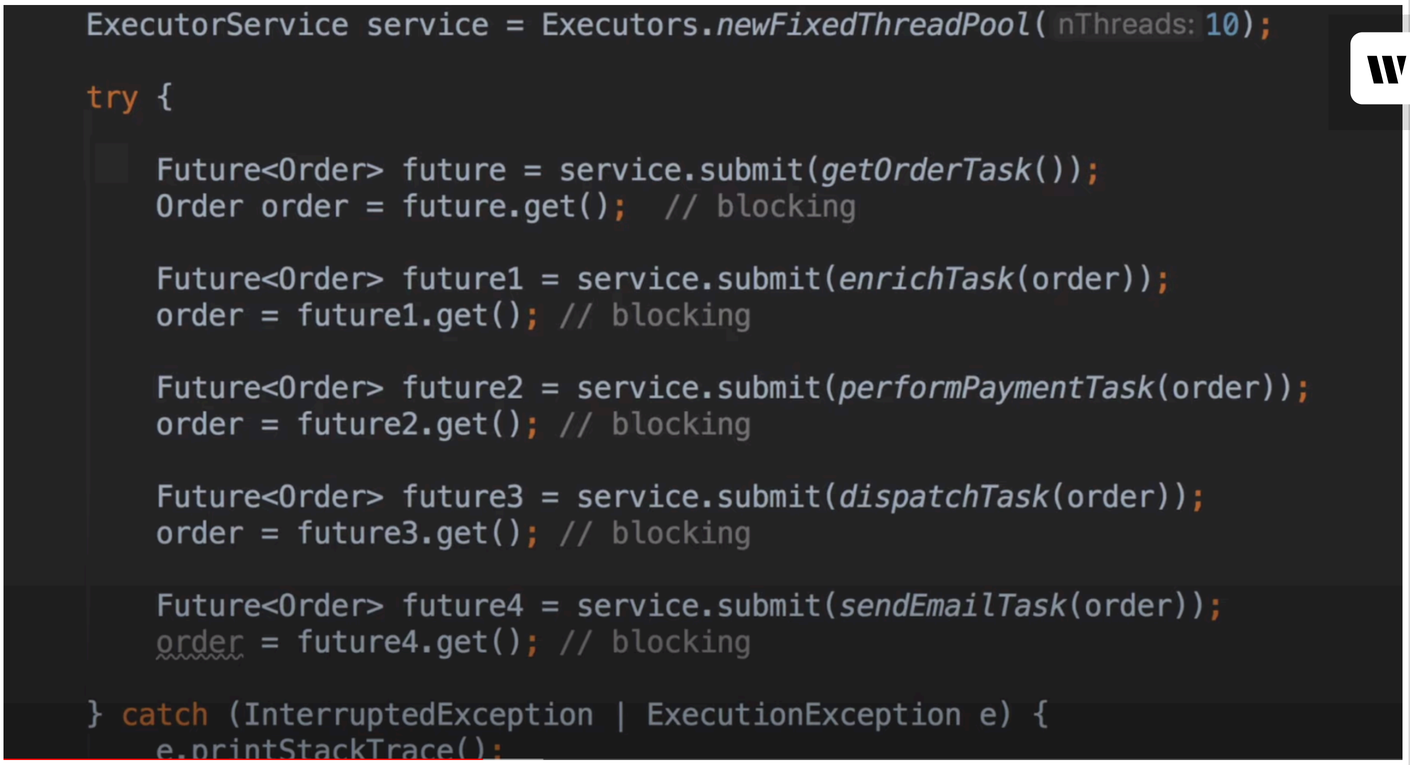
Perform possible asynchronous (non-blocking) computation and trigger-dependent computation which is also async handled.

Callable /Future: These work on a only one separate thread. We can't do further operation on it untill we get future.get which block main thread.

Eg: **For a order**

Fetch→ enrich → payment →dispatch → email

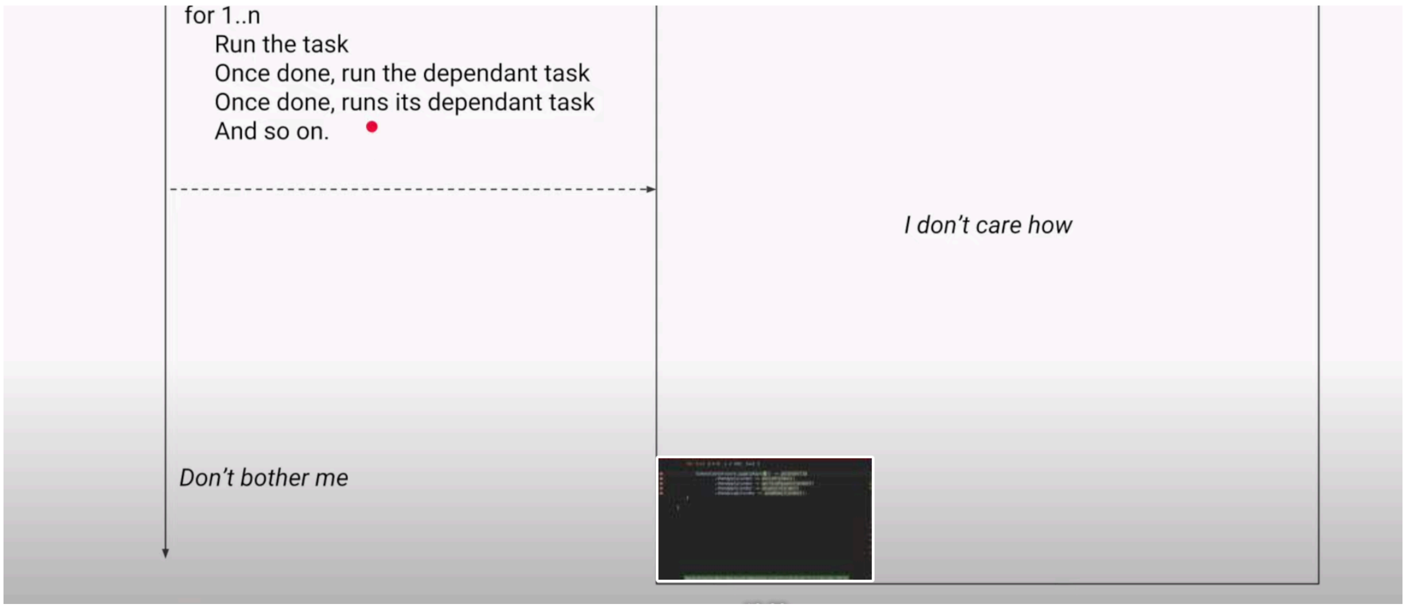
via future and callable: it would be like:

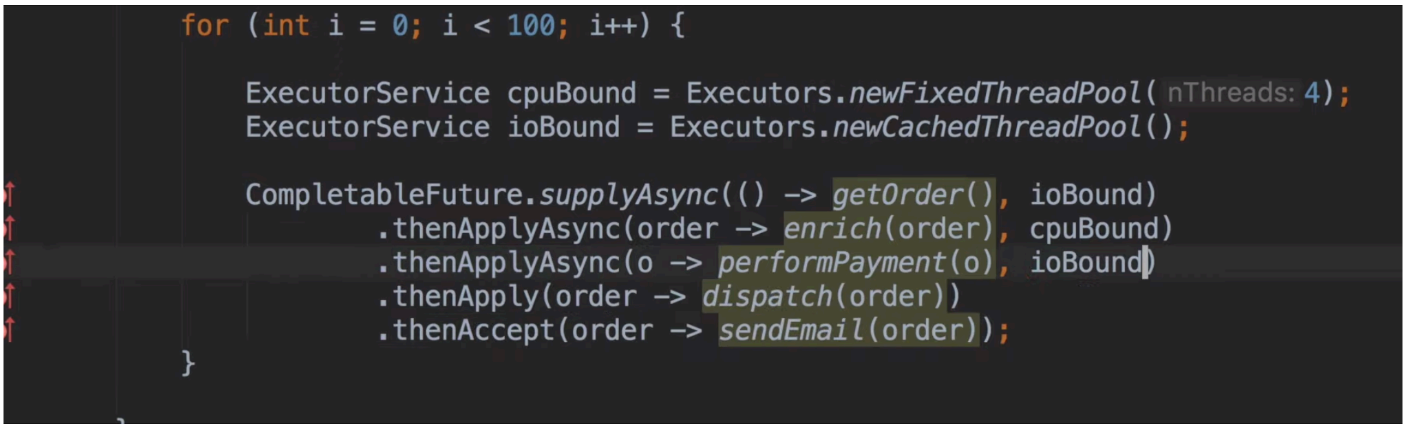


```
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    try {

        Future<Order> future = service.submit(getOrderTask());
        Order order = future.get();  // blocking

        Future<Order> future1 = service.submit(enrichTask(order));
        order = future1.get(); // blocking

        Future<Order> future2 = service.submit(performPaymentTask(order));
        order = future2.get(); // blocking

        Future<Order> future3 = service.submit(dispatchTask(order));
        order = future3.get(); // blocking

        Future<Order> future4 = service.submit(sendEmailTask(order));
        order = future4.get(); // blocking

    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
```

So main thread won't scale much.

Using CompletableFuture:


for 1..n
    Run the task
    Once done, run the dependant task
    Once done, runs its dependant task
    And so on.



```
for (int i = 0; i < 100; i++) {

    ExecutorService cpuBound = Executors.newFixedThreadPool( nThreads: 4);
    ExecutorService ioBound = Executors.newCachedThreadPool();

    CompletableFuture.supplyAsync(() -> getOrder(), ioBound)
            .thenApplyAsync(order -> enrich(order), cpuBound)
            .thenApplyAsync(o -> performPayment(o), ioBound)
            .thenApply(order -> dispatch(order))
            .thenAccept(order -> sendEmail(order));
}
```

Inside supplyAsnc, it used ForkJoinPool.commonPool() to provide threadPool.

We could add **exceptionally** to handle exceptions inside the async process.