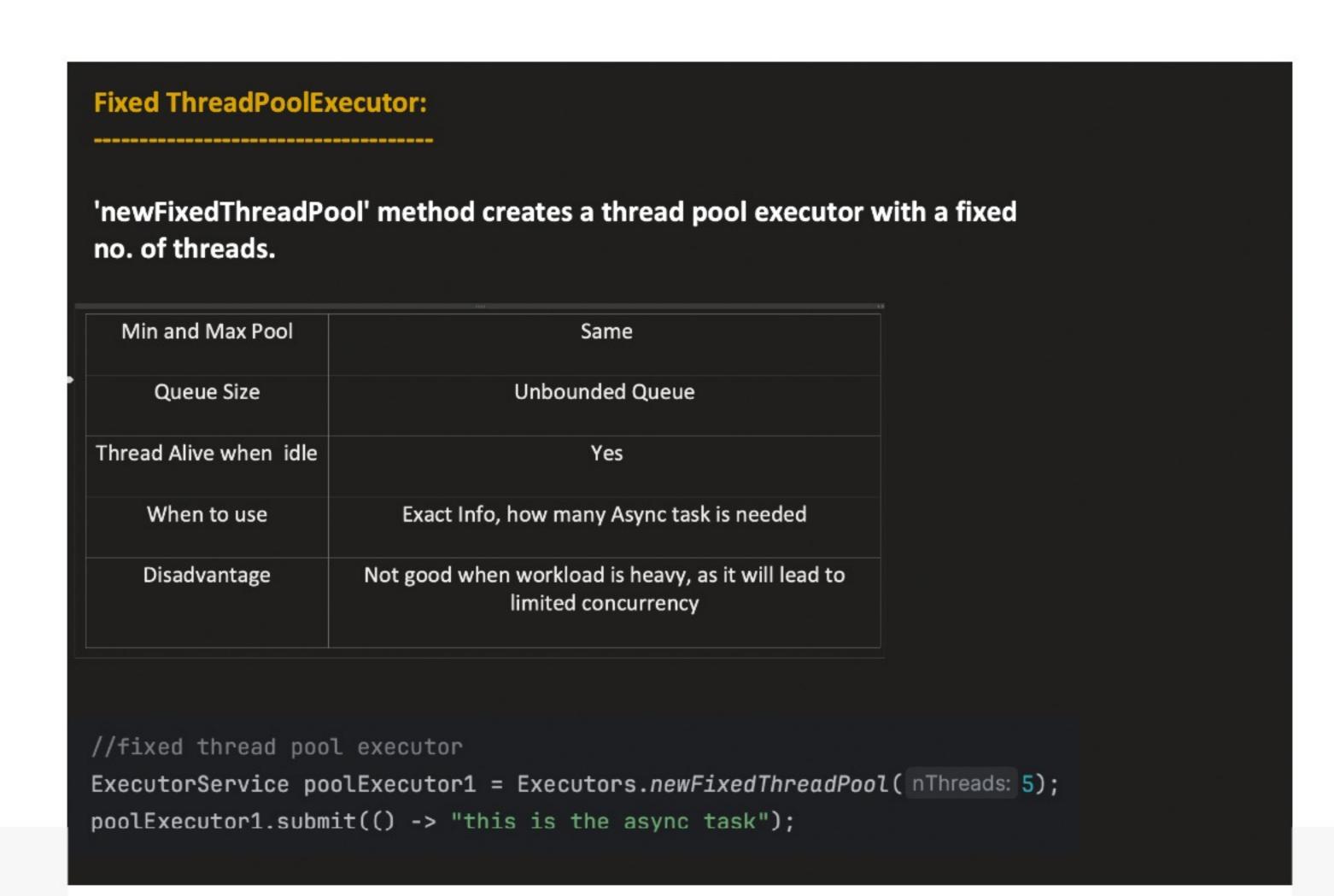
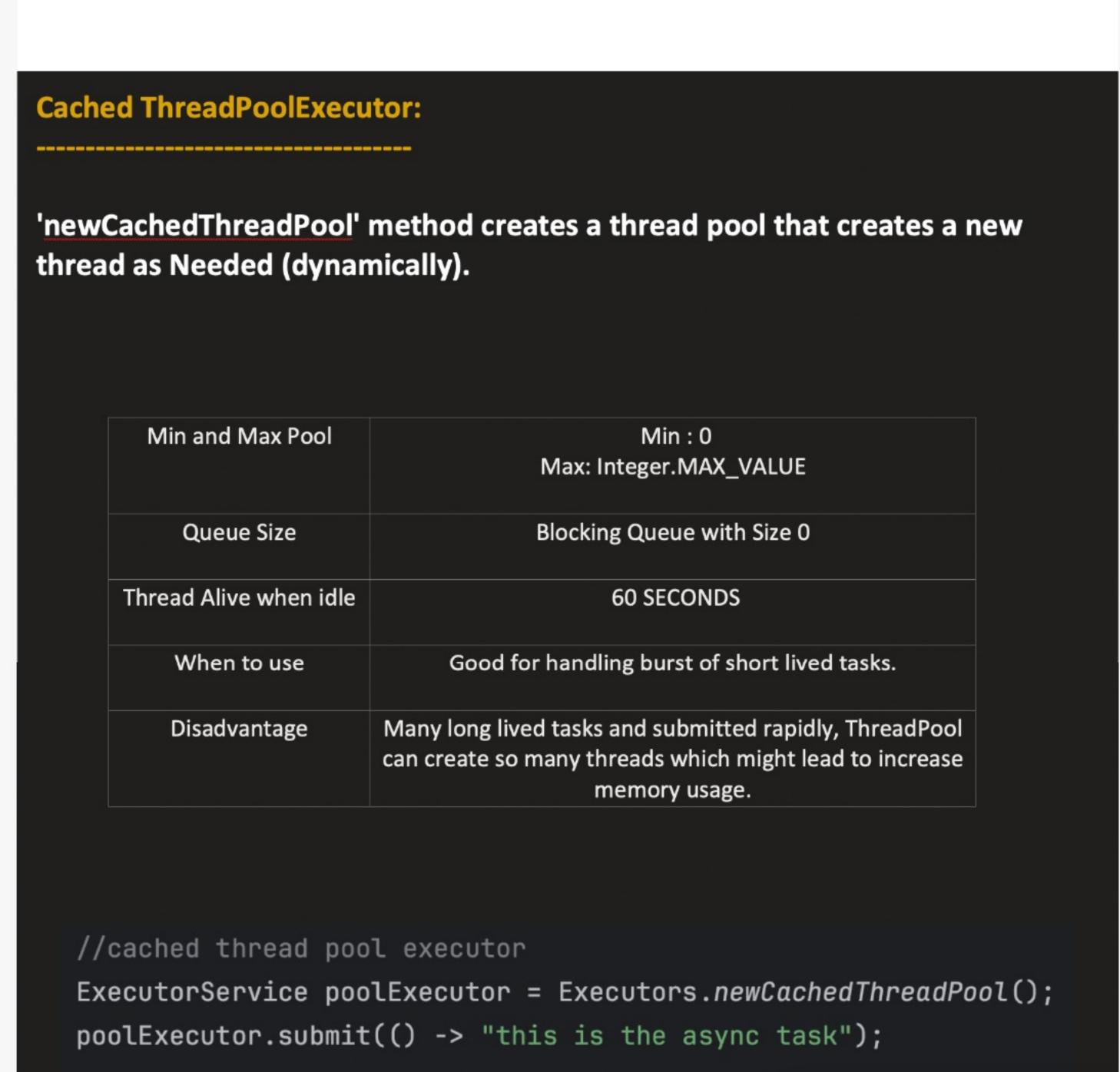
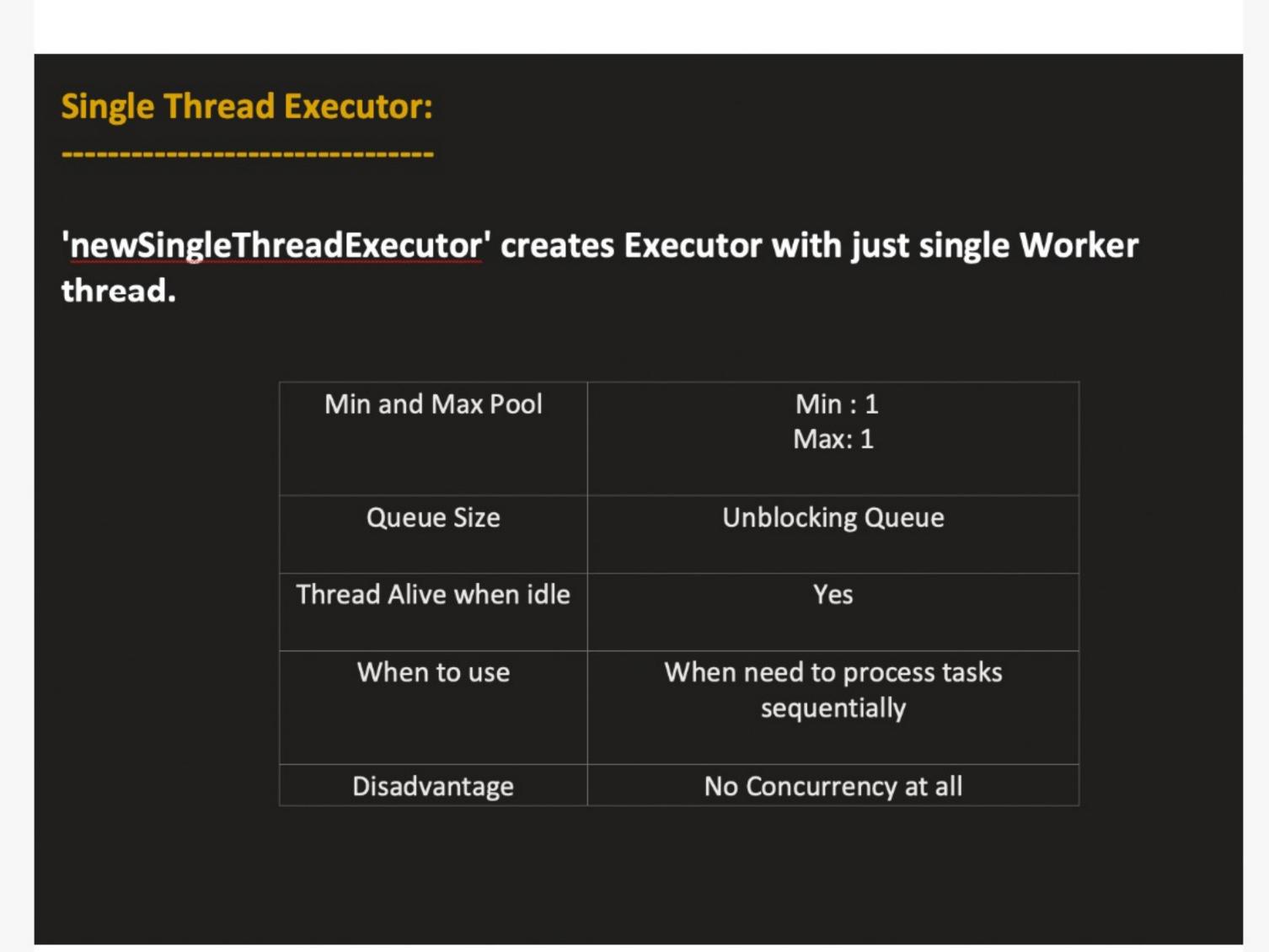
Java - Fork/Join Pool, Single, Fixed, CachedPool

"Concept && Coding" YT Video Notes



Report Abuse





```
WorkStealing Pool Executor:
- It creates a Fork-Join Pool Executor.
- Number of threads depends upon the Available Processors or we can specify
 in the parameter
- There are 2 queues:
    □ Submission Queue
    □ Work-Stealing Queue for each thread (it's a Dequeue)
- Steps:
    ☐ If all threads are busy, task would be placed in "Submission Queue". (or
       whenever we call submit() method, tasks goes into submission queue
       only)
    □ Lets say task1 picked by ThreadA. And if 2 subtasks created using fork()
       method. Subtask1 will be executed by ThreadA only and Subtask2 is
       put into the ThreadA work-stealing queue.
    ☐ If any other thread becomes free, and there is no task in Submission
       queue, it can "STEAL" the task from the other thread work-stealing
       queue.
- Task can be split into multiple small sub-tasks. For that Task should extend:
        ♦ RecursiveTask
        ♦ RecursiveAction
- We can create Fork-Join Pool using "newWorkStealingPool" method in
 ExecutorService.
 Or
 By calling ForkJoinPool.commonPool() method.
```

```
public class ExecutorsUtilityExample {
    public static void main(String args[]) {
        ForkJoinPool pool = ForkJoinPool.commonPool();
        Future<Integer> futureObj = pool.submit(new ComputeSumTask(0, 100));
        try {
            System.out.println(futureObj.get());
        }catch (Exception e){
       class ComputeSumTask extends RecursiveTask<Integer> {
           int start;
           int end;
           ComputeSumTask(int start, int end) {
               this.start = start;
               this.end = end;
           @Override
           protected Integer compute() {
               if (end - start <= 4) {
                   int totalSum = 0;
                   for (int i = start; i <= end; i++) {
                      totalSum += i;
                   return totalSum;
               } else {
                   //split the task
                   int mid = (start + end) / 2;
                   ComputeSumTask leftTask = new ComputeSumTask(start, mid);
                   ComputeSumTask rightTask = new ComputeSumTask(mid + 1, end);
                   // Fork the subtasks for parallel execution;
                   leftTask.fork();
                   rightTask.fork();
                   // Combine the results of subtasks
                       int leftResult = leftTask.join();
                       int rightResult = rightTask.join();
                   // Combine the results
                   return leftResult + rightResult;
```