```
===============================================================================================
                       ******************  EXERCISES  ******************
===============================================================================================
```

1. Let's do some thread-pool tuning! We will use CalculationTaskC as the task; but first update its call() method so that the 'for' loop inside it runs only for 10 times instead of 1000 and update the rest of the code accordingly. Now, create an ExecutorService and submit 1000 instances of the modified version of CalculationTaskC to it.

   (a) As ExecutorService, use a fixed-thread-pool of size 3. Note down the no. of threads and the time taken to execute all the tasks.
   (b) Increase the pool size by 3 and run again. Note down the no. of threads and the time taken to execute all the tasks.
   (c) Repeat (b). You will notice that the execution-time for executing all the 1000 tasks decreases a bit i.e. your throughput has increased! Continue repeating (b). At some point the execution-time will start increasing. Stop at this point!
   (d) Post the statistics in the following format on the discussion board:

```
    <format>
                         ====================================
                           # OF THREADS          TIME TAKEN
                         ====================================
                                 3               <25.543 s>
                                 6               <18.879 s>
                                 9               <12.799 s>
                                 .                    .
                                 .                    .
                                 n                    t
                         ====================================

                         Laptop Name/Model : <e.g. HP Probook>
                         Processor Make    : <e.g. Intel Core i5 vPro>
                         # of cores        : <e.g. 2>
                         RAM               : <e.g. 8 GB>
    </format>
```

2. Repeat Exercise-1 but use a cached-thread-pool instead. Obviously you will need to run it just once! Do note down the maximum no. threads created by the cached-thread-pool internally and the total time taken to execute all the tasks. Post the statistics too in the same format as Exercise-1.

3. Repeat Exercise-1 (with fixed-thread-pool only). This time vary the load - submit 5000 tasks. Post the statistics too.

4. Repeat Exercise-3 but use a cached-thread-pool (load remains the same i.e. 5000 tasks). Post the statistics too.

5. What are your observations and inferences from the data that you have collected for Exercises 1 to 4?

6. This exercise concerns naming an Executor thread that executes only one task for the whole of its life. I want to create a fixed-thread-pool with three threads. Each of these three threads will execute only one of the following tasks for the whole of its lifetime:

   - Thread-1 => Monitor a specific file for updates.
   - Thread-2 => Wait for and read the input from a specific port.
   - Thread-3 => Animate an object on the screen at random intervals

   What is the best way to name these threads?

7. Suppose you are working in Oracle and they are currently experimenting on implementing an Executors Framework (i.e this framework does not exist yet). You are asked to write some functionality that, somewhat, looks like this day's ExecutorService. So, you should write one or more classes that provide the following functionality:

   (a) Hold a pool of specifiable no. of threads internally.
   (b) Each thread should be capable of executing multiple tasks - one after the other i.e. the thread should not die after executing a task. Rather, it should pick up the next task from a queue and start executing it. If no task is available, then it should suspend till

a task becomes available.

(c) There should be a method, say submit(), that can accept Runnable tasks to be executed by the internal thread-pool.

(d) Lastly, there should be a method, say shutdown(), that should kill all the threads and shut the service down. All the existing tasks in the queue must first be executed by the service before the threads are killed. Once the shutdown() method is called, the service should not accept any further tasks and should throw an exception - ServiceClosedException!

8. You have seen many examples of scheduling tasks for repeated executions in the course. All those tasks were Runnables where one thread was scheduling the tasks while another thread was actually executing them. Now, for this exercise, you need to code so that you can "return" a value from your task to the calling thread; and that too - EVERY TIME it runs. Write using:

   (a) The Threads API
   (b) The Executors Framework