Executors are based on consumer-producer patterns. The tasks we produce for processing are consumed by threads. To better our understanding of how threads behave, imagine you are hired by a hedge fund on Wall Street and you are asked to design a method that can process client purchase orders as soon as possible. Let's see what are the possible ways to design this method.

Sequential Approach#

The method simply accepts an order and tries to execute it. The method blocks other requests till it has completed processing the current request.

```
    void receiveAndExecuteClientOrders() {

        while (true) {
            Order order = waitForNextOrder();
            order.execute();
        }
    }
```

You'll write the above code if you have never worked with concurrency. It sequentially processes each buy order and will not be *responsive* or have acceptable *throughput*.

Unbounded Thread Approach#

A newbie would fix the code above like so:

```
    void receiveAndExecuteClientOrdersBetter() {

        while (true) {
            final Order order = waitForNextOrder();

            Thread thread = new Thread(new Runnable() {

                public void run() {
                    order.execute();
                }
            });

            thread.start();
        }
    }
```

The above approach is an improvement over the sequential approach. The program now accepts an order and spawns off a thread to handle the order execution. The problem, however, is that now the application spawns off an unlimited number of threads. Creating threads without bound is not a wise approach for the following reasons:

- Thread creation and teardown isn't for free.
- Active threads consume memory even if they are idle. If there are less number of processors than threads then several of them will sit idle tying up memory.
- There is usually a limit imposed by JVM and the underlying OS on the number of threads that can be created.

Note that the above improvement may still make the application unresponsive. Imagine if several hundred requests are received between the time it takes for the method to receive an order request and spawn off a thread to deal with the request. In such a scenario, the method will end up with a growing backlog of requests and may cause the program to crash.

The next lesson introduces Threadpools which mitigate several of the issues we discussed here.