Thread Pools#

Thread pools in Java are implementations of the `Executor` interface or any of its sub-interfaces. Thread pools allow us to decouple task submission and execution. We have the option of exposing an executor's configuration while deploying an application or switching one executor for another seamlessly.

> A thread pool consists of homogenous worker threads that are assigned to execute tasks. Once a worker thread finishes a task, it is returned to the pool. Usually, thread pools are bound to a queue from which tasks are dequeued for execution by worker threads.

A thread pool can be tuned for the size of the threads it holds. A thread pool may also replace a thread if it dies of an unexpected exception. Using a thread pool immediately alleviates from the ails of manual creation of threads.

- There's no latency when a request is received and processed by a thread because no time is lost in creating a thread.
- The system will not go out of memory because threads are not created without any limits
- Fine tuning the thread pool will allow us to control the throughput of the system. We can have enough threads to keep all processors busy but not so many as to overwhelm the system.
- The application will degrade gracefully if the system is under load.

Below is the updated version of the stock order method using a thread pool.

```
    void receiveAndExecuteClientOrdersBest() {

        int expectedConcurrentOrders = 100;
        Executor executor = Executors.newFixedThreadPool(expectedConcurrentOrders);

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

            executor.execute(new Runnable() {

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

In the above code we have used the factory method exposed by the `Executors` class to get an instance of a thread pool. We discuss the different type of thread pools available in Java in the next section.