Types of Thread Pools

This lesson details the different types of thread pools available in the Java class library.

Java has preconfigured thread pool implementations that can be instantiated using the factory methods of the `Executors` class. The important ones are listed below:

- **newFixedThreadPool**: This type of pool has a fixed number of threads and any number of tasks can be submitted for execution. Once a thead finishes a task, it can reused to execute another task from the queue.
- **newSingleThreadExecutor**: This executor uses a single worker thread to take tasks off of queue and execute them. If the thread dies unexpectedly, then the executor will replace it with a new one.
- **newCachedThreadPool**: This pool will create new threads as required and use older ones when they become available. However, it'll terminate threads that remain idle for a certain configurable period of time to conserve memory. This pool can be a good choice for short-lived asynchronous tasks.
- **newScheduledThreadPool**: This pool can be used to execute tasks periodically or after a delay.

There is also another kind of pool which we'll only mention in passing as it's not widely used: `ForkJoinPool`. A prefconfigured version of it can be instantiated using the factory method `Executors.newWorkStealingPool()`. These pools are used for tasks which *fork* into smaller subtasks and then *join* results once the subtasks are finished to give an uber result. It's essentially the divide and conquer paradigm applied to tasks.

Using thread pools we are able to control the order in which a task is executed, the thread in which a task is executed, the maximum number of tasks that can be executed concurrently, maximum number of tasks that can be queued for execution, the selection criteria for rejecting tasks when the system is overloaded and finally actions to take before or after execution of tasks.

Executor Lifecycle

An executor has the following stages in its lify-cycle:

- Running
- Shutting Down
- Terminated

As mentioned earlier, JVM can't exit unless all non-daemon thread have terminated. Executors can be made to shutdown either abruptly or gracefully. When doing the former, the executor attempts to cancel all tasks in progress and doesn't work on any enqueued ones, whereas when doing the latter, the executor gives a chance for tasks already in execution to complete but also completes the enqueued tasks. If shutdown is initiated then the executor will refuse to accept new tasks and if any are submitted, they can be handled by providing a `RejectedExecutionHandler`.