Under light to moderate load, **thread per task** approach is an improvement over sequential execution. As long as the request arrival rate does not exceed the server's capacity to handle the requests, this approach offers better responsiveness.

Disadvantage of unbounded thread creation

For production use, however, the thread-per-task approach has some practical drawbacks, especially when a large number of threads may be created.

Thread lifecycle overhead:- Thread creation and teardown are not free. The actual overhead varies across platforms, but thread creation takes time, introducing lethargy into request processing, and requires some processing activity by JVM and OS. If requests are frequent and lightweight, as in most server applications, creating a new thread for each request can consume significant computing resources.

Resource consumption:- Active threads consume system resources, especially memory. When there are more runnable threads than available processors, threads sit idle. Having many idle threads can tie up a lot of memory

and having many threads competing for the CPUs can impose other performance costs as well.

Executors

If your program creates a large number of short-lived threads, then it should instead use a thread pool. A thread pool contains a number of idle threads that are ready to run. You give a Runnable to the pool , and one of the threads calls the run method. When the run method exits, the thread doesn't die but stays around to serve the next request.

Executors simplify concurrent programming by managing **Thread** objects for you. **Executors** allow you to manage the execution of asynchronous tasks without having to explicitly manage the lifecycle of threads.

We can use an **Executor** instead of explicitly creating **Thread** objects . An **ExecutorService** (an **Executor** with a service lifecycle – e.g. shutdown) knows how to build the appropriate context to execute **Runnable** objects.

An Executor implementation is likely to create threads for processing tasks. But the JVM can't exit until all the

threads have terminated, so failing to shutdown an Executor could prevent the JVM from exiting. Since Executors provide a service to the applications, they should be able to shutdown as well.

The call to **shutdown()** prevents new tasks from being submitted to that **Executor**. The current thread (e.g. **main thread**) will continue to run all tasks submitted before **shutdown()** was called. The program will exit as soon as all the tasks in the **Executor** finish.

Executors Factory Methods

a) newCachedThreadPool :- a cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bound on the size of the pool.

Grow and Shrink based on the demand.
No max limit.

b) newFixedThreadPool :- a fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected Exception).

Gradually grows upto a max limit.