Internal Working of Executor Service :

When we write below statements, ExecutorService e=Executors.newFixedThreadPool(3);

1. e.execute(runaable1);
2. e.execute(runaable2);
3. e.execute(runaable3);
4. e.execute(runaable4);
5. e.execute(runaable5);

till 3 execute methods ,three threads will be created,when 4th execute method will be called,no new thread will be created but work will be waiting for a thread to be free.

I do not understand this point "no new thread will be created but work will be waiting for a thread to be free." what i think when runnable1 will be given to first created thread,once runnable1's run method will be finished,Thread1's run will also be finished,thread1 will not be able to call run method of runnable4. So, how java manages to execute 5 Runnable with just 3 threads.

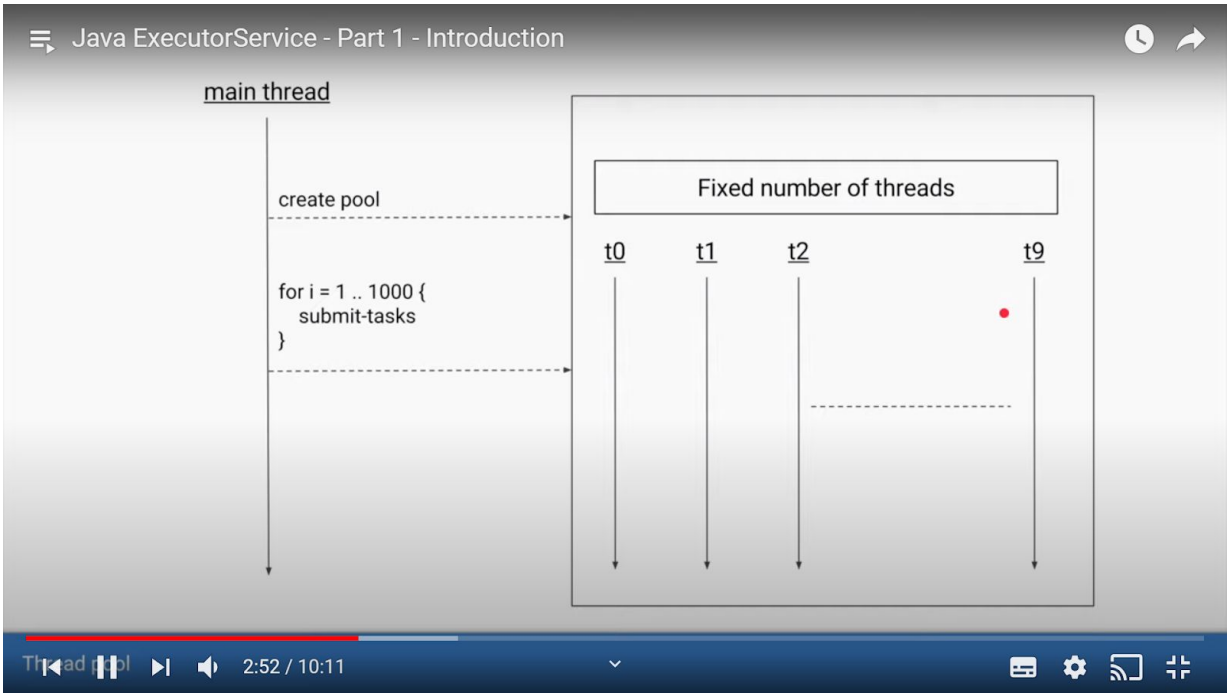
FYI: Here is a very simple implementation of a thread pool.

```
class MyThreadPool implements java.util.concurrent.Executor { private final
java.util.concurrent.BlockingQueue<Runnable> queue; public MyThreadPool(int
numThreads) { queue = new java.util.concurrent.LinkedBlockingQueue<>(); for (int
i=0 ; i<numThreads ; i++) { new Thread(new Runnable(){ @Override public void run()
{ while(true) { queue.take().run(); } } }).start(); } } @Override public void
execute(Runnable command) { queue.put(command); } }
```

This won't compile because I didn't deal with InterruptedException, and in a *real* thread pool, you would also want to deal with exceptions that might be thrown by the given `command`, but it should give you a rough idea of what a thread pool does.

It creates a queue and an arbitrary number of *worker threads*. The worker threads compete with one another to consume *commands* from the queue. The queue is a `BlockingQueue`, so the workers all sleep whenever the queue is empty.

Other threads may produce new commands (i.e., `Runnable` objects), and call the `execute(command)` method to put the new commands in the queue. The commands will be performed (i.e., their run methods will be called) by the worker threads in approximately* the same order that they were enqueued.



When you execute `ExecutorService e=Executors.newFixedThreadPool(3);` a thread pool will be created with 3 threads inside it. And these 3 threads will be used to execute any task executed on you `e` object.

When you try to execute tasks through `ExecutorService` then it will get added to a tasks queue, if number of threads in pool are greater than number of tasks, then as soon as some task will come, some free thread in the pool will be picked and used to execute task.

The tasks are added to a Blocking Queue. All the tasks submitted by the executor are added to a Blocking Queue internally, and the waiting threads keep on picking up the tasks from the blocking queue, and when they finish up the execution, they keep on picking up the next tasks from the queue.

It uses a blocking queue to keep tasks because the data structure should be thread safe as multiple threads are going to access it and pop tasks from it. Hence, blocking queue is ideal.

Types of Queue used :

Implementation of Blocking queue can vary based on the type of executor used, like LinkedBlockingQueue, ArrayBlockingQueue.

Pool	Queue Type	Why?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store all tasks. *Note: Since queue can never become full, new threads are never created.*
SingleThreadExecutor	LinkedBlockingQueue	
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Synchronous queue is a queue with single slot
ScheduledThreadPool	DelayedWorkQueue	Special queue that deals with schedules/time-delays
Custom	ArrayBlockingQueue	Bounded queue to store the tasks. If queue gets full, new thread is created (as long as count is less than maxPoolSize).

Queue Types

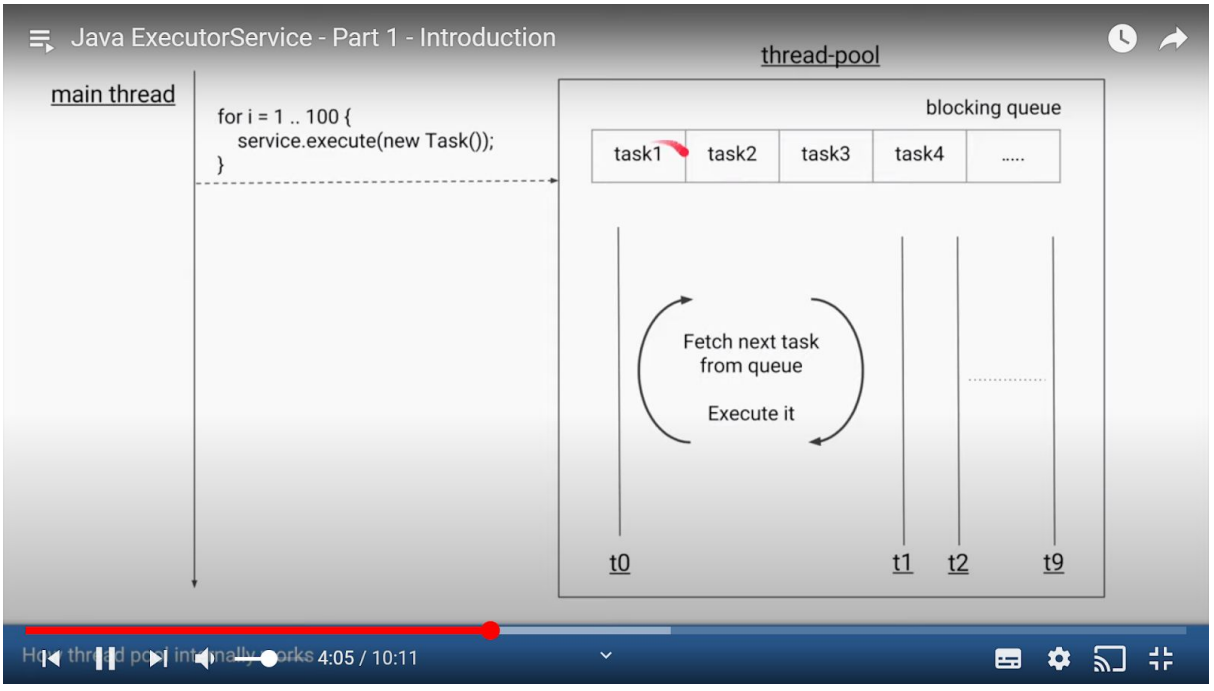
FixedThreadPool and SingleThreadExecutor use LinkedBlockingQueue because this queue has dynamic size, while ArrayBlockingQueue has fixed size. Since in these types of executors, the size of threads is fixed, hence the no of tasks can keep on increasing in the queue if the available threads are busy in execution. Hence, we need a queue which can dynamically increase, hence LinkedBlockingQueue.

CachedThreadPool does not need a lot of storage space for the tasks. Infact, the number of threads in cached thread pool dynamically increases when a task is added to the queue.

When a task is added and a thread is available to pick the task up, the it picks it up and executes it. But if there is no thread available to pick the task and all are busy in execution, then a new thread is created to execute the newly added task. Hence, there is only a single block of storage required because every time a task is added, it will be picked up by a thread. Synchronous queue is a type of such queue.

To maintain memory in cached thread pool, it keeps on terminating the threads which have finished execution and are idle for a certain time, to release memory, because otherwise there will be thousands of threads if there are thousands of tasks.

Scheduled Thread Pool uses a DelayedWorkQueue, because this thread pool can be required to execute tasks based on time intervals, so this type of queue is used.



When you use a singlethreadtaskexecutor, then there is only one thread in the thread pool, which is responsible for executing all the tasks. By using single thread task executor, we can make the execution of tasks sequential, because all the tasks added to the queue will be picked up from the queue in sequential order only by that single thread, and when the execution of one task is finished, the next one will be finished.

If we use many threads, then we do not know the order because any of the threads can pick up any task and anyone can be finished earlier or later, but with single thread taske executor we know it is going to be sequential.

When number of tasks become greater than number of threads in pool, then they will be added in pipleline in the queue, and as soon as some thread as finished execution, that thread will be used to execute task from the queue pipeline.

Thread pooling:

The thread which has just finished will not terminate. Please note that it is a thread pool, so threads are pooled, which means that they will not terminate (typically, until you have some timeout or some other mechanism) but will go back in pool so that they can be reused. And that's the reason your 4 and 5th runnable will be executed through those pooled threads.

"no new thread will be created but work will be waiting for a thread to be free."

This is same as discussed above that then they put be waiting in the queue, and as soon as some thread is done with the task execution, waiting tasks from the queue will be pooled and be executed from the free thread, and this will happen until all the tasks from the queue is cleaned up.

No new thread will be created because you have used `newFixedThreadPool`, so a fixed thread pool of 3 threads is created and they only will be used to process all the tasks requested coming to that particular instance of `ExecutorService` and in your case it is `e`.

Good videos :
https://www.youtube.com/watch?v=6Oo-9Can3H8&list=PLhfHPmPYPPRk6yMrcbfafFGSbE2EPK_A6&index=5