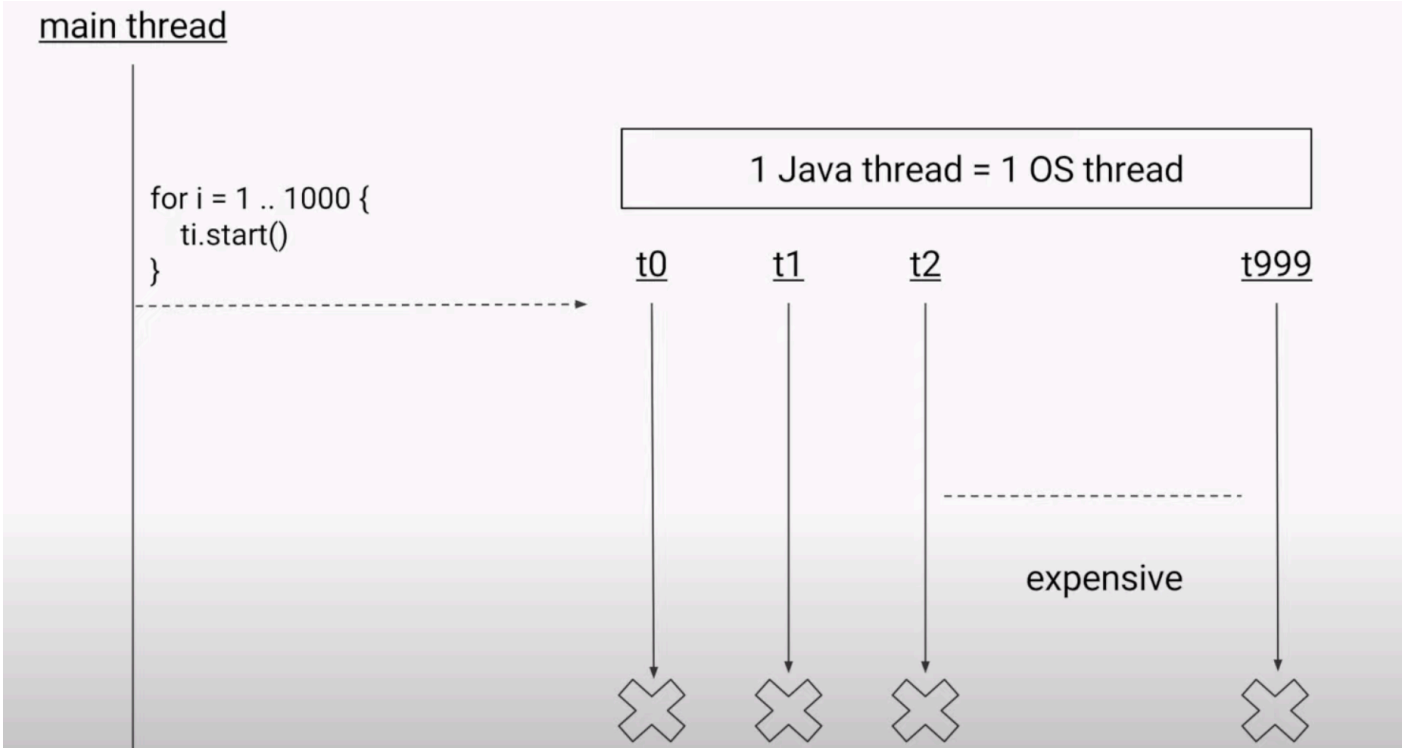
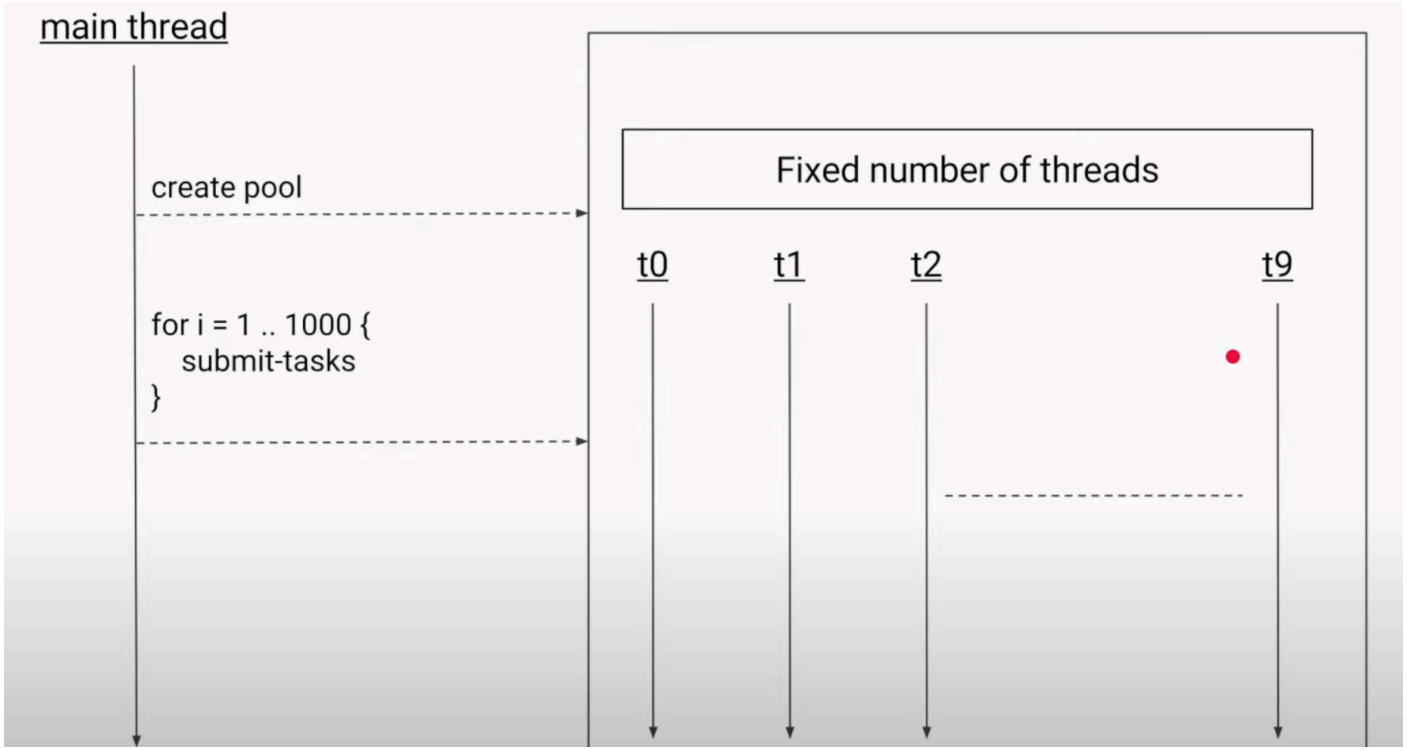
Multithreading 3: Executor service

Executor service:

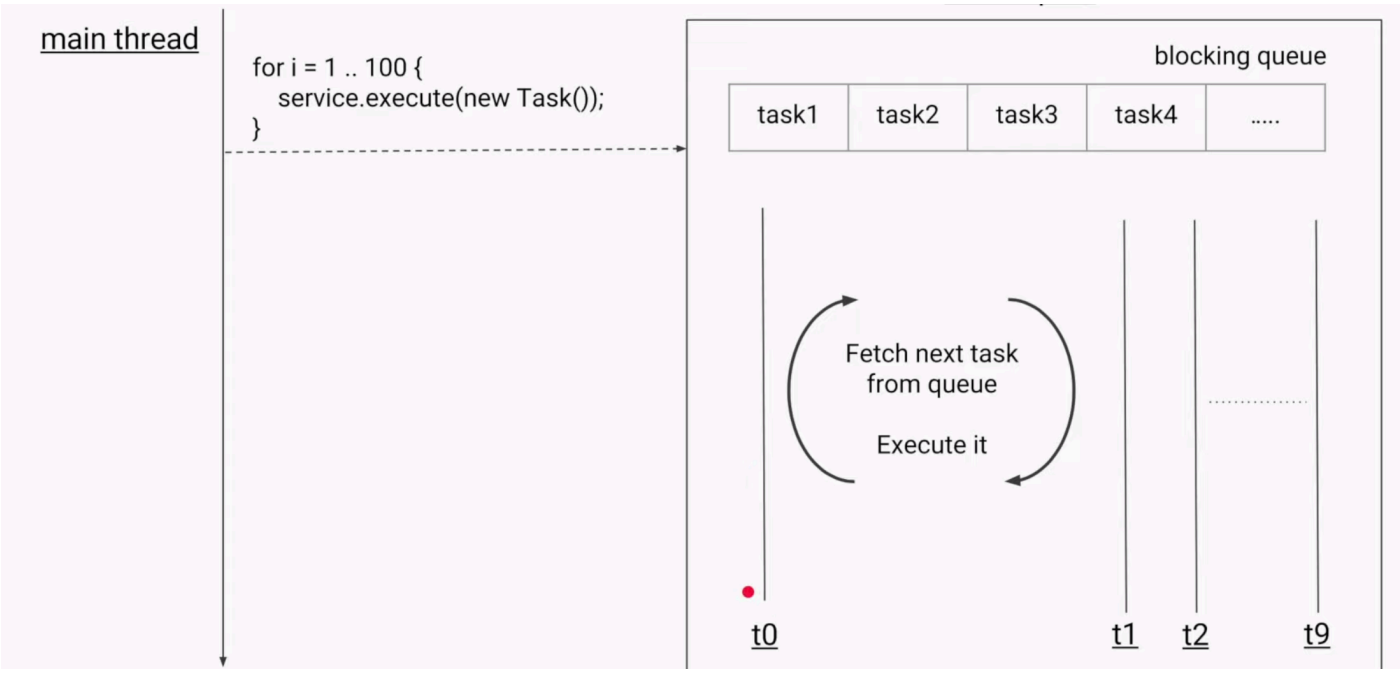
```
public static void main(String[] args) {

    for (int i = 0; i < 10; i++) {
        Thread thread = new Thread(new Task());
        thread.start();
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName(
    }
}
```



After implementing thread pool:




```java
public static void main(String[] args) {

    // create the pool
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new Task());
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName()
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getNa
    }
}
```



Also we want a queue which could work concurrently and thread safe so option is Blocking queue.

Ideal Pool Size:

It depend on type of task which we want to execute.

CPU 4 core means at a time max 4 trad can run and execute.

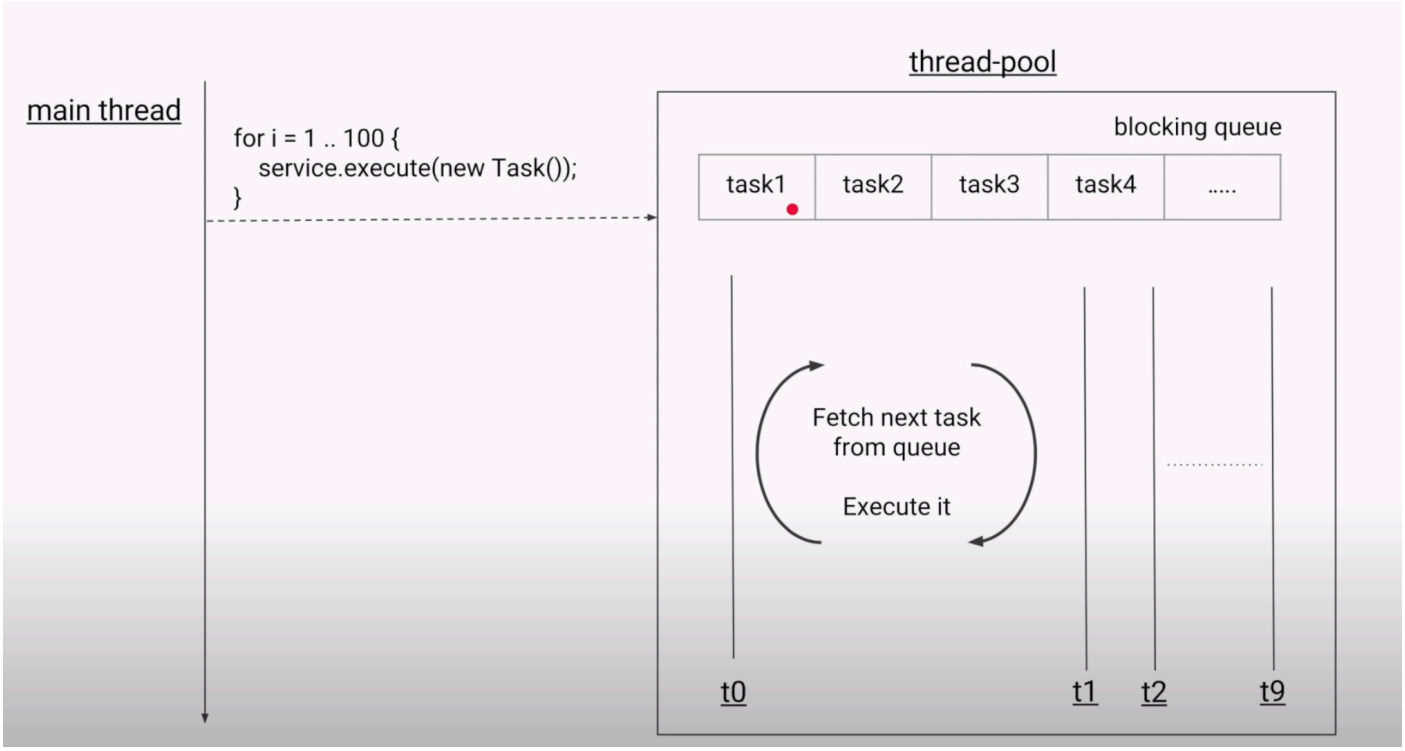
Task Type	Ideal pool size	Considerations
CPU intensive	CPU Core count	How many other applications (or other executors/threads) are running on the same CPU.
IO intensive	High	Exact number will depend on rate of task submissions and average task wait time. Too many threads will increase memory consumption too.

Thread pools:

1. Fixed thread pool
2. Cache Thread pool

3. Scheduled thread pool
4. Single Threaded executor

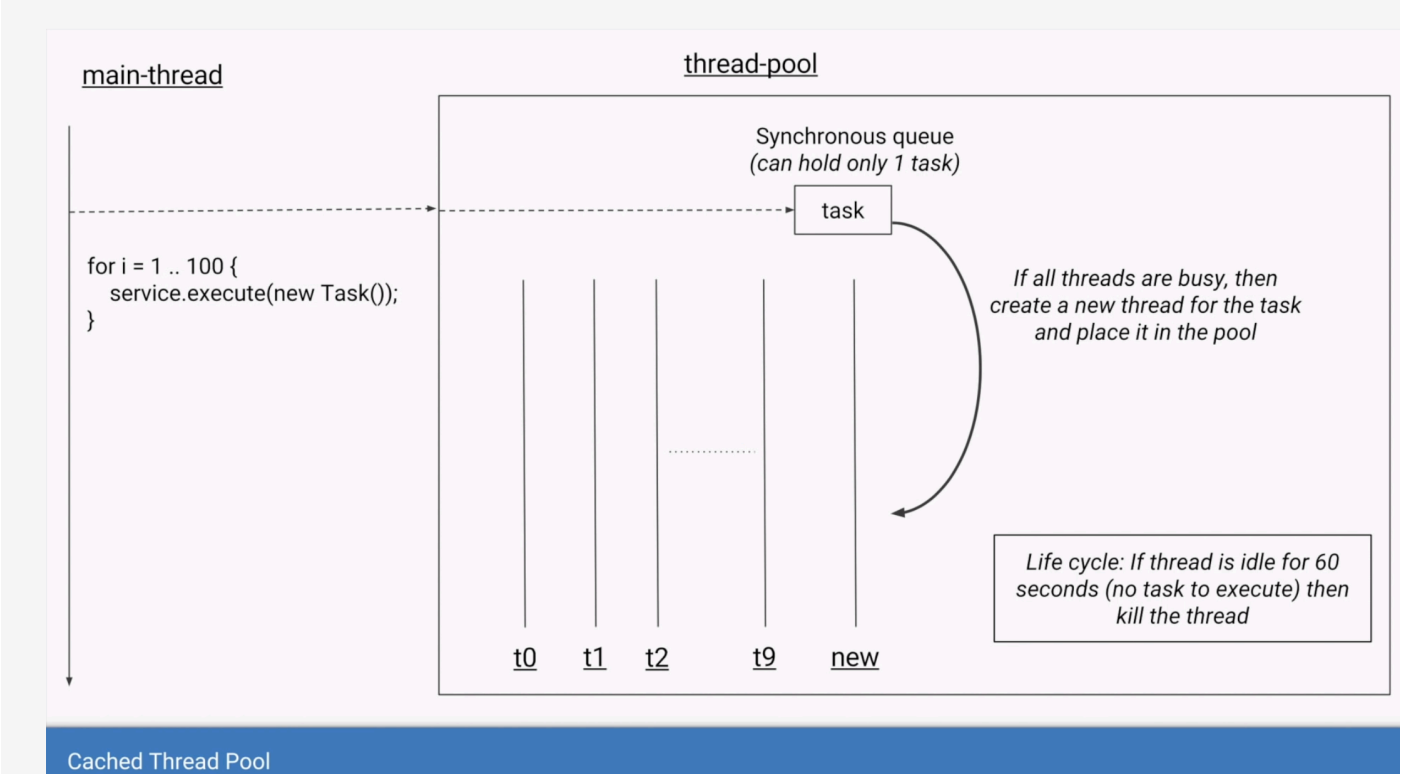
Fixed thread pool:



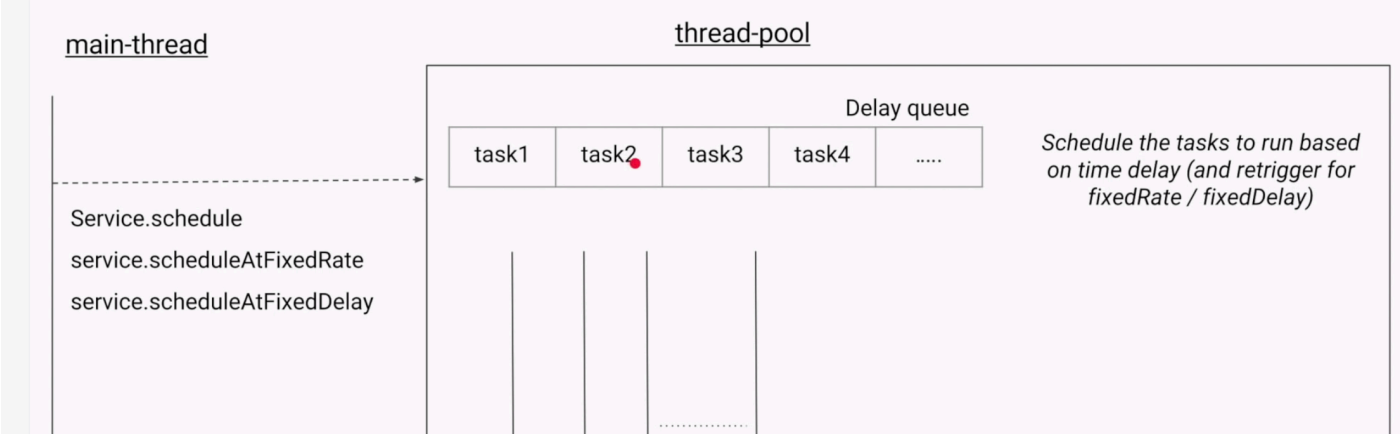

```
public static void main(String[] args) {

    // create the pool
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new Task());
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName()
    }
}
```

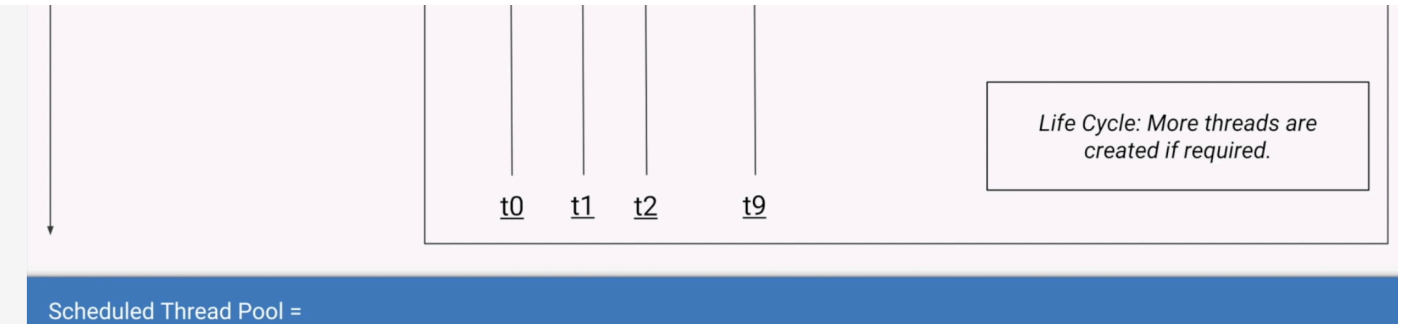
Cache thread pool:

Use Synchronus queue: Can have only 1 task at a time



Scheduled thread pool:





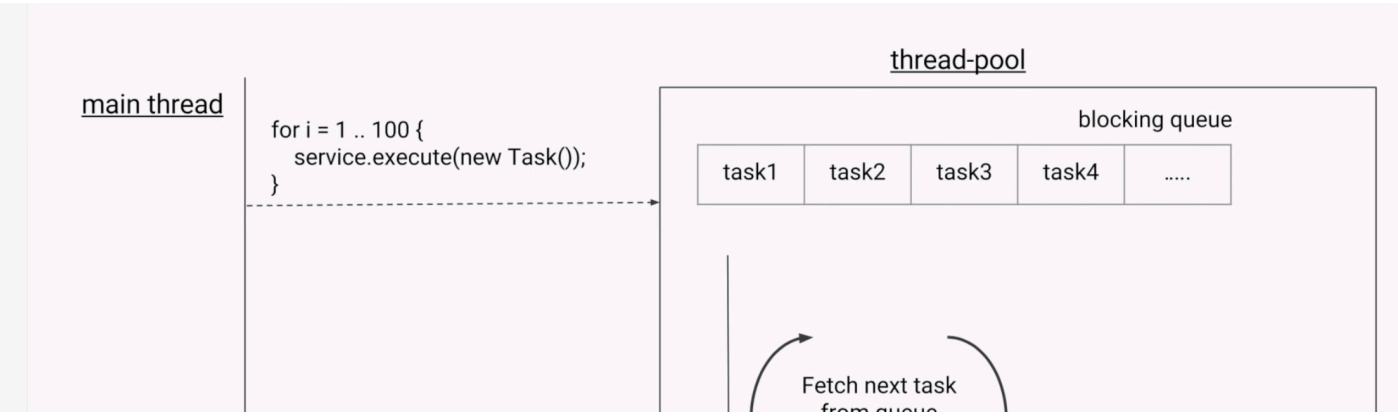
Scheduled Thread Pool =

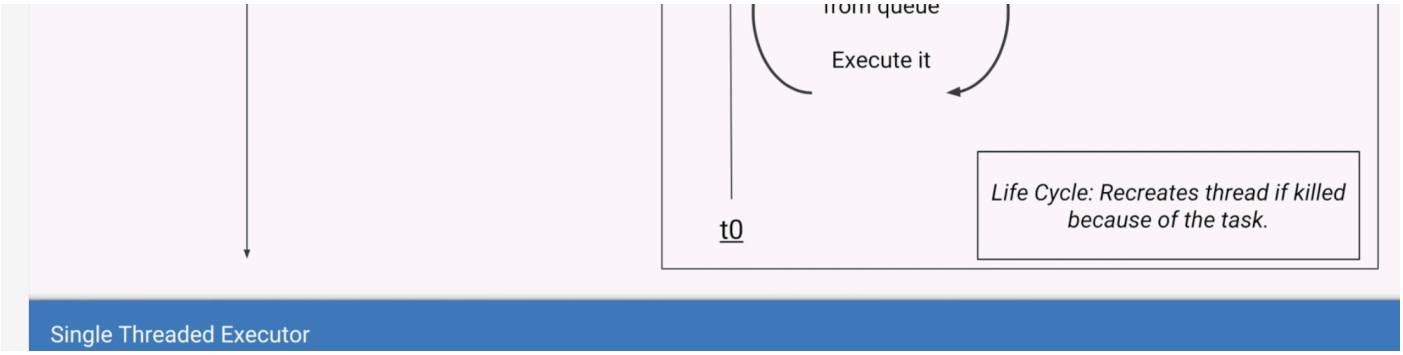
```
blic static void main(String[] args) {

  // for scheduling of tasks
  ScheduledExecutorService service = Executors.newScheduledThreadPool( corePoolSize: 10);

  // task to run after 10 second delay
  service.schedule(new Task(), delay: 10, SECONDS);

  // task to run repeatedly every 10 seconds
  service.scheduleAtFixedRate(new Task(), initialDelay: 15, period: 10, SECONDS);

  // task to run repeatedly 10 seconds after previous task completes
  service.scheduleWithFixedDelay(new Task(), initialDelay: 15, delay: 10, TimeUnit.SECONDS);


atic class Task implements Runnable {
  public void run() {
      // task that needs to run
      // based on schedule
  }
```

Scheduled Executor

Single Threaded executor:

It is similar to fixed thread pool with no of thread as 1. This is used when we want to make sure that always task 1 will execute before task 2.

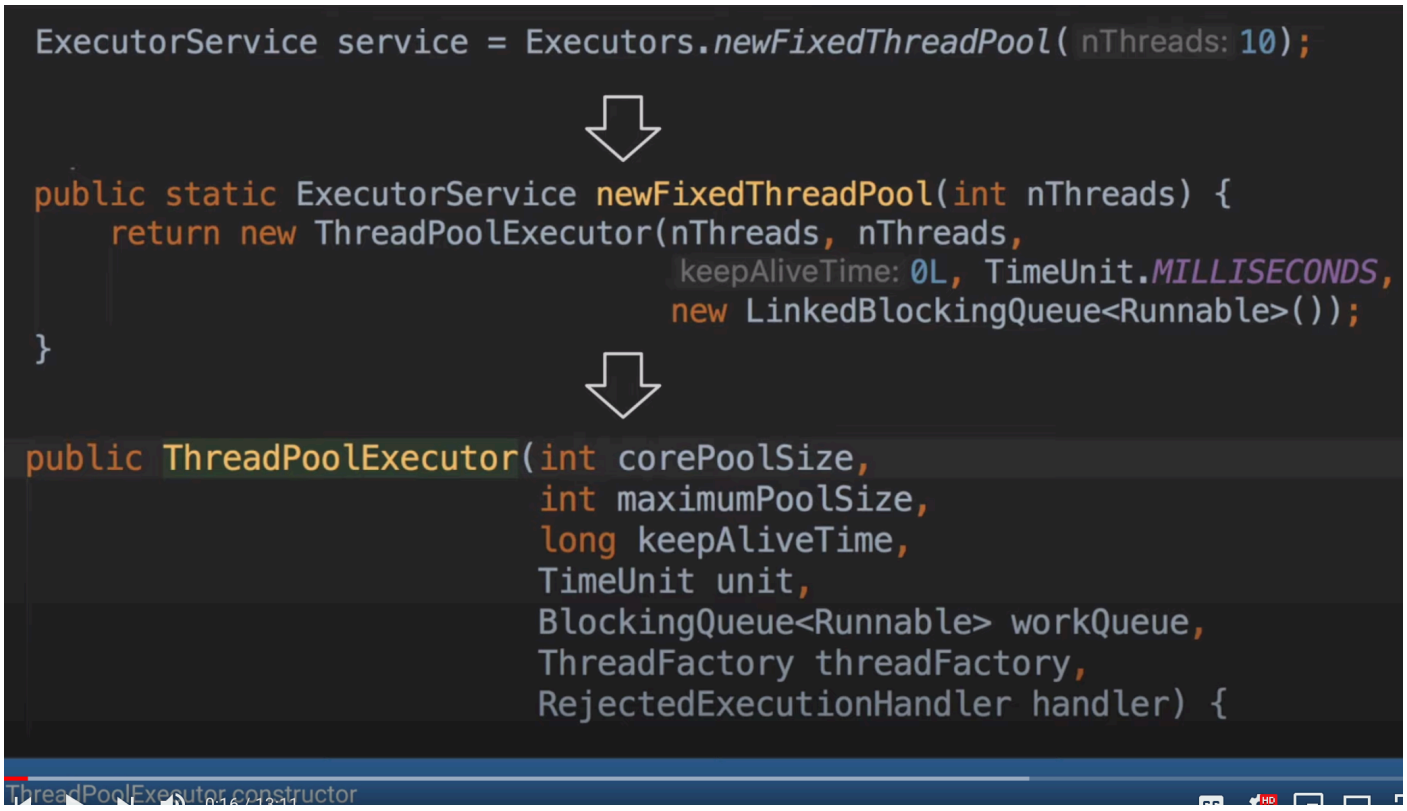




Constructor and life cycle:

1. Pool size changes
2. Queue size
3. Task rejection
4. Life cycle methods

Thread pool executor constructor:



Fixed thread pool also called internally ThreadPoolExecutor.

Parameter	FixedThreadPool	CachedThreadPool	ScheduledThreadPool	SingleThreaded
corePoolSize	constructor-arg	0	constructor-arg	1
maxPoolSize	same as corePoolSize	Integer.MAX_VALUE	Integer.MAX_VALUE	1
keepAliveTime	0 seconds	60 seconds	60 seconds	0 seconds

Note: Core pool threads are never killed unless allowCoreThreadTimeOut(boolean value) is set to true.

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).

CALLABLE AND FUTURE:

In case of runnable it does not return a value.

```
public class _12 {
```

```
    public static void main(String[] args) {

        // create the pool
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

        // submit the tasks for execution
        for (int i = 0; i < 100; i++) {
            service.execute(new Task());
        }
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }

    static class Task implements Runnable {
        public void run() {
            System.out.println("Thread Name: " + Thread.currentThread().getName());
            return 3;
        }
```

But this could be possible via **Callable**.

Eg:

```
Task implements Callable<Integer>{
public Integer call(){
    return 3;
    }
}
```

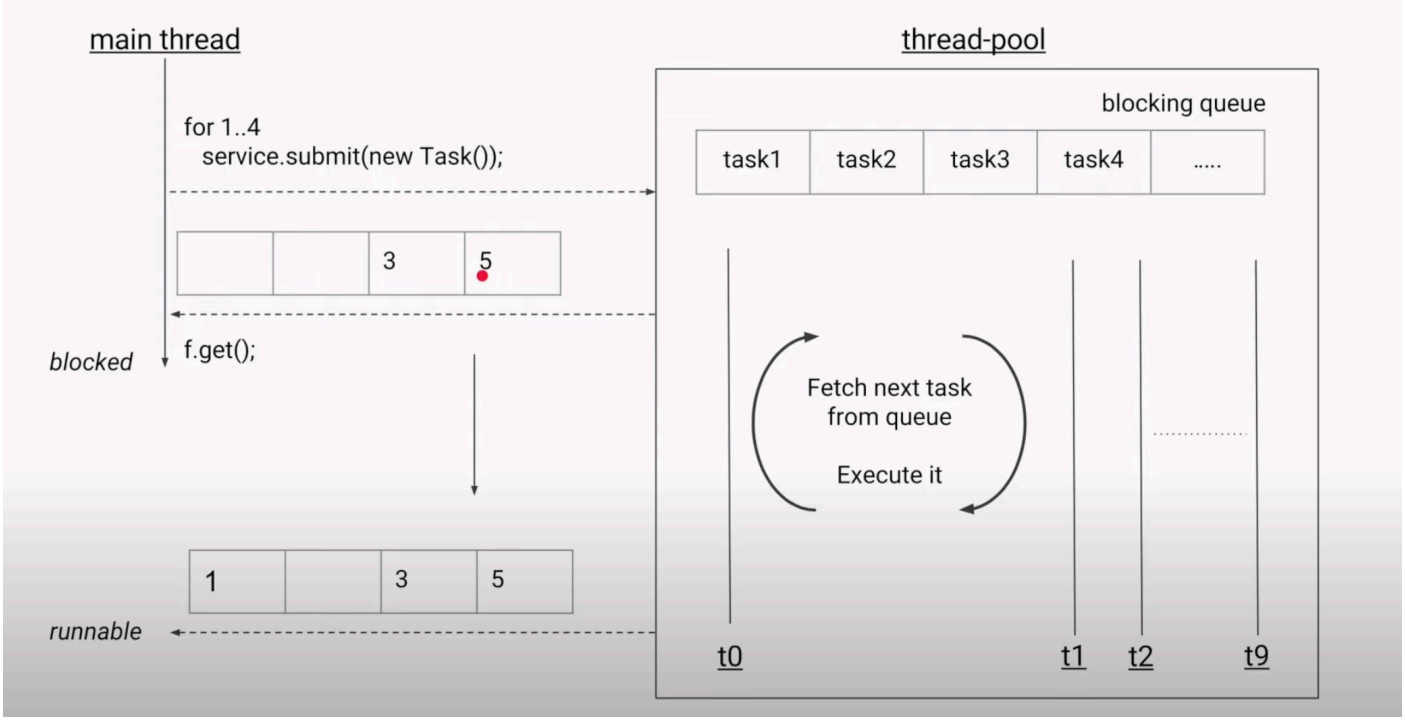
Runnable —> run method-- > execute

Callable —> call method → submit

Future is like the value will arrive sometime in the future.

```
    public static void main(String[] args) throws ExecutionException,
InterruptedException {

        // create the pool
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

        // submit the tasks for execution
        Future<Integer> future = service.submit(new Task());

        // perform some unrelated operations
```

```
        // 1 sec
        Integer result = future.get(); // blocking

        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }

    static class Task implements Callable<Integer> {
        @Override
        public Integer call() throws Exception {
            Thread.sleep( millis: 3000);
            return new Random().nextInt();
        }
```



After Calling **future.get()** →Main thread will go in blocking state and Once we will get value of future then only main thread proceed.

To solve this problem we could introduce Timeout with get call.

eg: future.get(timeout, Timeunit.SECONDS);

For cancelling the task.

future.cancel()

future.isCancel()