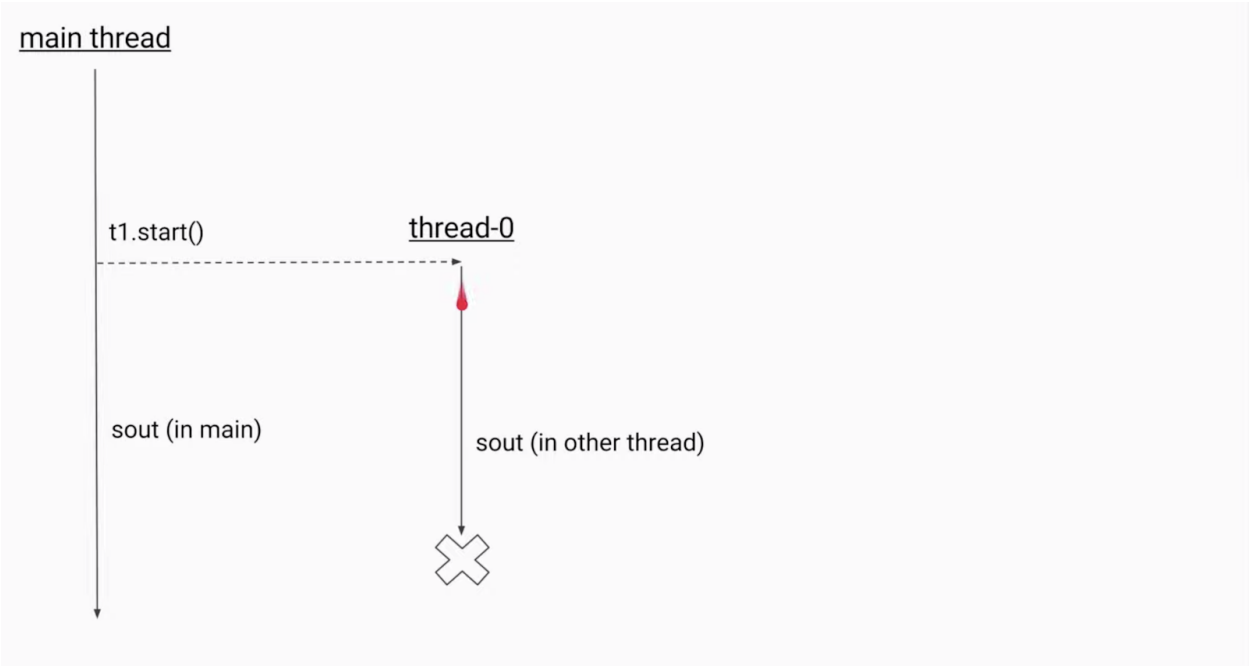
THREAD POOLS , EXECUTOR SERVICE , EXECUTOR COMPLETION SERVICE IN JAVA

In Java, it is quite easy to run a task or method asynchronously. That is if you have a main method which has a main thread, you can always have another thread and run a task in it simultaneously.
This can be done easily using the following piece of code.

```
public class Threading {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new Task());
        thread1.start();
        System.out.println("ThreadName  Main Thread: " +
Thread.currentThread().getName());
    }

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

}
```

Create a class that is a Task and implements Runnable, create a new thread and execute the runnable task. This can be visualized as follows.



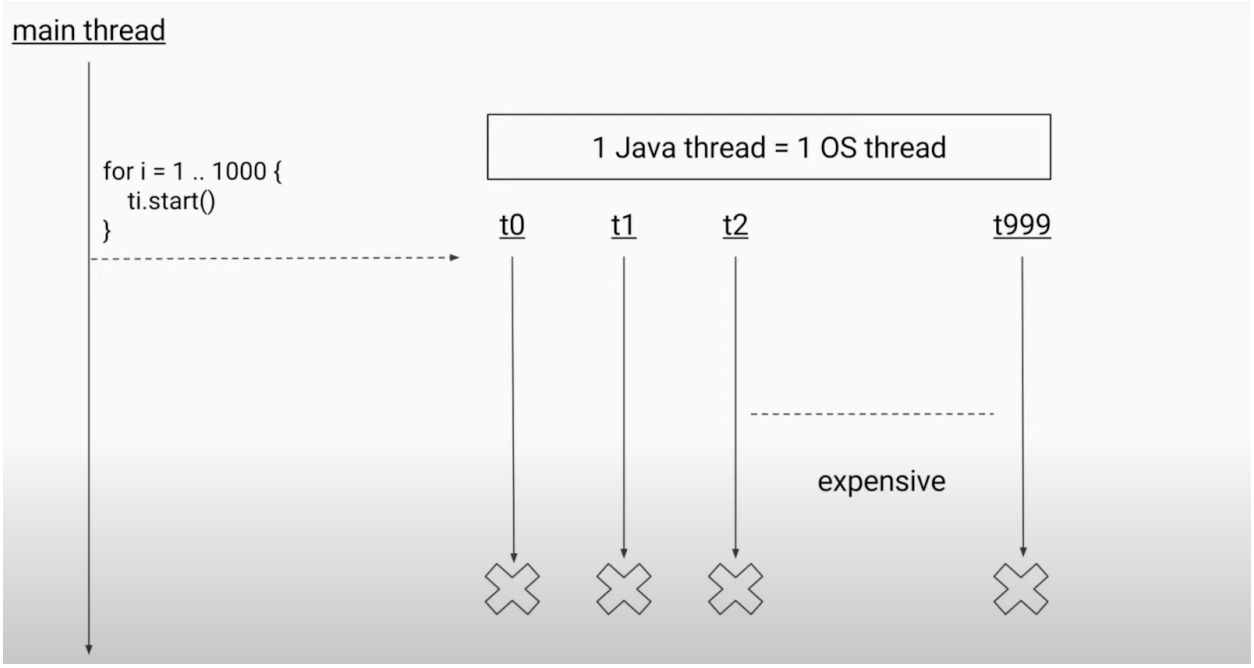
The task will be executed in a separate thread apart from the main thread. Once the execution is done, Java will kill that thread and the main thread will keep going.

To extend this if you want to run 10 tasks, That can be done as follows.

```
public class Threading1 {
    public static void main(String[] args) {

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

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

}
```

Here there will be 10 tasks executed in 10 separate threads. All these 10 created threads will be killed by Java when each of the tasks running in those threads are completed.

Similarly when done with 1000 threads and 1000 tasks, the visualization will be as follows

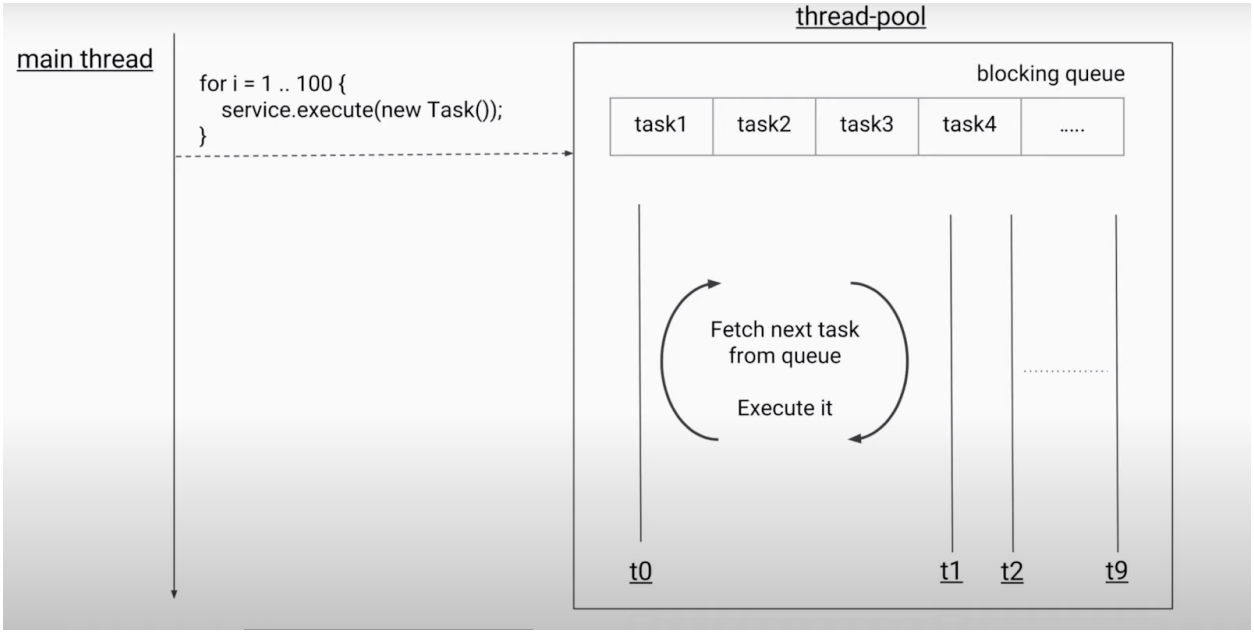


But the problem here is that in Java , one Java thread is equal to one OS thread. That is if you run 1000 tasks as above, it will create 1000 threads and kill 1000 threads post their execution and creating and killing of thread itself is an expensive task.

What could be better is that if we could create say 10 threads (let's call it a pool) and submit 1000 tasks to them and then these threads pick up tasks one after the other.

```
public class ThreadPooling {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(10);
        for (int i = 0; i < 100; i++) {
            service.execute(new Task());
        }
        System.out.println("ThreadName Main Thread : " + Thread.currentThread().getName());
            }

    static class Task implements Runnable{
            public void run() {
```

```
            System.out.println( "ThreadName Child Thread: " +  Thread.currentThread().getName());
        }
    }
}
```



Instead of creating new threads, here we are submitting the new tasks to the executor service and it internally will execute the tasks in these threads.

It internally uses a blocking queue, which stores the tasks that you have submitted and all the threads will perform the same 2 steps:

1. Fetch the new task from the queue
2. Execute it.

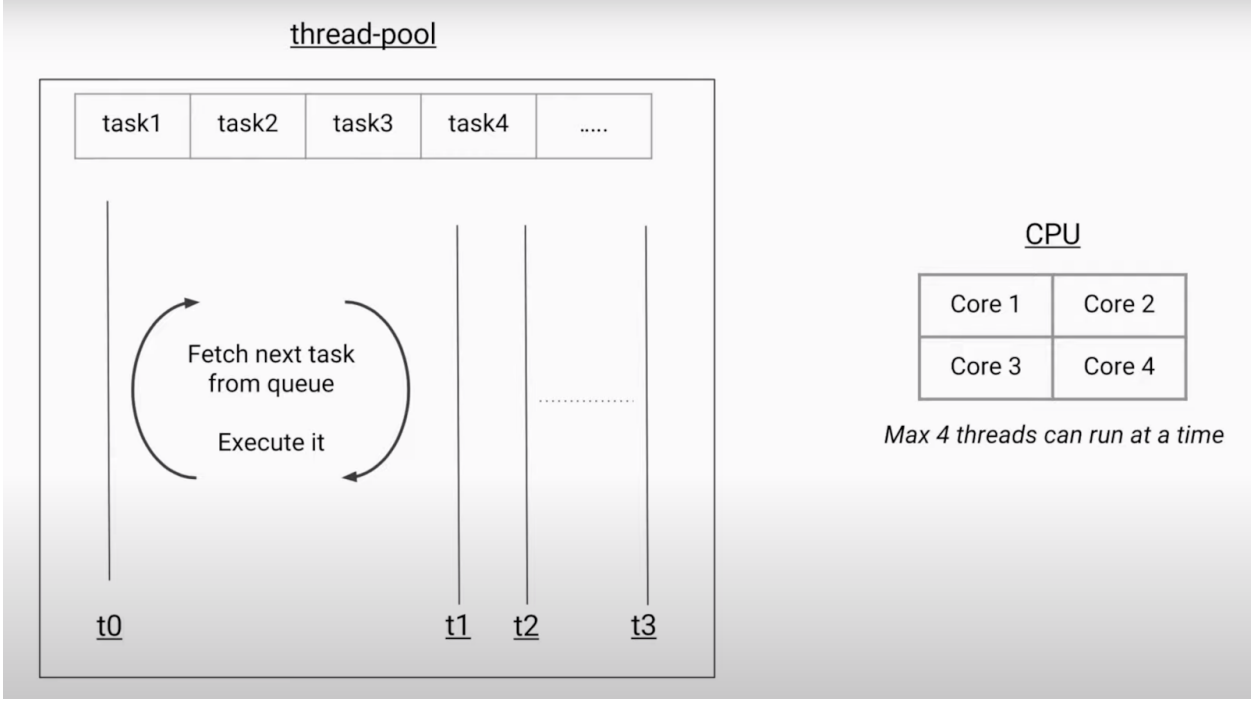
And since all these threads will try to acquire a task from the queue at the same time, there we need a queue that will handle this concurrency and hence we use the blocking queue over here.

IDEAL SIZE OF THE POOL ?

Task of CPU Intensive Nature :

Say the task that we are performing is of heavy computing nature (creating a hash , or a cryptographic function), in this case the task will take a lot of CPU , And as in Java one java thread is equal to one OS thread, so if you have a CPU with 4 cores then at a time you can have only 4 cores running at a time. So that means no matter how many threads you create, the CPU being capable of 4 cores will do split scheduling (like it will give some time to one thread then bump it off and give some time to another thread and so).

So having too many threads is not particularly a good idea. In such cases the ideal number of threads to have is the same as the number of cores in the CPU.



This can be implemented in Java as :

```
public class ThreadPooling {
   public static void main(String[] args) {
      ExecutorService service =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
      for (int i = 0; i < 100; i++) {
         service.execute(new CPUIntensiveTask());
      }
      System.out.println("ThreadName Main Thread : " + Thread.currentThread().getName());
         }
```

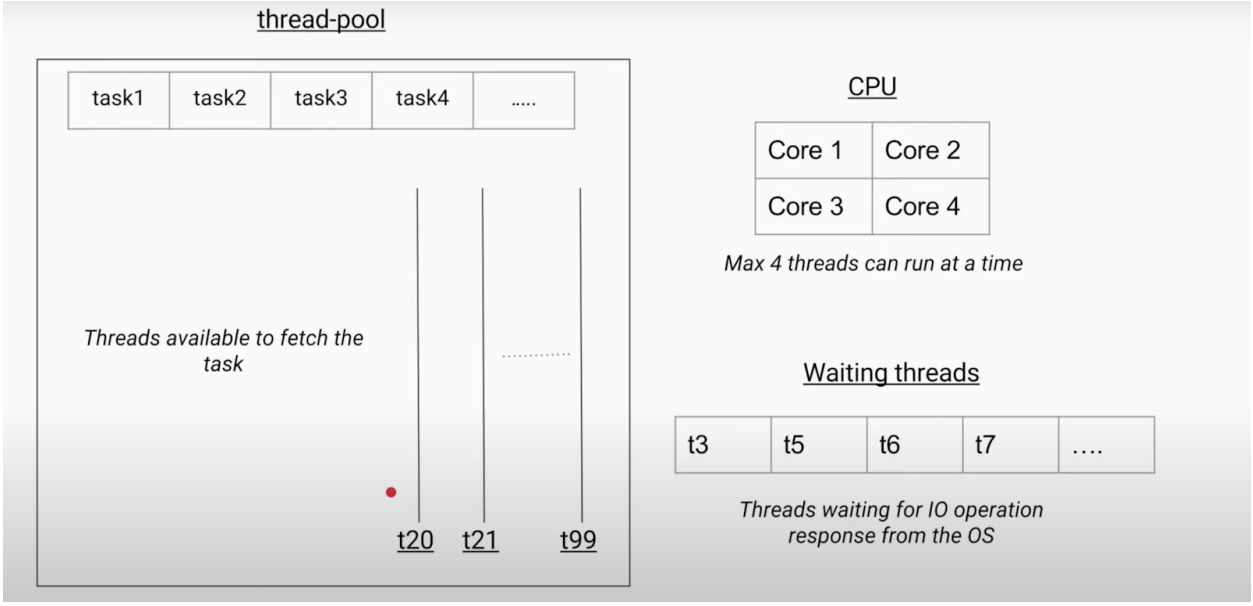
```
   static class CPUIntensiveTask implements Runnable{
           public void run() {
          System.out.println( "ThreadName Child Thread: " +
Thread.currentThread().getName());
       }
   }
}
```

Task of I/O intensive Nature :

Let's assume the task is to get data from DB or the task calls an HTTP client , it will wait for the OS to get a response, and all the threads will be in a waiting state. So even though you have 4 cores and 4 threads are being executed, since all the 4 threads are waiting for some kind of a call back and there are no other threads available to be executed in the CPU.

So in such cases, it is better to have the pool size to be large enough that at most of the times there is some thread running in the CPU core.

Let's assume the number of cores is 4 and the number of threads is 100, then even though there are 50 tasks in the waiting state, there will be 50 threads that are ready to fetch the task and execute.



The code here will be the same just that the size of the thread pool will be set to a higher number.

```
public class ThreadPooling {
   public static void main(String[] args) {
      ExecutorService service = Executors.newFixedThreadPool(100);
      for (int i = 0; i < 100; i++) {
         service.execute(new CPUIntensiveTask());
      }
      System.out.println("ThreadName Main Thread : " +
Thread.currentThread().getName());
         }

   static class CPUIntensiveTask implements Runnable{
          public void run() {
         System.out.println( "ThreadName Child Thread: " +
Thread.currentThread().getName());
      }
   }
}
```

This overall can be summarized as follows:

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.

TYPES OF POOLS

Fixed thread pool.

This is exactly the one we saw earlier. It has a fixed number of threads (say 10), and you keep submitting the tasks to it. All the tasks that are submitted are stored in a queue, since that

queue ahs to be thread safe , therefore blocking queue and all these thread will fetc tasks from the queue and execute it one after the other.

We have already seen how the code for this will be .

Cache thread Pool
Here we do not have a fixed number of threads and we also do not have a queue that will hold the tasks we submit. Instead the queue here is replaced by something called a synchronous queue which has only space for a single item. SO everytime a particular task is submitted, the pool will hold that task in the synchronous queue and search for a thread that is already created and free. If it fails to find a thread that is free, a new thread will be created in the pool that will pick up this new task and execute it.

Cached thread pool also has the ability to kill threads once they have been idle for more than 60s. That is when the size of the thread pool will start shrinking.

Scheduled thread pool.
Here it is specifically for the task that you want to execute after a certain delay (say security checks, logging checks after every 10s etc).

What happens here is that all the tasks here will be stored in a delay queue, where the tasks are not arranged sequentially but are arranged on the basis of when the tasks need to be executed.

In here the tasks can be run in 3 ways :
Scheduled Task : Here we can submit a task and define exactly after how much of time do we want the task to execute.

Scheduled at fixed rate : Here the task that we submit will be executed after every fixed amount of time.

Scheduled at fixed delay : Here the task that we submit will be executed repeatedly at fixed amount of delay after the execution of the previous task.

Single Threaded Executor :
It is exactly the same as a fixed thread pool executor just that the number of threads here is fixed as 1. That is only one task is fetched from the blocking queue and executed.

Now the reason for this special case of fixed thread pool is that suppose there are 3 tasks in the blocking queue and we want them to be executed sequentially one after the other only, it is only possible with only 1 thread.

CONSTRUCTOR AND LIFE CYCLE METHODS :
When taken a look at the source code, internally all the threadPoolExecutors call the same constructor with different params for creation of different type thread pool in the executor service.

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
  }
```

Parameter	Type	Meaning
corePoolSize	int	Minimum/Base size of the pool
maxPoolSize	int	Maximum size of the pool
keepAliveTime + unit	long	Time to keep an idle thread alive (after which it is killed)
workQueue	BlockingQueue	Queue to store the tasks from which threads fetch them
threadFactory	ThreadFactory	The factory to use to create new threads
handler	RejectedExecutionHandler	Callback to use when tasks submitted are rejected

The core pool size is the initial or base size for the pool. Based on the type of the thread pool, the current size of the pool might increase, the max pool size here defines the upper limit of the pool size.

Keep alive time is the time for which if a thread remains idle and there is no other task to be picked up by the thread, the executor service will kill the thread. In this case the pool size will shrink down to the core pool size but not go less than that.

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 kellied unless allowCoreThreadTimeOut(boolean value) is set to true

CorePoolSize is the initial size of thread pool for fixedThreadPool. As no killing or creating of new threads happens the max pool size is also the same. KeepAlive time here as 0 is misleading, but what it means is that keepAliveTime is not applicable here as killing of threads does not happen here.

In the Cached thread pool the minimum size is 0. New threads are created when there is no thread existing for the current task(expanding of threadPool). The upper limit is set as maxPoolSize. The keepAliveTime here is set to 60, i.e after 60s of idleness, the thread is killed (shrinking of threadPool).

In the Scheduled thread pool the minimum pool size is fixed as corePoolSize. MaxPoolSize is set as expansion of thread happens here and the keepAliveTime is set to 60s by default.

In case of single threaded , it being a special case of fixed thread pool, the core and maximum size of the pool here is set to 1. And similarly as no killing of threads is done there is no concept of keepAliveTime in this case.

TYPES OF QUEUES :

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).

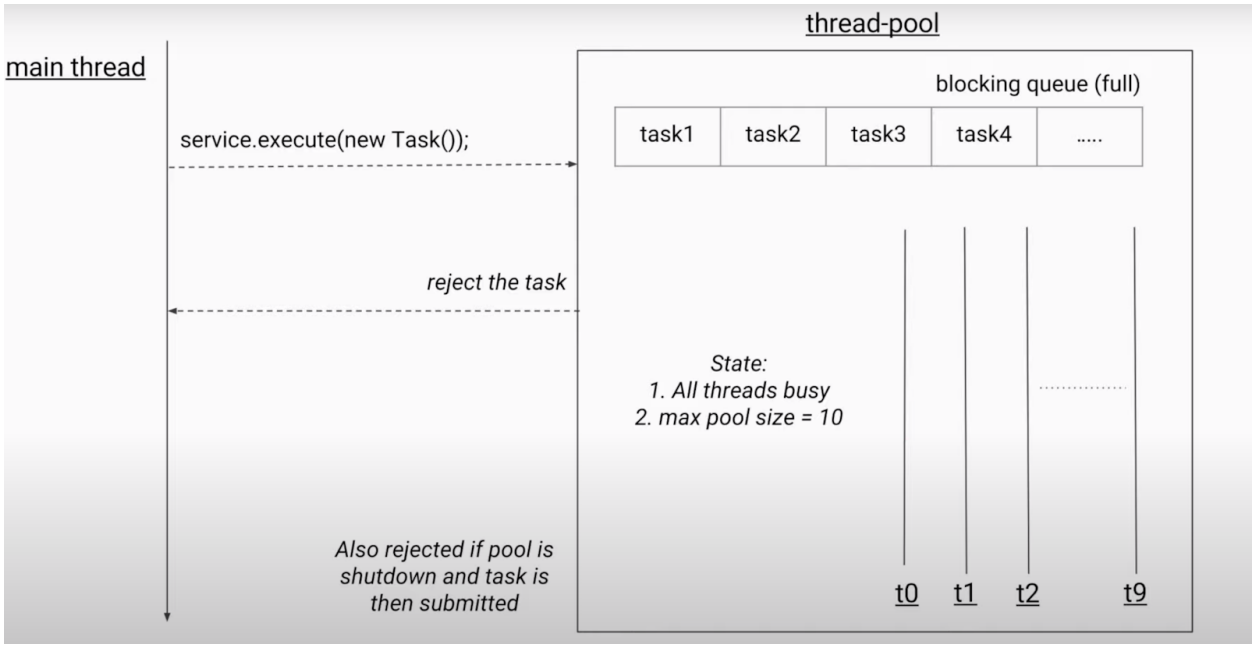
For both FixedThreadPool and SingleThreadedPool a LinkedBlockingQueue is used to store the incoming tasks as the size of thread pool here itself is limited and that means the number of tasks waiting in the queue can keep on increasing (when all the threads are already occupied). Using a limited sized ArrayBlockingQueue would mean that the number of tasks can only increase to a certain limit which is actually unknown in most of the cases, Hence a linkedBlockingQueue is used in this case.

In case of CachedThreadPool the tasks do not keep on increasing in a queue, either a free thread will be used for the new task if available, or else a new thread will be created for the new task. Hence a synchronous queue is used here.

In the case of ScheduleThreadPool deals with schedules or time based execution of tasks. This queue will return the tasks based on whether the schedu;ed time has passed or not.

In case of custom thread pools which one can create by using the ThreadPoolExecutor constructor itself, an ArrayBlockingQueue can be used if required, to put a boundation on the maximum number of tasks that are in waiting state.

REJECTION HANDLER :
Say that we have a thread pool where all the 10 threads are busy and the of queue that we are using for storing of tasks is full (ArrayBoundedQueue), When the next task is submitted to the threadpool, there is no thread that is free to pick up that task, neither does the queue has space enough to occupy another task in it, so in this case the threadpool will reject the task.



We can choose the way we want to handle this rejection of tasks by the thread pool by defining the policy that we want to use.

Policy	What it means?
AbortPolicy	Submitting new tasks throws RejectedExecutionException (Runtime exception)
DiscardPolicy	Submitting new tasks silently discards it.
DiscardOldestPolicy	Submitting new tasks drops existing oldest task, and new task is added to the queue.
CallerRunsPolicy	Submitting new tasks will execute the task on the caller thread itself. This can create feedback loop where caller thread is busy executing the task and cannot submit new tasks at fast pace.

Assume the scenario, all the 10 threads are busy, the blocking queue of size 100 is full.

Abort Policy : Task 111 which will be called for submission will actually be rejected and a runti error of type RejectedExecutionException is raised.

Discard Policy : The Task 111 is simply rejected here and no error is thrown.

DiscardOldestPolicy : In this case the Task 11 i.e the oldest task (first task in the waiting list) is discarded and the new queue is added to the queue.

CallerRunsPolicy : This kind of provides a feedback Mechanism. When the Task 111 arrives, the main thread itself will execute this task and hence Task 112 cannot be submitted.

LIFE CYCLE METHODS :

service.shutdown()

This will just initiate the shutdown, and not immediately shutdown as there can be some tasks in the blocking queue that are pending. But any new tasks submitted after this will not be accepted, but the already submitted ones will be executed.

service.isShutdown()
This call will return true if shutdown has begun.

service.isTerminated()
This call will return true if all the tasks that were being performed in the threads and also in the blocking queue are completed.

service.awaitTermination(long timeout, TimeUnit unit)
blocks until all tasks have completed execution after a shutdown request, or the timeout occurs.

List<Runnable> runnables = service.shitdownNow()
This call will begin the shutdown, it will execute all the tasks that are in the thread already, but will not execute any of the tasks that are in the waiting queue, unlike the isShutdown() Method.

Difference between serve.execute and service.submit???

WHAT IF A RUNNABLE WANTS TO RETURN A VALUE?

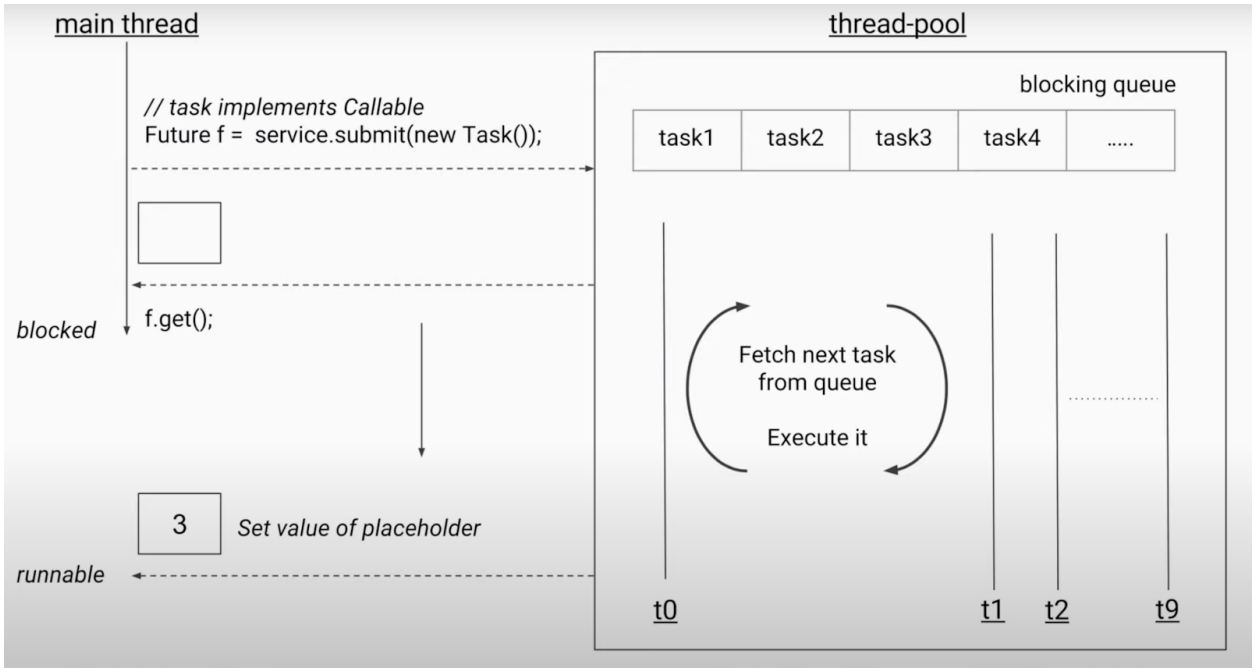
If you see the runnable interface, we can see that it has only one implementable method “run” which returns void. Hence one cannot actually return anything in a task that implements runnable.

But if the use case requires one to return something from the task, we can make a task implement Callable interface.

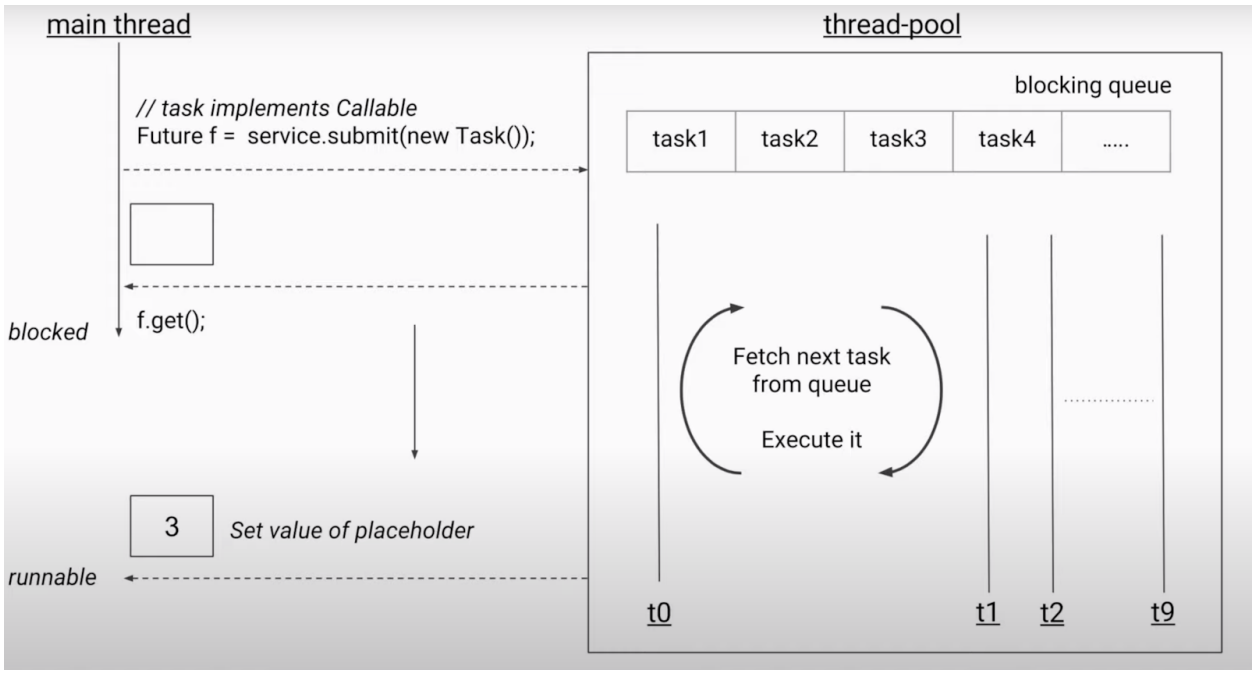
```
public class Threading13 {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        // create the pool
        ExecutorService service = Executors.newFixedThreadPool(10);
        // submit the task for execution
```

```
        Future<Integer> future = service.submit(new Task());
        // Perform some unrelated operations X Y Z

        System.out.println("ThreadName Main Thread : " + Thread.currentThread().getName()
);

        future.get();
        //If the  operation unrelated tasks X Y Z are all executed in say 1 sec,
        //and 4 sec of processing for the thread is still remaining
        //the future.get will  block the main thread for 4 seconds until the task returns the
integer
        }

    static class Task implements Callable<Integer> {

        public Integer call() throws Exception {
            System.out.println("ThreadName Child Thread : " +
Thread.currentThread().getName());
            // Say the operation takes a long time
            Thread.sleep(5000);
            return new Random().nextInt(100);
        }
    }
}
```

We here want to accept the integer value returned by the task in a variable future, now here future should be of type Integer, but it is here of type Future (It is a placeholder for the value that will arrive at sometime in the future, based on how long the call operation takes).



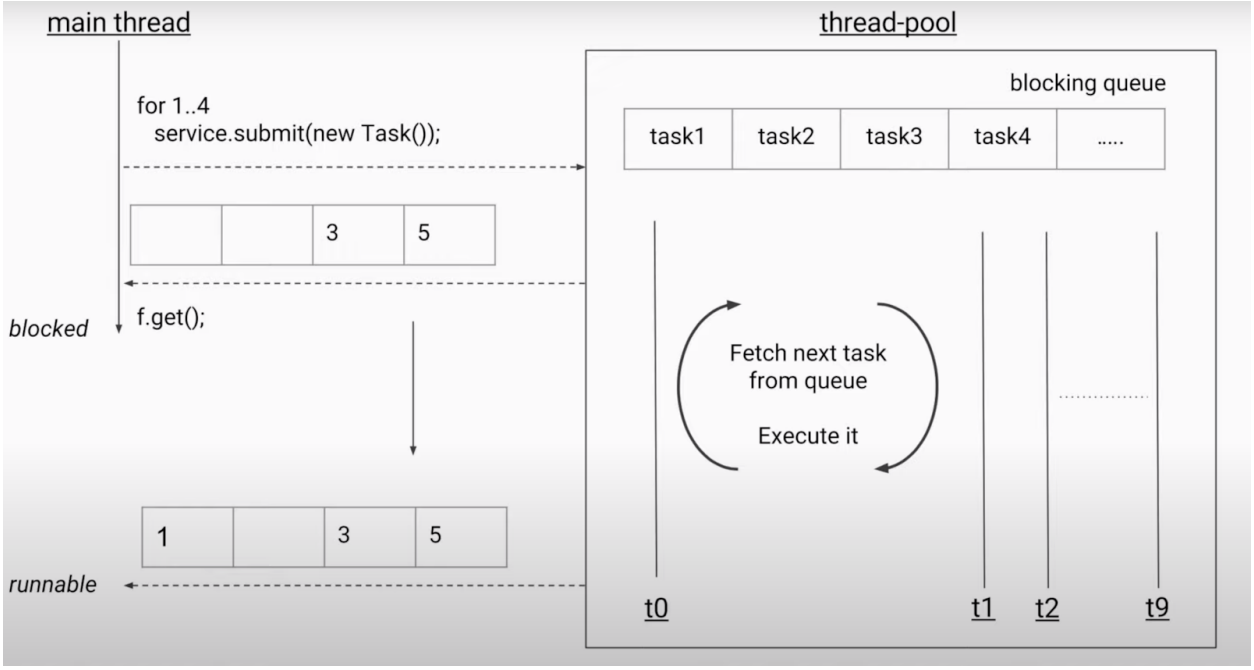
Let us assume that the unrelated tasks X Y Z in the main thread were executed in 1 sec and the we have made the future.get() call already even before the completion of the callable task in other thread (4 more sec to go), not the main thread will wait for the future.get() to return the integer value until the task execution is completed. Thus the main thread will be blocked unnecessarily for 4 seconds here.



Consider we are running 4 such tasks and we will return 4 such future integer values in an array list.

```
public class Threading13 {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      // create the pool
      ExecutorService service = Executors.newFixedThreadPool(10);

      List<Future> allFutures = new ArrayList<Future>();

      // submit the tasks for execution
      for (int i = 0; i < 4; i++)
      {
         Future<Integer> future = service.submit(new Task());
         allFutures.add(future);
      }
      //100 futures with 100 placeholders

      // Perform some unrelated operations P Q R

      for(int j = 0; j < 4; j++)
      {
         Future<Integer> future = allFutures.get(j);
         Integer result = future.get();
         System.out.println("Result of future" +  j + "  is " + result );

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


      //If the  operation unrelated tasks X Y Z are all executed in say 1 sec,
      //and 4 sec of processing for the thread is still remaining
      //the future.get will  block the main thread for 4 seconds until the task returns the integer
      }

   static class Task implements Callable<Integer> {

      public Integer call() throws Exception {
         System.out.println("ThreadName Child Thread : " +
Thread.currentThread().getName());
         // Say the operation takes a long time
         Thread.sleep(1000);
         return new Random().nextInt(100);
}
```

```
    }
}
```



Say that the tasks P Q R were executed in less time and the future.get tasks for all the tasks are being made but not all tasks are completed.

The execution of task 3 and 4 is completed, while the placeholders for Task 1 and 2 are still empty, when future.get() will be made for the first task holder, the main thread will go in a blocked state and wait for the integer response returned at the end of execution of Task 1.

Now once we get the response and the placeholder is set with the integer value, we will move to the next task holder which is the return value of Task 2. As this is also not present, again the main thread will go in a blocked state. For the future.get() calls for Tasks 3 and 4, as the placeholders have already received the return values, the main thread won't be blocked anymore.

To avoid blocking for more than a certain extent, we can also set the timeout parameter.

Integer result = future.get(1, TimeUnit.SECONDS);

This will keep the main thread blocked for not more than 1 second. If the placeholder still does not have a value after the timeout, future.get will throw a TimeoutException.
We can catch this exception and do logging or handle the following methods as well.

1. future.cancel() :
   This method can be called for cancelling the task. If the execution of the task has not begun in the threadpool executor, then it is well and good and the task will be cancelled.

   But say if the execution of the task has already begun in the threadpool then we need to interrupt the thread. In the future.cancel() method we do pass a boolean parameter, which if false means that interruption of thread should not be done and let the task execute if it is picked by the thread already. If true then the task in the thread will be killed.

2. future.isCancelled() :
   Returns true if the task was cancelled.

3. future.isDone() :
   Returns true if the task is completed.Although this is supposed to return true even if the task was executed as a failure.

EXECUTOR COMPLETION SERVICE

```
import java.util.*;
import java.util.concurrent.*;

public class Attempt2 {
  public static void main(String[] args) throws ExecutionException, InterruptedException {
    // create the pool
    ExecutorService service = Executors.newCachedThreadPool();
    CompletionService<Integer> executorCompletionService = new
ExecutorCompletionService(service);
    Map<Future<Integer>, Callable<Integer>> map = new HashMap<Future<Integer>,
Callable<Integer>>();

    Task0 task0 = new Task0();
    Task1 task1 = new Task1();
    Task2 task2 = new Task2();
    Task3 task3 = new Task3();
    Task4 task4 = new Task4();
    Future<Integer> future0 = executorCompletionService.submit(task0);
```

```
        map.put(future0,(task0));
        Future<Integer> future1 = executorCompletionService.submit(task1);
        map.put(future1,(task1));
        Future<Integer> future2 = executorCompletionService.submit(task2);
        map.put(future2,(task2));
        Future<Integer> future3 = executorCompletionService.submit(task3);
        map.put(future3,(task3));
        Future<Integer> future4 = executorCompletionService.submit(task4);
        map.put(future4,(task4));


        for (int i =0 ; i < 5; i++)
        {
            Future<Integer> future = executorCompletionService.take();
            map.get(future);
            System.out.println("Pehle output aya of task " +
map.get(future).getClass().getSimpleName()+ "and output is " + future.get());

        }

        System.out.println("ThreadName Main Thread : " + Thread.currentThread().getName()
);

    }

    static class Task0 implements Callable<Integer> {
        public Integer call() throws Exception {
            System.out.println("ThreadName Child Thread : " +
Thread.currentThread().getName());
            // Say the operation takes a long time
            Thread.sleep(1000);
            return 0;
        }
    }
    static class Task1 implements Callable<Integer> {
        public Integer call() throws Exception {
            System.out.println("ThreadName Child Thread : " +
Thread.currentThread().getName());
            // Say the operation takes a long time
            Thread.sleep(1000);
            return 7;
        }
    }
    static class Task2 implements Callable<Integer> {
```

```
    public Integer call() throws Exception {
        System.out.println("ThreadName Child Thread : " +
Thread.currentThread().getName());
        // Say the operation takes a long time
        Thread.sleep(1000);
        return 14;
    }
  }
  static class Task3 implements Callable<Integer> {
    public Integer call() throws Exception {
        System.out.println("ThreadName Child Thread : " +
Thread.currentThread().getName());
        // Say the operation takes a long time
        Thread.sleep(1000);
        return 21;
    }
  }
  static class Task4 implements Callable<Integer> {
    public Integer call() throws Exception {
        System.out.println("ThreadName Child Thread : " +
Thread.currentThread().getName());
        // Say the operation takes a long time
        Thread.sleep(1000);
        return 28;
    }
  }
}
```

Let's say you have 5 tasks, you submit it to the executors and you want to perform some operation as soon as the task completes. Let's also assume that the first task takes the longest time.

If you use Executors in this case, when you call future.get() on the first task, get operation will be blocked and even though other tasks may have completed, you won't be able to proceed further.

To solve this issue, you can use ExecutorCompletionService. ExecutorCompletionService returns futures objects based on completion order, so whichever task finishes execution first, will

be returned first. You just need to call executorCompletionService.take() to get the completed Future object.