<http://tutorials.jenkov.com/java-concurrency/thread-pools.html>

# Thread Pools

* [Thread Pool Tutorial Video](http://tutorials.jenkov.com/java-concurrency/thread-pools.html#thread-pool-tutorial-video)
* [How a Thread Pool Works](http://tutorials.jenkov.com/java-concurrency/thread-pools.html#how-a-thread-pool-works)
* [Thread Pool Use Cases](http://tutorials.jenkov.com/java-concurrency/thread-pools.html#thread-pool-use-cases)
* [Built-in Java Thread Pool](http://tutorials.jenkov.com/java-concurrency/thread-pools.html#built-in-java-thread-pool)
* [Java Thread Pool Implementation](http://tutorials.jenkov.com/java-concurrency/thread-pools.html#java-thread-pool-implementation)

|  |  |
| --- | --- |
| C:\Users\Quan_Zhang\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\8A70EEDD.tmp | Jakob Jenkov Last update: 2020-11-24  [C:\Users\Quan_Zhang\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\15D681F3.tmp](https://twitter.com/#!/jjenkov) [C:\Users\Quan_Zhang\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\C3908379.tmp](http://www.linkedin.com/pub/jakob-jenkov/0/a8/4a3) [C:\Users\Quan_Zhang\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\AB37ACEF.tmp](https://www.youtube.com/user/jjenkov) [C:\Users\Quan_Zhang\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\574873D5.tmp](http://jenkov.com/rss.xml) [C:\Users\Quan_Zhang\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\222669AB.tmp](https://t.me/jenkov_com) |

A thread pool is a pool threads that can be "reused" to execute tasks, so that each thread may execute more than one task. A thread pool is an alternative to creating a new thread for each task you need to execute.

Creating a new thread comes with a performance overhead compared to reusing a thread that is already created. That is why reusing an existing thread to execute a task can result in a higher total throughput than creating a new thread per task.

Additionally, using a thread pool can make it easier to control how many threads are active at a time. Each thread consumes a certain amount of computer resources, such as memory (RAM), so if you have too many threads active at the same time, the total amount of resources (e.g. RAM) that is consumed may cause the computer to slow down - e.g. if so much RAM is consumed that the operating system (OS) starts swapping RAM out to disk.

In this thread pool tutorial I will explain how thread pools work, how they are used, and how to implement a Java thread pool. Keep in mind, that Java already contains a built-in thread pool - the [**Java ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html) - so you can use a thread pool in Java without having to implement it yourself. However, once in a while you might want to implement your own thread pool - to add functionality that is not supported by the ExecutorService. Or, you may want to implement your own Java thread pool simply as a learning experience.

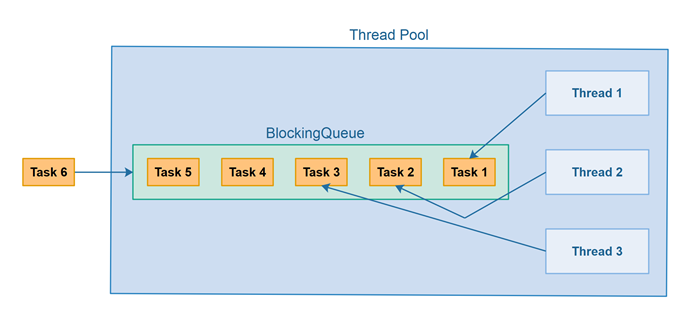
## Thread Pool Tutorial Video

If you prefer video, I have a video version of this thread pool tutorial here: [**Threads Pools in Java**](https://www.youtube.com/watch?v=ZcKt5FYd3bU&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=11)

**[](https://www.youtube.com/watch?v=ZcKt5FYd3bU&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=11)**

## How a Thread Pool Works

Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a [**Blocking Queue**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.



## Thread Pool Use Cases

Thread pools are often used in multi threaded servers. Each connection arriving at the server via the network is wrapped as a task and passed on to a thread pool. The threads in the thread pool will process the requests on the connections concurrently. A later trail will get into detail about implementing multithreaded servers in Java.

## Built-in Java Thread Pool

Java comes with built in thread pools in the java.util.concurrent package, so you don't have to implement your own thread pool. You can read more about it in my text on the [**java.util.concurrent.ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html). Still it can be useful to know a bit about the implementation of a thread pool anyways.

## Java Thread Pool Implementation

Here is a simple thread pool implementation. The implementation uses the standard [**Java BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) that comes with Java from Java 5.

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

public class ThreadPool {

private BlockingQueue taskQueue = null;

private List<PoolThreadRunnable> runnables = new ArrayList<>();

private boolean isStopped = false;

public ThreadPool(int noOfThreads, int maxNoOfTasks){

taskQueue = new ArrayBlockingQueue(maxNoOfTasks);

for(int i=0; i<noOfThreads; i++){

PoolThreadRunnable poolThreadRunnable =

new PoolThreadRunnable(taskQueue);

runnables.add(poolThreadRunnable);

}

for(PoolThreadRunnable runnable : runnables){

new Thread(runnable).start();

}

}

public synchronized void execute(Runnable task) throws Exception{

if(this.isStopped) throw

new IllegalStateException("ThreadPool is stopped");

this.taskQueue.offer(task);

}

public synchronized void stop(){

this.isStopped = true;

for(PoolThreadRunnable runnable : runnables){

runnable.doStop();

}

}

public synchronized void waitUntilAllTasksFinished() {

while(this.taskQueue.size() > 0) {

try {

Thread.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

Below here is the PoolThreadRunnable class which implements the Runnable interface, so it can be executed by a Java thread:

import java.util.concurrent.BlockingQueue;

public class PoolThreadRunnable implements Runnable {

private Thread thread = null;

private BlockingQueue taskQueue = null;

private boolean isStopped = false;

public PoolThreadRunnable(BlockingQueue queue){

taskQueue = queue;

}

public void run(){

this.thread = Thread.currentThread();

while(!isStopped()){

try{

Runnable runnable = (Runnable) taskQueue.take();

runnable.run();

} catch(Exception e){

//log or otherwise report exception,

//but keep pool thread alive.

}

}

}

public synchronized void doStop(){

isStopped = true;

//break pool thread out of dequeue() call.

this.thread.interrupt();

}

public synchronized boolean isStopped(){

return isStopped;

}

}

And here is finally an example of how to use the ThreadPool above:

public class ThreadPoolMain {

public static void main(String[] args) throws Exception {

ThreadPool threadPool = new ThreadPool(3, 10);

for(int i=0; i<10; i++) {

int taskNo = i;

threadPool.execute( () -> {

String message =

Thread.currentThread().getName()

+ ": Task " + taskNo ;

System.out.println(message);

});

}

threadPool.waitUntilAllTasksFinished();

threadPool.stop();

}

}

The thread pool implementation consists of two parts. A ThreadPool class which is the public interface to the thread pool, and a PoolThread class which implements the threads that execute the tasks.

To execute a task the method ThreadPool.execute(Runnable r) is called with a Runnable implementation as parameter. The Runnable is enqueued in the [**blocking queue**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) internally, waiting to be dequeued.

The Runnable will be dequeued by an idle PoolThread and executed. You can see this in the PoolThread.run() method. After execution the PoolThread loops and tries to dequeue a task again, until stopped.

To stop the ThreadPool the method ThreadPool.stop() is called. The stop called is noted internally in the isStopped member. Then each thread in the pool is stopped by calling doStop() on each thread. Notice how the execute() method will throw an IllegalStateException if execute() is called after stop() has been called.

The threads will stop after finishing any task they are currently executing. Notice the this.interrupt() call in PoolThread.doStop(). This makes sure that a thread blocked in a wait() call inside the taskQueue.dequeue() call breaks out of the wait() call, and leaves the dequeue() method call with an InterruptedException thrown. This exception is caught in the PoolThread.run() method, reported, and then the isStopped variable is checked. Since isStopped is now true, the PoolThread.run() will exit and the thread dies.