## 1.What is thread pool and how to create it ?

<http://programmers.stackexchange.com/questions/173575/what-is-a-thread-pool>

A thread pool is a group of pre-instantiated, idle threads which stand ready to be given work. These are preferred over instantiating new threads for each task when there is a large number of short tasks to be done rather than a small number of long ones. This prevents having to incur the overhead of creating a thread a large number of times.

Implementation will vary by environment, but in simplified terms, you need the following(also check the manually implementation as below example):

* A way to create threads and hold them in an idle state. This can be accomplished by having each thread wait at a barrier until the pool hands it work. (This could be done with mutexes as well.)
* A container to store the created threads, such as a queue or any other structure that has a way to add a thread to the pool and pull one out.
* A standard interface or abstract class for the threads to use in doing work. This might be an abstract class called Task with an execute() method that does the work and then returns.

When the thread pool is created, it will either instantiate a certain number of threads to make available or create new ones as needed depending on the needs of the implementation.

When the pool is handed a Task, it takes a thread from the container (or waits for one to become available if the container is empty), hands it a Task, and meets the barrier. This causes the idle thread to resume execution, invoking the execute() method of the Task it was given. Once execution is complete, the thread hands itself back to the pool to be put into the container for re-use and then meets its barrier, putting itself to sleep until the cycle repeats.

Manually create thread pool

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

Thread pool contain 2 major parts: One is Task Queue which implement as Blocking Queue, used for store tasks, another is a container to store a group of Pool Thread which extends from Thread, used for impelementing consume(dequeue) task.

ThreadPool.java

public class ThreadPool {

private BlockingQueue taskQueue = null; 🡪 Part 1

private List<PoolThread> threads = new ArrayList<PoolThread>(); 🡪 Part 2

private boolean isStopped = false;

Note: Both TaskQueue and PoolThread(s) should contained in constructor of ThreadPool which wiring together as an integration of enqueue & dequeue.

public ThreadPool(int noOfThreads, int maxNoOfTasks){

taskQueue = new BlockingQueue(maxNoOfTasks);

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

threads.add(new PoolThread(taskQueue)); 🡪 BlockingQueue will be created as a shared source which defined as member variable

}

for(PoolThread thread : threads){

thread.start();

}

}

public synchronized void execute(Runnable task) throws Exception{

if(this.isStopped) throw

new IllegalStateException("ThreadPool is stopped");

this.taskQueue.enqueue(task);

} 🡪 Note: PoolThread will used for insert tasks onto Task Queue by execute()

public synchronized void stop(){

this.isStopped = true;

for(PoolThread thread : threads){

thread.doStop();

}

}

}

PoolThread.java

public class PoolThread extends Thread {

private BlockingQueue taskQueue = null;

private boolean isStopped = false;

public PoolThread(BlockingQueue queue){

taskQueue = queue;

}

public void run(){

while(!isStopped()){

try{

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

🡪Note:PoolThread will implement dequeue() method in run() method, when any PoolThread running, it will consume a task from Task Queue.

runnable.run();

} catch(Exception e){

//log or otherwise report exception,

//but keep pool thread alive.

}

}

}

public synchronized void doStop(){

isStopped = true;

this.interrupt(); //break pool thread out of dequeue() call.

}

public synchronized boolean isStopped(){

return isStopped;

}

}

Create with ExecutorService

<http://tutorials.jenkov.com/java-util-concurrent/executorservice.html>

An ExecutorService is thus very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

First create an ExecutorService instance,

ExecutorService executorService = Executors.newFixedThreadPool(10);

Then as it contains different methods,

(1)execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously.

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

(2) submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

(3) submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method.

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

(4) invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

Refer to

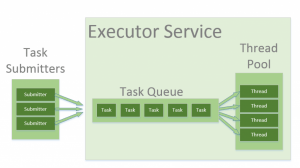
<https://www.baeldung.com/thread-pool-java-and-guava>

## ****2. The Thread Pool****

In Java, threads are mapped to system-level threads which are operating system’s resources. If you create threads uncontrollably, you may run out of these resources quickly.

The context switching between threads is done by the operating system as well – in order to emulate parallelism. A simplistic view is that – the more threads you spawn, the less time each thread spends doing actual work.

The Thread Pool pattern helps to save resources in a multithreaded application, and also to contain the parallelism in certain predefined limits.

When you use a thread pool, you **write your concurrent code in the form of parallel tasks and submit them for execution to an instance of a thread pool**. This instance controls several re-used threads for executing these tasks.  
[](https://www.baeldung.com/wp-content/uploads/2016/08/2016-08-10_10-16-52-1024x572.png)

The pattern allows you to **control the number of threads the application is creating**, their lifecycle, as well as to schedule tasks’ execution and keep incoming tasks in a queue.

## ****3. Thread Pools in Java****

### ****3.1.****Executors****,****Executor****and****ExecutorService

The Executors helper class contains several methods for creation of pre-configured thread pool instances for you. Those classes are a good place to start with – use it if you don’t need to apply any custom fine-tuning.

The Executor and ExecutorService interfaces are used to work with different thread pool implementations in Java. Usually, you should **keep your code decoupled from the actual implementation of the thread pool**and use these interfaces throughout your application.

**The Executor interface has a single execute method to submit Runnable instances for execution.**

**Here’s a quick example** of how you can use the Executors API to acquire an Executor instance backed by a single thread pool and an unbounded queue for executing tasks sequentially. Here, we execute a single task that simply prints “Hello World” on the screen. The task is submitted as a lambda (a Java 8 feature) which is inferred to be Runnable.

|  |  |
| --- | --- |
| 1  2 | Executor executor = Executors.newSingleThreadExecutor();  executor.execute(() -> System.out.println("Hello World")); |

The ExecutorService interface contains a large number of methods for **controlling the progress of the tasks and managing the termination of the service**. Using this interface, you can submit the tasks for execution and also control their execution using the returned Future instance.

**In the following example**, we create an ExecutorService, submit a task and then use the returned Future‘s get method to wait until the submitted task is finished and the value is returned:

|  |  |
| --- | --- |
| 1  2  3  4 | ExecutorService executorService = Executors.newFixedThreadPool(10);  Future<String> future = executorService.submit(() -> "Hello World");  // some operations  String result = future.get(); |

Of course, in a real-life scenario you usually don’t want to call future.get() right away, but defer calling it until you actually need the value of the computation.

The submit method is overloaded to take either Runnable or Callable both of which are functional interfaces and can be passed as lambdas (starting with Java 8).

Runnable‘s single method does not throw an exception and does not return value. Callable interface may be more convenient, as it allows to throw an exception and return a value.

Finally – to let the compiler infer the Callable type, simply return a value from the lambda.

For more examples on using the ExecutorService interface and futures, have a look at “[A Guide to the Java ExecutorService](https://www.baeldung.com/java-executor-service-tutorial)“.

### ****3.2.****ThreadPoolExecutor

The ThreadPoolExecutor is an extensible thread pool implementation with lots of parameters and hooks for fine-tuning.

The main configuration parameters that we’ll discuss here are: **corePoolSize**, **maximumPoolSize**, and **keepAliveTime**.

The pool consists of a fixed number of core threads that are kept inside all the time, and some excessive threads that may be spawned and then terminated when they are not needed anymore. The corePoolSize parameter is the amount of core threads which will be instantiated and kept in the pool. If all core threads are busy and more tasks are submitted, then the pool is allowed to grow up to a maximumPoolSize.

The keepAliveTime parameter is the interval of time for which the excessive threads (i.e. threads that are instantiated in excess of the corePoolSize) are allowed to exist in the idle state.

These parameters cover a wide range of use cases, but **the most typical configurations are predefined in the Executors static methods**.

**For example**, newFixedThreadPool method creates a ThreadPoolExecutor with equal corePoolSizeand maximumPoolSize parameter values and a zero keepAliveTime. This means that the number of threads in this thread pool is always the same:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | ThreadPoolExecutor executor =    (ThreadPoolExecutor) Executors.newFixedThreadPool(2);  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });    assertEquals(2, executor.getPoolSize());  assertEquals(1, executor.getQueue().size()); |

In the example above we instantiate a ThreadPoolExecutor with a fixed thread count of 2. This means that if the amount of simultaneously running tasks is less or equal to two at all times, then they get executed right away. Otherwise **some of these tasks may be put into a queue to wait for their turn**.

We created three Callable tasks that imitate heavy work by sleeping for 1000 milliseconds. The first two tasks will be executed at once, and the third one will have to wait in the queue. We can verify it by calling the getPoolSize() and getQueue().size() methods immediately after submitting the tasks.

Another pre-configured ThreadPoolExecutor can be created with the Executors.newCachedThreadPool() method. This method does not receive a number of threads at all. The corePoolSize is actually set to 0, and the maximumPoolSize is set to Integer.MAX\_VALUE for this instance. The keepAliveTime is 60 seconds for this one.

These parameter values mean that **the cached thread pool may grow without bounds to accommodate any amount of submitted tasks**. But when the threads are not needed anymore, they will be disposed of after 60 seconds of inactivity. A typical use case is when you have a lot of short-living tasks in your application.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | ThreadPoolExecutor executor =    (ThreadPoolExecutor) Executors.newCachedThreadPool();  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });    assertEquals(3, executor.getPoolSize());  assertEquals(0, executor.getQueue().size()); |

The queue size in the example above will always be zero because internally a SynchronousQueueinstance is used. In a SynchronousQueue, pairs of insert and remove operations always occur simultaneously, so the queue never actually contains anything.

The Executors.newSingleThreadExecutor() API creates another typical form of ThreadPoolExecutorcontaining a single thread. **The single thread executor is ideal for creating an event loop.** The corePoolSize and maximumPoolSize parameters are equal to 1, and the keepAliveTime is zero.

Tasks in the above example will be executed sequentially, so the flag value will be 2 after task’s completion:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | AtomicInteger counter = new AtomicInteger();    ExecutorService executor = Executors.newSingleThreadExecutor();  executor.submit(() -> {      counter.set(1);  });  executor.submit(() -> {      counter.compareAndSet(1, 2);  }); |

Additionally, this ThreadPoolExecutor is decorated with an immutable wrapper, so it cannot be reconfigured after creation. Note that also this is the reason we cannot cast it to a ThreadPoolExecutor.

### ****3.3.****ScheduledThreadPoolExecutor

The ScheduledThreadPoolExecutor extends the ThreadPoolExecutor class and also implements the ScheduledExecutorService interface with several additional methods:

* schedule method allows to execute a task once after a specified delay;
* scheduleAtFixedRate method allows to execute a task after a specified initial delay and then execute it repeatedly with a certain period; the period argument is the time **measured between the starting times of the tasks**, so the execution rate is fixed;
* scheduleWithFixedDelay method is similar to scheduleAtFixedRate in that it repeatedly executes the given task, but the specified delay is **measured between the end of the previous task and the start of the next**; the execution rate may vary depending on the time it takes to execute any given task.

The Executors.newScheduledThreadPool() method is typically used to create a ScheduledThreadPoolExecutor with a given corePoolSize, unbounded maximumPoolSize and zero keepAliveTime. Here’s how to schedule a task for execution in 500 milliseconds:

|  |  |
| --- | --- |
| 1  2  3  4 | ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);  executor.schedule(() -> {      System.out.println("Hello World");  }, 500, TimeUnit.MILLISECONDS); |

The following code shows how to execute a task after 500 milliseconds delay and then repeat it every 100 milliseconds. After scheduling the task, we wait until it fires three times using the CountDownLatch lock, then cancel it using the Future.cancel() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CountDownLatch lock = new CountDownLatch(3);    ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);  ScheduledFuture<?> future = executor.scheduleAtFixedRate(() -> {      System.out.println("Hello World");      lock.countDown();  }, 500, 100, TimeUnit.MILLISECONDS);    lock.await(1000, TimeUnit.MILLISECONDS);  future.cancel(true); |

### ****3.4.****ForkJoinPool

ForkJoinPool is the central part of the fork/join framework introduced in Java 7. It solves a common problem of **spawning multiple tasks in recursive algorithms**. Using a simple ThreadPoolExecutor, you will run out of threads quickly, as every task or subtask requires its own thread to run.

In a fork/join framework, any task can spawn (fork) a number of subtasks and wait for their completion using the join method. The benefit of the fork/join framework is that it **does not create a new thread for each task or subtask**, implementing the Work Stealing algorithm instead. This framework is thoroughly described in the article “[Guide to the Fork/Join Framework in Java](https://www.baeldung.com/java-fork-join)“

Let’s look at a simple example of using ForkJoinPool to traverse a tree of nodes and calculate the sum of all leaf values. Here’s a simple implementation of a tree consisting of a node, an int value and a set of child nodes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | static class TreeNode {        int value;        Set<TreeNode> children;        TreeNode(int value, TreeNode... children) {          this.value = value;          this.children = Sets.newHashSet(children);      }  } |

Now if we want to sum all values in a tree in parallel, we need to implement a RecursiveTask<Integer>interface. Each task receives its own node and adds its value to the sum of values of its children. To calculate the sum of children values, the task implementation does the following:

* streams the children set,
* maps over this stream, creating a new CountingTask for each element,
* executes each subtask by forking it,
* collects the results by calling the join method on each forked task,
* sums the results using the Collectors.summingInt collector.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public static class CountingTask extends RecursiveTask<Integer> {        private final TreeNode node;        public CountingTask(TreeNode node) {          this.node = node;      }        @Override      protected Integer compute() {          return node.value + node.children.stream()            .map(childNode -> new CountingTask(childNode).fork())            .collect(Collectors.summingInt(ForkJoinTask::join));      }  } |

The code to run the calculation on an actual tree is very simple:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | TreeNode tree = new TreeNode(5,    new TreeNode(3), new TreeNode(2,      new TreeNode(2), new TreeNode(8)));    ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();  int sum = forkJoinPool.invoke(new CountingTask(tree)); |

## 4. Thread Pool’s Implementation in Guava

[Guava](https://github.com/google/guava) is a popular Google library of utilities. It has many useful concurrency classes, including several handy implementations of ExecutorService. The implementing classes are not accessible for direct instantiation or subclassing, so the only entry point for creating their instances is the MoreExecutorshelper class.

### ****4.1. Adding Guava as a Maven Dependency****

Add the following dependency to your Maven pom file to include the Guava library to your project. You can find the latest version of Guava library in the [Maven Central](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22com.google.guava%22%20AND%20a%3A%22guava%22) repository:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>com.google.guava</groupId>      <artifactId>guava</artifactId>      <version>19.0</version>  </dependency> |

### ****4.2. Direct Executor and Direct Executor Service****

Sometimes you want to execute the task either in the current thread, or in a thread pool, depending on some conditions. You would prefer to use a single Executor interface and just switch the implementation. Although it is not so hard to come up with an implementation of Executor or ExecutorService that executes the tasks in the current thread, it still requires writing some boilerplate code.

Gladly, Guava provides predefined instances for us.

**Here’s an example** that demonstrates execution of a task in the same thread. Although the provided task sleeps for 500 milliseconds, it **blocks the current thread**, and the result is available immediately after the execute call is finished:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Executor executor = MoreExecutors.directExecutor();    AtomicBoolean executed = new AtomicBoolean();    executor.execute(() -> {      try {          Thread.sleep(500);      } catch (InterruptedException e) {          e.printStackTrace();      }      executed.set(true);  });    assertTrue(executed.get()); |

The instance returned by the directExecutor() method is actually a static singleton, so using this method does not provide any overhead on object creation at all.

You should prefer this method to the MoreExecutors.newDirectExecutorService(), because that API creates a full-fledged executor service implementation on every call.

### ****4.3. Exiting Executor Services****

Another common problem is **shutting down the virtual machine** while a thread pool is still running its tasks. Even with a cancellation mechanism in place, there is no guarantee that the tasks will behave nicely and stop their work when the executor service shuts down. This may cause JVM to hang indefinitely while the tasks keep doing their work.

To solve this problem, Guava introduces a family of exiting executor services. They are based on **daemon threads which terminate together with the JVM**.

These services also add a shutdown hook with the Runtime.getRuntime().addShutdownHook()method and prevent the VM from terminating for a configured amount of time before giving up on hung tasks.

In the following example we’re submitting the task that contains an infinite loop, but we use an exiting executor service with a configured time of 100 milliseconds to wait for the tasks upon VM termination. Without the exitingExecutorService in place, this task would cause the VM to hang indefinitely:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | ThreadPoolExecutor executor =    (ThreadPoolExecutor) Executors.newFixedThreadPool(5);  ExecutorService executorService =    MoreExecutors.getExitingExecutorService(executor,      100, TimeUnit.MILLISECONDS);    executorService.submit(() -> {      while (true) {      }  }); |

### ****4.4. Listening Decorators****

Listening decorators allow you to wrap the ExecutorService and receive ListenableFuture instances upon task submission instead of simple Future instances. The ListenableFuture interface extends Future and has a single additional method addListener. This method allows adding a listener that is called upon future completion.

You’ll rarely want to use ListenableFuture.addListener() method directly, but it is **essential to most of the helper methods in the Futures utility class**. For instance, with the Futures.allAsList() method you can combine several ListenableFuture instances in a single ListenableFuture that completes upon the successful completion of all the futures combined:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | ExecutorService executorService = Executors.newCachedThreadPool();  ListeningExecutorService listeningExecutorService =    MoreExecutors.listeningDecorator(executorService);    ListenableFuture<String> future1 =    listeningExecutorService.submit(() -> "Hello");  ListenableFuture<String> future2 =    listeningExecutorService.submit(() -> "World");    String greeting = Futures.allAsList(future1, future2).get()    .stream()    .collect(Collectors.joining(" "));  assertEquals("Hello World", greeting); |