**[Implement Thread pool in java](https://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html)**

*You are here :* [***Home***](http://www.javamadesoeasy.com/) ***/*** [***Core Java Tutorials***](http://www.javamadesoeasy.com/2015/04/core-java-tutorial.html) ***/*** [*Threads/Multi-Threading tutorial in java*](http://www.javamadesoeasy.com/2015/04/thread-tutorial-in-java.html)

*Contents of page :*

* *What is ThreadPool?*
* *Need/Advantage of ThreadPool?*
* [*Life cycle of threads*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *in ThreadPool >*
* *How ThreadPool works?*
  + How threads in ThreadPool can be stopped?
* *Program to implement ThreadPool in java>*
* *Let’s discuss output in detail, to get better understanding of ThreadPool program*
* *How performance of applications is improved by reusing threads?*

*What is ThreadPool?*

ThreadPool is a pool of threads which **reuses a fixed number of threads**  to execute tasks.

At any point, **at most nThreads threads will be active processing tasks**. **If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available**.

ThreadPool implementation internally uses [LinkedBlockingQueue](http://www.javamadesoeasy.com/2015/03/custom-implementation-of.html) for adding and removing tasks.

In this post i will be using LinkedBlockingQueue provided by java Api, you can refer this post for [implementing ThreadPool using custom LinkedBlockingQueue](http://www.javamadesoeasy.com/2015/03/implementing-threadpool-using-custom.html).

We may use [Executor and ExecutorService framework in java](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html) for managing thread life cycle.

*Need/Advantage of ThreadPool?*

**Instead of creating new thread every time for executing tasks**, we can create **ThreadPool** which **reuses a fixed number of threads for executing tasks**.

As threads are reused, performance of our application improves drastically.

[*Life cycle of threads*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *in ThreadPool >*

When threads are **created** in **constructor of ThreadPool** they are in [**New state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

|  |
| --- |
| **new** ThreadPoolsThread(taskQueue,**this**); |

When threads are **started** in **constructor of ThreadPool** they enter **Runnable state**.

|  |
| --- |
| threadPoolsThread.start(); |

When threads enter **run()** method of **ThreadPoolsThread** class they enter **Running state**.

|  |
| --- |
| **class** ThreadPoolsThread **extends** Thread {   . . .  **public** **void** run() {          . . .      }   . . .  } |

Thread can go from **running to waiting state** when taskQueue.take() is called and taskQueue’s size is 0. Thread will wait for tasks to become available.

How can **task become available**/ Threads could go from **waiting to runnable state**?

When execute() method of **ThreadPool** is called, it internally calls put() method on taskQueue to add tasks.

|  |
| --- |
| taskQueue.put(task); |

Once task is available thread can go from **waiting to runnable state**. And later thread scheduler puts thread from **runnable to running state** at discretion of implementation.

Once shutdown of ThreadPool is initiated, **previously submitted tasks are executed** by threads and then threads enter **dead state.**

*How ThreadPool works?*

We will instantiate ThreadPool, in ThreadPool’s **constructor** nThreads number of threads are created and started.

|  |
| --- |
| ThreadPool threadPool=**new** ThreadPool(2); |

Here 2 threads will be created and started in ThreadPool.

Then, threads will enter **run()** method of **ThreadPoolsThread** class and will call take() method on taskQueue.

* If tasks are available thread will execute task by entering run() method of task (As tasks executed always implements Runnable).

|  |
| --- |
| **public** **void** run() {  . . .  **while** (**true**) {          . . .  **Runnable runnable = taskQueue.take();**  **runnable.run();**          . . .      }  . . .  } |

* Else waits for tasks to become available.

**When tasks are added?**

When execute() method of **ThreadPool** is called, it internally calls put() method on taskQueue to add tasks.

|  |
| --- |
| taskQueue.put(task); |

Once tasks are available all waiting threads are notified that task is available.

**How threads in ThreadPool can be stopped?**

**shutDown()** method can be used to stop threads executing in threadPool, once shutdown of ThreadPool is initiated, **previously submitted tasks are executed, but no new tasks could be accepted**.

After thread has executed task

1. Check whether pool shutDown has been initiated or not, if pool shutDown has been initiated and
2. taskQueue does not contain any unExecuted task (i.e. taskQueue's size is 0 )

than interrupt() the thread.

|  |
| --- |
| **public** **void** run() {  . . .  **while** (**true**) {          . . .  **runnable.run();**  **//task EXECUTED**          . . .  **if(this.threadPool.isPoolShutDownInitiated() &&**  **this.taskQueue.size()==0)**  **this.interrupt();**        }  . . .  } |

*Program to implement ThreadPool in java>*

|  |
| --- |
| **package** ThreadPool;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.LinkedBlockingQueue;  /\*\*  \* ThreadPool is a class which creates a thread pool that reuses a fixed  \* number of threads to execute tasks.  \* At any point, at most nThreads threads will be active processing tasks.  \* If additional tasks are submitted when all threads are active,  \* they will wait in the queue until a thread is available.  \*  \* Once shutdown of ThreadPool is initiated, previously submitted tasks are  \* executed, but no new tasks will be accepted.  \*  \* **@author** AnkitMittal  \* Copyright (c), AnkitMittal . [JavaMadeSoEasy.com](http://javamadesoeasy.com/)  \* All Contents are copyrighted and must not be reproduced in any form.  \*/  **class** ThreadPool {  **private** BlockingQueue<Runnable> taskQueue;       /\*      \* Once pool shutDown will be initiated, poolShutDownInitiated will become true.      \*/  **private** **boolean** poolShutDownInitiated = **false**;     /\* Constructor of ThreadPool      \* nThreads= is a number of threads that exist in ThreadPool.      \* nThreads number of threads are created and started. \*      \*/  **public** ThreadPool(**int** nThreads){         taskQueue = **new** LinkedBlockingQueue<Runnable>(nThreads);         //Create and start nThreads number of threads.  **for**(**int** i=1; i<=nThreads; i++){            ThreadPoolsThread threadPoolsThread=**new** ThreadPoolsThread(taskQueue,**this**);          threadPoolsThread.setName("Thread-"+i);          System.*out*.println("Thread-"+i +" created in ThreadPool.");          threadPoolsThread.start();   //start thread         }       }       /\*\*      \* Execute the task, task must be of Runnable type.      \*/  **public** **synchronized** **void**  execute(Runnable task) **throws** Exception{  **if**(**this**.poolShutDownInitiated)  **throw** **new** Exception("ThreadPool has been shutDown, no further tasks can be added");         /\*       \* Add task in sharedQueue,       \* and notify all waiting threads that task is available.            \*/         System.*out*.println("task has been added.");  **this**.taskQueue.put(task);     }  **public** **boolean** isPoolShutDownInitiated() {  **return** poolShutDownInitiated;     }     /\*\*      \* Initiates shutdown of ThreadPool, previously submitted tasks      \* are executed, but no new tasks will be accepted.      \*/  **public** **synchronized** **void** shutdown(){  **this**.poolShutDownInitiated = **true**;         System.*out*.println("ThreadPool SHUTDOWN initiated.");     }  }  /\*\*  \* These threads are created and started from constructor of ThreadPool class.  \*/  **class** ThreadPoolsThread **extends** Thread {  **private** BlockingQueue<Runnable> taskQueue;  **private** ThreadPool threadPool;  **public** ThreadPoolsThread(BlockingQueue<Runnable> queue,                   ThreadPool threadPool){         taskQueue = queue;  **this**.threadPool=threadPool;       }  **public** **void** run() {  **try** {                   /\*                   \* ThreadPool's threads will keep on running                   \* until ThreadPool is not shutDown (shutDown will interrupt thread) and                   \* taskQueue contains some unExecuted tasks.                   \*/  **while** (**true**) {                         System.*out*.println(Thread.*currentThread*().getName()                                       +" is READY to execute task.");                         /\*ThreadPool's thread will take() task from sharedQueue                         \* only if tasks are available else                         \* waits for tasks to become available.                         \*/                         Runnable runnable = taskQueue.take();                         System.*out*.println(Thread.*currentThread*().getName()                                       +" has taken task.");                         //Now, execute task with current thread.                         runnable.run();                           System.*out*.println(Thread.*currentThread*().getName()                                       +" has EXECUTED task.");                           /\*                         \* 1) Check whether pool shutDown has been initiated or not,                         \* if pool shutDown has been initiated and                         \* 2) taskQueue does not contain any                         \*    unExecuted task (i.e. taskQueue's size is 0 )                         \* than  interrupt() the thread.                         \*/  **if**(**this**.threadPool.isPoolShutDownInitiated()                                       &&  **this**.taskQueue.size()==0){  **this**.interrupt();                             /\*                                \*  Interrupting basically sends a message to the thread                                \*  indicating it has been interrupted but it doesn't cause                                \*  a thread to stop immediately,                                \*                                \*  if sleep is called, thread immediately throws InterruptedException                                \*/                                Thread.*sleep*(1);                         }                     }            } **catch** (InterruptedException e) {                   System.*out*.println(Thread.*currentThread*().getName()+" has been STOPPED.");            }     }  }  /\*\*  \* Task class which implements Runnable.  \*/  **class** Task **implements** Runnable{     @Override  **public** **void** run() {  **try** {                   Thread.*sleep*(2000);                   System.*out*.println(Thread.*currentThread*().getName()                                +" is executing task.");            } **catch** (InterruptedException e) {                   e.printStackTrace();            }     }  };  /\*\*  \* Test ThreadPool.  \*/  **public** **class** ThreadPoolTest{  **public** **static** **void** main(String[] args) **throws** Exception {            ThreadPool threadPool=**new** ThreadPool(2); //create 2 threads in ThreadPool            Runnable task=**new** Task();            threadPool.execute(task);            threadPool.execute(task);              threadPool.shutdown();     }    }  /\*OUTPUT  Thread-1 created in ThreadPool.  Thread-2 created in ThreadPool.  Thread-1 is READY to execute task.  Thread-2 is READY to execute task.  task has been added.  task has been added.  Thread-1 has taken task.  Thread-2 has taken task.  ThreadPool SHUTDOWN initiated.  Thread-1 is executing task.  Thread-1 has EXECUTED task.  Thread-1 has been STOPPED.  Thread-2 is executing task.  Thread-2 has EXECUTED task.  Thread-2 has been STOPPED.  \*/ |

*Let’s discuss output in detail, to get better understanding of ThreadPool program >*

**Note** : I have mentioned output in **green** text.

Total number of thread created in ThreadPool was 2.

Thread-1 **created** in ThreadPool.

Till now Thread-1 have been created.

Thread-2 **created** in ThreadPool.

Till now Thread-2 have been created.

Thread-1 is **READY** to execute task.

Thread-1 have entered **run()** method and taskQueue’s size is 0. So its waiting for task to become available.

Thread-2 is **READY** to execute task.

Thread-2 have entered **run()** method and taskQueue’s size is 0. So its waiting for task to become available.

task has been **added**.

execute() method of **ThreadPool** is called by main thread, it internally calls **put()** method on taskQueue to add tasks. Once tasks is available all waiting threads are notified that task is available.

task has been **added**.

execute() method of **ThreadPool** is called by main thread, it internally calls **put()** method on taskQueue to add tasks. Once tasks is available all waiting threads are notified that task is available.

Thread-1 has taken task.

As waiting Thread-1 has been notified it takes task.

Thread-2 has taken task.

As waiting Thread-2 has been notified it takes task.

ThreadPool SHUTDOWN initiated.

threadPool.**shutdown()** is called by main thread, previously submitted tasks are executed, but no new tasks will be accepted.

Thread-1 is executing task.

Thread-1 is executing task, it’s in run() method of Task class (shutdown was initiated, but  previously submitted tasks are executed ).

Thread-1 has EXECUTED task.

Thread-1 has executed task.

Thread-1 has been STOPPED.

Thread-1 has been stopped.

Thread-2 is executing task.

Thread-2 is executing task, it’s in run() method of Task class.

Thread-2 has EXECUTED task.

Thread-2 has executed task.

Thread-2 has been STOPPED.

Thread-2 has been stopped.

*How performance of applications is improved by reusing threads?*

So, after constructor and before shutdown is called on ThreadPool, threads will remain either in Running, Runnable or Waiting state. **Therefore excluding overhead of being in New and Dead state.**

Therefore, for every task executed by thread it would never go in new and dead state hence saving the time and will improve applications performance.