| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/ThreadPoolExecutor.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/java/util/concurrent/ThreadFactory.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.AbortPolicy.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/ThreadPoolExecutor.html)    [**NO FRAMES**](http://docs.google.com/ThreadPoolExecutor.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: [NESTED](#3znysh7) | FIELD | [CONSTR](#2et92p0) | [METHOD](#tyjcwt) | DETAIL: FIELD | [CONSTR](#4d34og8) | [METHOD](#lnxbz9) |

## **java.util.concurrent**

Class ThreadPoolExecutor

[java.lang.Object](http://docs.google.com/java/lang/Object.html)  
 [java.util.concurrent.AbstractExecutorService](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html)  
 **java.util.concurrent.ThreadPoolExecutor**

**All Implemented Interfaces:** [Executor](http://docs.google.com/java/util/concurrent/Executor.html), [ExecutorService](http://docs.google.com/java/util/concurrent/ExecutorService.html) **Direct Known Subclasses:** [ScheduledThreadPoolExecutor](http://docs.google.com/java/util/concurrent/ScheduledThreadPoolExecutor.html)

public class **ThreadPoolExecutor**extends [AbstractExecutorService](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html)

An [ExecutorService](http://docs.google.com/java/util/concurrent/ExecutorService.html) that executes each submitted task using one of possibly several pooled threads, normally configured using [Executors](http://docs.google.com/java/util/concurrent/Executors.html) factory methods.

Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.

To be useful across a wide range of contexts, this class provides many adjustable parameters and extensibility hooks. However, programmers are urged to use the more convenient [Executors](http://docs.google.com/java/util/concurrent/Executors.html) factory methods [Executors.newCachedThreadPool()](http://docs.google.com/java/util/concurrent/Executors.html#newCachedThreadPool()) (unbounded thread pool, with automatic thread reclamation), [Executors.newFixedThreadPool(int)](http://docs.google.com/java/util/concurrent/Executors.html#newFixedThreadPool(int)) (fixed size thread pool) and [Executors.newSingleThreadExecutor()](http://docs.google.com/java/util/concurrent/Executors.html#newSingleThreadExecutor()) (single background thread), that preconfigure settings for the most common usage scenarios. Otherwise, use the following guide when manually configuring and tuning this class:

Core and maximum pool sizes A ThreadPoolExecutor will automatically adjust the pool size (see [getPoolSize()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getPoolSize())) according to the bounds set by corePoolSize (see [getCorePoolSize()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getCorePoolSize())) and maximumPoolSize (see [getMaximumPoolSize()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getMaximumPoolSize())). When a new task is submitted in method [execute(java.lang.Runnable)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#execute(java.lang.Runnable)), and fewer than corePoolSize threads are running, a new thread is created to handle the request, even if other worker threads are idle. If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created only if the queue is full. By setting corePoolSize and maximumPoolSize the same, you create a fixed-size thread pool. By setting maximumPoolSize to an essentially unbounded value such as Integer.MAX\_VALUE, you allow the pool to accommodate an arbitrary number of concurrent tasks. Most typically, core and maximum pool sizes are set only upon construction, but they may also be changed dynamically using [setCorePoolSize(int)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setCorePoolSize(int)) and [setMaximumPoolSize(int)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setMaximumPoolSize(int)). On-demand construction By default, even core threads are initially created and started only when new tasks arrive, but this can be overridden dynamically using method [prestartCoreThread()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#prestartCoreThread()) or [prestartAllCoreThreads()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#prestartAllCoreThreads()). You probably want to prestart threads if you construct the pool with a non-empty queue. Creating new threads New threads are created using a [ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html). If not otherwise specified, a [Executors.defaultThreadFactory()](http://docs.google.com/java/util/concurrent/Executors.html#defaultThreadFactory()) is used, that creates threads to all be in the same [ThreadGroup](http://docs.google.com/java/lang/ThreadGroup.html) and with the same NORM\_PRIORITY priority and non-daemon status. By supplying a different ThreadFactory, you can alter the thread's name, thread group, priority, daemon status, etc. If a ThreadFactory fails to create a thread when asked by returning null from newThread, the executor will continue, but might not be able to execute any tasks. Keep-alive times If the pool currently has more than corePoolSize threads, excess threads will be terminated if they have been idle for more than the keepAliveTime (see [getKeepAliveTime(java.util.concurrent.TimeUnit)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getKeepAliveTime(java.util.concurrent.TimeUnit))). This provides a means of reducing resource consumption when the pool is not being actively used. If the pool becomes more active later, new threads will be constructed. This parameter can also be changed dynamically using method [setKeepAliveTime(long, java.util.concurrent.TimeUnit)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setKeepAliveTime(long,%20java.util.concurrent.TimeUnit)). Using a value of Long.MAX\_VALUE [TimeUnit.NANOSECONDS](http://docs.google.com/java/util/concurrent/TimeUnit.html#NANOSECONDS) effectively disables idle threads from ever terminating prior to shut down. By default, the keep-alive policy applies only when there are more than corePoolSizeThreads. But method [allowCoreThreadTimeOut(boolean)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#allowCoreThreadTimeOut(boolean)) can be used to apply this time-out policy to core threads as well, so long as the keepAliveTime value is non-zero. Queuing Any [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html) may be used to transfer and hold submitted tasks. The use of this queue interacts with pool sizing:

* If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.
* If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.
* If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.

There are three general strategies for queuing:

1. *Direct handoffs.* A good default choice for a work queue is a [SynchronousQueue](http://docs.google.com/java/util/concurrent/SynchronousQueue.html) that hands off tasks to threads without otherwise holding them. Here, an attempt to queue a task will fail if no threads are immediately available to run it, so a new thread will be constructed. This policy avoids lockups when handling sets of requests that might have internal dependencies. Direct handoffs generally require unbounded maximumPoolSizes to avoid rejection of new submitted tasks. This in turn admits the possibility of unbounded thread growth when commands continue to arrive on average faster than they can be processed.
2. *Unbounded queues.* Using an unbounded queue (for example a [LinkedBlockingQueue](http://docs.google.com/java/util/concurrent/LinkedBlockingQueue.html) without a predefined capacity) will cause new tasks to wait in the queue when all corePoolSize threads are busy. Thus, no more than corePoolSize threads will ever be created. (And the value of the maximumPoolSize therefore doesn't have any effect.) This may be appropriate when each task is completely independent of others, so tasks cannot affect each others execution; for example, in a web page server. While this style of queuing can be useful in smoothing out transient bursts of requests, it admits the possibility of unbounded work queue growth when commands continue to arrive on average faster than they can be processed.
3. *Bounded queues.* A bounded queue (for example, an [ArrayBlockingQueue](http://docs.google.com/java/util/concurrent/ArrayBlockingQueue.html)) helps prevent resource exhaustion when used with finite maximumPoolSizes, but can be more difficult to tune and control. Queue sizes and maximum pool sizes may be traded off for each other: Using large queues and small pools minimizes CPU usage, OS resources, and context-switching overhead, but can lead to artificially low throughput. If tasks frequently block (for example if they are I/O bound), a system may be able to schedule time for more threads than you otherwise allow. Use of small queues generally requires larger pool sizes, which keeps CPUs busier but may encounter unacceptable scheduling overhead, which also decreases throughput.

Rejected tasks New tasks submitted in method [execute(java.lang.Runnable)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#execute(java.lang.Runnable)) will be *rejected* when the Executor has been shut down, and also when the Executor uses finite bounds for both maximum threads and work queue capacity, and is saturated. In either case, the execute method invokes the [RejectedExecutionHandler.rejectedExecution(java.lang.Runnable, java.util.concurrent.ThreadPoolExecutor)](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html#rejectedExecution(java.lang.Runnable,%20java.util.concurrent.ThreadPoolExecutor)) method of its [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html). Four predefined handler policies are provided:

1. In the default [ThreadPoolExecutor.AbortPolicy](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.AbortPolicy.html), the handler throws a runtime [RejectedExecutionException](http://docs.google.com/java/util/concurrent/RejectedExecutionException.html) upon rejection.
2. In [ThreadPoolExecutor.CallerRunsPolicy](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.CallerRunsPolicy.html), the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.
3. In [ThreadPoolExecutor.DiscardPolicy](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.DiscardPolicy.html), a task that cannot be executed is simply dropped.
4. In [ThreadPoolExecutor.DiscardOldestPolicy](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.DiscardOldestPolicy.html), if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)

It is possible to define and use other kinds of [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) classes. Doing so requires some care especially when policies are designed to work only under particular capacity or queuing policies. Hook methods This class provides protected overridable [beforeExecute(java.lang.Thread, java.lang.Runnable)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#beforeExecute(java.lang.Thread,%20java.lang.Runnable)) and [afterExecute(java.lang.Runnable, java.lang.Throwable)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#afterExecute(java.lang.Runnable,%20java.lang.Throwable)) methods that are called before and after execution of each task. These can be used to manipulate the execution environment; for example, reinitializing ThreadLocals, gathering statistics, or adding log entries. Additionally, method [terminated()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#terminated()) can be overridden to perform any special processing that needs to be done once the Executor has fully terminated.

If hook or callback methods throw exceptions, internal worker threads may in turn fail and abruptly terminate.

Queue maintenance Method [getQueue()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getQueue()) allows access to the work queue for purposes of monitoring and debugging. Use of this method for any other purpose is strongly discouraged. Two supplied methods, [remove(java.lang.Runnable)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#remove(java.lang.Runnable)) and [purge()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#purge()) are available to assist in storage reclamation when large numbers of queued tasks become cancelled. Finalization A pool that is no longer referenced in a program *AND* has no remaining threads will be shutdown automatically. If you would like to ensure that unreferenced pools are reclaimed even if users forget to call [shutdown()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#shutdown()), then you must arrange that unused threads eventually die, by setting appropriate keep-alive times, using a lower bound of zero core threads and/or setting [allowCoreThreadTimeOut(boolean)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#allowCoreThreadTimeOut(boolean)).

**Extension example**. Most extensions of this class override one or more of the protected hook methods. For example, here is a subclass that adds a simple pause/resume feature:

class PausableThreadPoolExecutor extends ThreadPoolExecutor {  
 private boolean isPaused;  
 private ReentrantLock pauseLock = new ReentrantLock();  
 private Condition unpaused = pauseLock.newCondition();  
  
 public PausableThreadPoolExecutor(...) { super(...); }  
  
 protected void beforeExecute(Thread t, Runnable r) {  
 super.beforeExecute(t, r);  
 pauseLock.lock();  
 try {  
 while (isPaused) unpaused.await();  
 } catch (InterruptedException ie) {  
 t.interrupt();  
 } finally {  
 pauseLock.unlock();  
 }  
 }  
  
 public void pause() {  
 pauseLock.lock();  
 try {  
 isPaused = true;  
 } finally {  
 pauseLock.unlock();  
 }  
 }  
  
 public void resume() {  
 pauseLock.lock();  
 try {  
 isPaused = false;  
 unpaused.signalAll();  
 } finally {  
 pauseLock.unlock();  
 }  
 }  
 }

**Since:** 1.5

| **Nested Class Summary** | |
| --- | --- |
| static class | [**ThreadPoolExecutor.AbortPolicy**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.AbortPolicy.html)            A handler for rejected tasks that throws a RejectedExecutionException. |
| static class | [**ThreadPoolExecutor.CallerRunsPolicy**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.CallerRunsPolicy.html)            A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute method, unless the executor has been shut down, in which case the task is discarded. |
| static class | [**ThreadPoolExecutor.DiscardOldestPolicy**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.DiscardOldestPolicy.html)            A handler for rejected tasks that discards the oldest unhandled request and then retries execute, unless the executor is shut down, in which case the task is discarded. |
| static class | [**ThreadPoolExecutor.DiscardPolicy**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.DiscardPolicy.html)            A handler for rejected tasks that silently discards the rejected task. |

| **Constructor Summary** | |
| --- | --- |
| [**ThreadPoolExecutor**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#ThreadPoolExecutor(int,%20int,%20long,%20java.util.concurrent.TimeUnit,%20java.util.concurrent.BlockingQueue))(int corePoolSize, int maximumPoolSize, long keepAliveTime, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit, [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue)            Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory and rejected execution handler. |
| [**ThreadPoolExecutor**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#ThreadPoolExecutor(int,%20int,%20long,%20java.util.concurrent.TimeUnit,%20java.util.concurrent.BlockingQueue,%20java.util.concurrent.RejectedExecutionHandler))(int corePoolSize, int maximumPoolSize, long keepAliveTime, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit, [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue, [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) handler)            Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory. |
| [**ThreadPoolExecutor**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#ThreadPoolExecutor(int,%20int,%20long,%20java.util.concurrent.TimeUnit,%20java.util.concurrent.BlockingQueue,%20java.util.concurrent.ThreadFactory))(int corePoolSize, int maximumPoolSize, long keepAliveTime, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit, [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue, [ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) threadFactory)            Creates a new ThreadPoolExecutor with the given initial parameters and default rejected execution handler. |
| [**ThreadPoolExecutor**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#ThreadPoolExecutor(int,%20int,%20long,%20java.util.concurrent.TimeUnit,%20java.util.concurrent.BlockingQueue,%20java.util.concurrent.ThreadFactory,%20java.util.concurrent.RejectedExecutionHandler))(int corePoolSize, int maximumPoolSize, long keepAliveTime, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit, [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue, [ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) threadFactory, [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) handler)            Creates a new ThreadPoolExecutor with the given initial parameters. |

| **Method Summary** | |
| --- | --- |
| protected  void | [**afterExecute**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#afterExecute(java.lang.Runnable,%20java.lang.Throwable))([Runnable](http://docs.google.com/java/lang/Runnable.html) r, [Throwable](http://docs.google.com/java/lang/Throwable.html) t)            Method invoked upon completion of execution of the given Runnable. |
| void | [**allowCoreThreadTimeOut**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#allowCoreThreadTimeOut(boolean))(boolean value)            Sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive. |
| boolean | [**allowsCoreThreadTimeOut**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#allowsCoreThreadTimeOut())()            Returns true if this pool allows core threads to time out and terminate if no tasks arrive within the keepAlive time, being replaced if needed when new tasks arrive. |
| boolean | [**awaitTermination**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#awaitTermination(long,%20java.util.concurrent.TimeUnit))(long timeout, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)            Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first. |
| protected  void | [**beforeExecute**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#beforeExecute(java.lang.Thread,%20java.lang.Runnable))([Thread](http://docs.google.com/java/lang/Thread.html) t, [Runnable](http://docs.google.com/java/lang/Runnable.html) r)            Method invoked prior to executing the given Runnable in the given thread. |
| void | [**execute**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#execute(java.lang.Runnable))([Runnable](http://docs.google.com/java/lang/Runnable.html) command)            Executes the given task sometime in the future. |
| protected  void | [**finalize**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#finalize())()            Invokes shutdown when this executor is no longer referenced. |
| int | [**getActiveCount**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getActiveCount())()            Returns the approximate number of threads that are actively executing tasks. |
| long | [**getCompletedTaskCount**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getCompletedTaskCount())()            Returns the approximate total number of tasks that have completed execution. |
| int | [**getCorePoolSize**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getCorePoolSize())()            Returns the core number of threads. |
| long | [**getKeepAliveTime**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getKeepAliveTime(java.util.concurrent.TimeUnit))([TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)            Returns the thread keep-alive time, which is the amount of time that threads in excess of the core pool size may remain idle before being terminated. |
| int | [**getLargestPoolSize**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getLargestPoolSize())()            Returns the largest number of threads that have ever simultaneously been in the pool. |
| int | [**getMaximumPoolSize**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getMaximumPoolSize())()            Returns the maximum allowed number of threads. |
| int | [**getPoolSize**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getPoolSize())()            Returns the current number of threads in the pool. |
| [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> | [**getQueue**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getQueue())()            Returns the task queue used by this executor. |
| [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) | [**getRejectedExecutionHandler**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getRejectedExecutionHandler())()            Returns the current handler for unexecutable tasks. |
| long | [**getTaskCount**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getTaskCount())()            Returns the approximate total number of tasks that have ever been scheduled for execution. |
| [ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) | [**getThreadFactory**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getThreadFactory())()            Returns the thread factory used to create new threads. |
| boolean | [**isShutdown**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#isShutdown())()            Returns true if this executor has been shut down. |
| boolean | [**isTerminated**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#isTerminated())()            Returns true if all tasks have completed following shut down. |
| boolean | [**isTerminating**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#isTerminating())()            Returns true if this executor is in the process of terminating after shutdown or shutdownNow but has not completely terminated. |
| int | [**prestartAllCoreThreads**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#prestartAllCoreThreads())()            Starts all core threads, causing them to idly wait for work. |
| boolean | [**prestartCoreThread**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#prestartCoreThread())()            Starts a core thread, causing it to idly wait for work. |
| void | [**purge**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#purge())()            Tries to remove from the work queue all [Future](http://docs.google.com/java/util/concurrent/Future.html) tasks that have been cancelled. |
| boolean | [**remove**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#remove(java.lang.Runnable))([Runnable](http://docs.google.com/java/lang/Runnable.html) task)            Removes this task from the executor's internal queue if it is present, thus causing it not to be run if it has not already started. |
| void | [**setCorePoolSize**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setCorePoolSize(int))(int corePoolSize)            Sets the core number of threads. |
| void | [**setKeepAliveTime**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setKeepAliveTime(long,%20java.util.concurrent.TimeUnit))(long time, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)            Sets the time limit for which threads may remain idle before being terminated. |
| void | [**setMaximumPoolSize**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setMaximumPoolSize(int))(int maximumPoolSize)            Sets the maximum allowed number of threads. |
| void | [**setRejectedExecutionHandler**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setRejectedExecutionHandler(java.util.concurrent.RejectedExecutionHandler))([RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) handler)            Sets a new handler for unexecutable tasks. |
| void | [**setThreadFactory**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setThreadFactory(java.util.concurrent.ThreadFactory))([ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) threadFactory)            Sets the thread factory used to create new threads. |
| void | [**shutdown**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#shutdown())()            Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. |
| [List](http://docs.google.com/java/util/List.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> | [**shutdownNow**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#shutdownNow())()            Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. |
| protected  void | [**terminated**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#terminated())()            Method invoked when the Executor has terminated. |

| **Methods inherited from class java.util.concurrent.**[**AbstractExecutorService**](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html) |
| --- |
| [invokeAll](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#invokeAll(java.util.Collection)), [invokeAll](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#invokeAll(java.util.Collection,%20long,%20java.util.concurrent.TimeUnit)), [invokeAny](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#invokeAny(java.util.Collection)), [invokeAny](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#invokeAny(java.util.Collection,%20long,%20java.util.concurrent.TimeUnit)), [newTaskFor](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#newTaskFor(java.util.concurrent.Callable)), [newTaskFor](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#newTaskFor(java.lang.Runnable,%20T)), [submit](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#submit(java.util.concurrent.Callable)), [submit](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#submit(java.lang.Runnable)), [submit](http://docs.google.com/java/util/concurrent/AbstractExecutorService.html#submit(java.lang.Runnable,%20T)) |

| **Methods inherited from class java.lang.**[**Object**](http://docs.google.com/java/lang/Object.html) |
| --- |
| [clone](http://docs.google.com/java/lang/Object.html#clone()), [equals](http://docs.google.com/java/lang/Object.html#equals(java.lang.Object)), [getClass](http://docs.google.com/java/lang/Object.html#getClass()), [hashCode](http://docs.google.com/java/lang/Object.html#hashCode()), [notify](http://docs.google.com/java/lang/Object.html#notify()), [notifyAll](http://docs.google.com/java/lang/Object.html#notifyAll()), [toString](http://docs.google.com/java/lang/Object.html#toString()), [wait](http://docs.google.com/java/lang/Object.html#wait()), [wait](http://docs.google.com/java/lang/Object.html#wait(long)), [wait](http://docs.google.com/java/lang/Object.html#wait(long,%20int)) |

| **Constructor Detail** |
| --- |

### ThreadPoolExecutor

public **ThreadPoolExecutor**(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit,  
 [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue)

Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory and rejected execution handler. It may be more convenient to use one of the [Executors](http://docs.google.com/java/util/concurrent/Executors.html) factory methods instead of this general purpose constructor.

**Parameters:**corePoolSize - the number of threads to keep in the pool, even if they are idle.maximumPoolSize - the maximum number of threads to allow in the pool.keepAliveTime - when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.unit - the time unit for the keepAliveTime argument.workQueue - the queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method. **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if corePoolSize or keepAliveTime less than zero, or if maximumPoolSize less than or equal to zero, or if corePoolSize greater than maximumPoolSize. [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if workQueue is null

### ThreadPoolExecutor

public **ThreadPoolExecutor**(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit,  
 [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue,  
 [ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) threadFactory)

Creates a new ThreadPoolExecutor with the given initial parameters and default rejected execution handler.

**Parameters:**corePoolSize - the number of threads to keep in the pool, even if they are idle.maximumPoolSize - the maximum number of threads to allow in the pool.keepAliveTime - when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.unit - the time unit for the keepAliveTime argument.workQueue - the queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method.threadFactory - the factory to use when the executor creates a new thread. **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if corePoolSize or keepAliveTime less than zero, or if maximumPoolSize less than or equal to zero, or if corePoolSize greater than maximumPoolSize. [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if workQueue or threadFactory are null.

### ThreadPoolExecutor

public **ThreadPoolExecutor**(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit,  
 [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue,  
 [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) handler)

Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory.

**Parameters:**corePoolSize - the number of threads to keep in the pool, even if they are idle.maximumPoolSize - the maximum number of threads to allow in the pool.keepAliveTime - when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.unit - the time unit for the keepAliveTime argument.workQueue - the queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method.handler - the handler to use when execution is blocked because the thread bounds and queue capacities are reached. **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if corePoolSize or keepAliveTime less than zero, or if maximumPoolSize less than or equal to zero, or if corePoolSize greater than maximumPoolSize. [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if workQueue or handler are null.

### ThreadPoolExecutor

public **ThreadPoolExecutor**(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit,  
 [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> workQueue,  
 [ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) threadFactory,  
 [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) handler)

Creates a new ThreadPoolExecutor with the given initial parameters.

**Parameters:**corePoolSize - the number of threads to keep in the pool, even if they are idle.maximumPoolSize - the maximum number of threads to allow in the pool.keepAliveTime - when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.unit - the time unit for the keepAliveTime argument.workQueue - the queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method.threadFactory - the factory to use when the executor creates a new thread.handler - the handler to use when execution is blocked because the thread bounds and queue capacities are reached. **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if corePoolSize or keepAliveTime less than zero, or if maximumPoolSize less than or equal to zero, or if corePoolSize greater than maximumPoolSize. [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if workQueue or threadFactory or handler are null.

| **Method Detail** |
| --- |

### execute

public void **execute**([Runnable](http://docs.google.com/java/lang/Runnable.html) command)

Executes the given task sometime in the future. The task may execute in a new thread or in an existing pooled thread. If the task cannot be submitted for execution, either because this executor has been shutdown or because its capacity has been reached, the task is handled by the current RejectedExecutionHandler.

**Parameters:**command - the task to execute **Throws:** [RejectedExecutionException](http://docs.google.com/java/util/concurrent/RejectedExecutionException.html) - at discretion of RejectedExecutionHandler, if task cannot be accepted for execution [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if command is null

### shutdown

public void **shutdown**()

Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Invocation has no additional effect if already shut down.

**Throws:** [SecurityException](http://docs.google.com/java/lang/SecurityException.html) - if a security manager exists and shutting down this ExecutorService may manipulate threads that the caller is not permitted to modify because it does not hold [RuntimePermission](http://docs.google.com/java/lang/RuntimePermission.html)("modifyThread"), or the security manager's checkAccess method denies access.

### shutdownNow

public [List](http://docs.google.com/java/util/List.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> **shutdownNow**()

Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. These tasks are drained (removed) from the task queue upon return from this method.

There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. This implementation cancels tasks via [Thread.interrupt()](http://docs.google.com/java/lang/Thread.html#interrupt()), so any task that fails to respond to interrupts may never terminate.

**Returns:**list of tasks that never commenced execution **Throws:** [SecurityException](http://docs.google.com/java/lang/SecurityException.html) - if a security manager exists and shutting down this ExecutorService may manipulate threads that the caller is not permitted to modify because it does not hold [RuntimePermission](http://docs.google.com/java/lang/RuntimePermission.html)("modifyThread"), or the security manager's checkAccess method denies access.

### isShutdown

public boolean **isShutdown**()

**Description copied from interface:** [**ExecutorService**](http://docs.google.com/java/util/concurrent/ExecutorService.html#isShutdown()) Returns true if this executor has been shut down.

**Returns:**true if this executor has been shut down

### isTerminating

public boolean **isTerminating**()

Returns true if this executor is in the process of terminating after shutdown or shutdownNow but has not completely terminated. This method may be useful for debugging. A return of true reported a sufficient period after shutdown may indicate that submitted tasks have ignored or suppressed interruption, causing this executor not to properly terminate.

**Returns:**true if terminating but not yet terminated

### isTerminated

public boolean **isTerminated**()

**Description copied from interface:** [**ExecutorService**](http://docs.google.com/java/util/concurrent/ExecutorService.html#isTerminated()) Returns true if all tasks have completed following shut down. Note that isTerminated is never true unless either shutdown or shutdownNow was called first.

**Returns:**true if all tasks have completed following shut down

### awaitTermination

public boolean **awaitTermination**(long timeout,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

**Description copied from interface:** [**ExecutorService**](http://docs.google.com/java/util/concurrent/ExecutorService.html#awaitTermination(long,%20java.util.concurrent.TimeUnit)) Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

**Parameters:**timeout - the maximum time to waitunit - the time unit of the timeout argument **Returns:**true if this executor terminated and false if the timeout elapsed before termination **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if interrupted while waiting

### finalize

protected void **finalize**()

Invokes shutdown when this executor is no longer referenced.

**Overrides:**[finalize](http://docs.google.com/java/lang/Object.html#finalize()) in class [Object](http://docs.google.com/java/lang/Object.html)

### setThreadFactory

public void **setThreadFactory**([ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) threadFactory)

Sets the thread factory used to create new threads.

**Parameters:**threadFactory - the new thread factory **Throws:** [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if threadFactory is null**See Also:**[getThreadFactory()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getThreadFactory())

### getThreadFactory

public [ThreadFactory](http://docs.google.com/java/util/concurrent/ThreadFactory.html) **getThreadFactory**()

Returns the thread factory used to create new threads.

**Returns:**the current thread factory**See Also:**[setThreadFactory(java.util.concurrent.ThreadFactory)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setThreadFactory(java.util.concurrent.ThreadFactory))

### setRejectedExecutionHandler

public void **setRejectedExecutionHandler**([RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) handler)

Sets a new handler for unexecutable tasks.

**Parameters:**handler - the new handler **Throws:** [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if handler is null**See Also:**[getRejectedExecutionHandler()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getRejectedExecutionHandler())

### getRejectedExecutionHandler

public [RejectedExecutionHandler](http://docs.google.com/java/util/concurrent/RejectedExecutionHandler.html) **getRejectedExecutionHandler**()

Returns the current handler for unexecutable tasks.

**Returns:**the current handler**See Also:**[setRejectedExecutionHandler(java.util.concurrent.RejectedExecutionHandler)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setRejectedExecutionHandler(java.util.concurrent.RejectedExecutionHandler))

### setCorePoolSize

public void **setCorePoolSize**(int corePoolSize)

Sets the core number of threads. This overrides any value set in the constructor. If the new value is smaller than the current value, excess existing threads will be terminated when they next become idle. If larger, new threads will, if needed, be started to execute any queued tasks.

**Parameters:**corePoolSize - the new core size **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if corePoolSize less than zero**See Also:**[getCorePoolSize()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getCorePoolSize())

### getCorePoolSize

public int **getCorePoolSize**()

Returns the core number of threads.

**Returns:**the core number of threads**See Also:**[setCorePoolSize(int)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setCorePoolSize(int))

### prestartCoreThread

public boolean **prestartCoreThread**()

Starts a core thread, causing it to idly wait for work. This overrides the default policy of starting core threads only when new tasks are executed. This method will return false if all core threads have already been started.

**Returns:**true if a thread was started

### prestartAllCoreThreads

public int **prestartAllCoreThreads**()

Starts all core threads, causing them to idly wait for work. This overrides the default policy of starting core threads only when new tasks are executed.

**Returns:**the number of threads started

### allowsCoreThreadTimeOut

public boolean **allowsCoreThreadTimeOut**()

Returns true if this pool allows core threads to time out and terminate if no tasks arrive within the keepAlive time, being replaced if needed when new tasks arrive. When true, the same keep-alive policy applying to non-core threads applies also to core threads. When false (the default), core threads are never terminated due to lack of incoming tasks.

**Returns:**true if core threads are allowed to time out, else false**Since:** 1.6

### allowCoreThreadTimeOut

public void **allowCoreThreadTimeOut**(boolean value)

Sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive. When false, core threads are never terminated due to lack of incoming tasks. When true, the same keep-alive policy applying to non-core threads applies also to core threads. To avoid continual thread replacement, the keep-alive time must be greater than zero when setting true. This method should in general be called before the pool is actively used.

**Parameters:**value - true if should time out, else false **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if value is true and the current keep-alive time is not greater than zero.**Since:** 1.6

### setMaximumPoolSize

public void **setMaximumPoolSize**(int maximumPoolSize)

Sets the maximum allowed number of threads. This overrides any value set in the constructor. If the new value is smaller than the current value, excess existing threads will be terminated when they next become idle.

**Parameters:**maximumPoolSize - the new maximum **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if the new maximum is less than or equal to zero, or less than the [core pool size](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getCorePoolSize())**See Also:**[getMaximumPoolSize()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getMaximumPoolSize())

### getMaximumPoolSize

public int **getMaximumPoolSize**()

Returns the maximum allowed number of threads.

**Returns:**the maximum allowed number of threads**See Also:**[setMaximumPoolSize(int)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setMaximumPoolSize(int))

### setKeepAliveTime

public void **setKeepAliveTime**(long time,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)

Sets the time limit for which threads may remain idle before being terminated. If there are more than the core number of threads currently in the pool, after waiting this amount of time without processing a task, excess threads will be terminated. This overrides any value set in the constructor.

**Parameters:**time - the time to wait. A time value of zero will cause excess threads to terminate immediately after executing tasks.unit - the time unit of the time argument **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if time less than zero or if time is zero and allowsCoreThreadTimeOut**See Also:**[getKeepAliveTime(java.util.concurrent.TimeUnit)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#getKeepAliveTime(java.util.concurrent.TimeUnit))

### getKeepAliveTime

public long **getKeepAliveTime**([TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)

Returns the thread keep-alive time, which is the amount of time that threads in excess of the core pool size may remain idle before being terminated.

**Parameters:**unit - the desired time unit of the result **Returns:**the time limit**See Also:**[setKeepAliveTime(long, java.util.concurrent.TimeUnit)](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#setKeepAliveTime(long,%20java.util.concurrent.TimeUnit))

### getQueue

public [BlockingQueue](http://docs.google.com/java/util/concurrent/BlockingQueue.html)<[Runnable](http://docs.google.com/java/lang/Runnable.html)> **getQueue**()

Returns the task queue used by this executor. Access to the task queue is intended primarily for debugging and monitoring. This queue may be in active use. Retrieving the task queue does not prevent queued tasks from executing.

**Returns:**the task queue

### remove

public boolean **remove**([Runnable](http://docs.google.com/java/lang/Runnable.html) task)

Removes this task from the executor's internal queue if it is present, thus causing it not to be run if it has not already started.

This method may be useful as one part of a cancellation scheme. It may fail to remove tasks that have been converted into other forms before being placed on the internal queue. For example, a task entered using submit might be converted into a form that maintains Future status. However, in such cases, method [purge()](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.html#purge()) may be used to remove those Futures that have been cancelled.

**Parameters:**task - the task to remove **Returns:**true if the task was removed

### purge

public void **purge**()

Tries to remove from the work queue all [Future](http://docs.google.com/java/util/concurrent/Future.html) tasks that have been cancelled. This method can be useful as a storage reclamation operation, that has no other impact on functionality. Cancelled tasks are never executed, but may accumulate in work queues until worker threads can actively remove them. Invoking this method instead tries to remove them now. However, this method may fail to remove tasks in the presence of interference by other threads.

### getPoolSize

public int **getPoolSize**()

Returns the current number of threads in the pool.

**Returns:**the number of threads

### getActiveCount

public int **getActiveCount**()

Returns the approximate number of threads that are actively executing tasks.

**Returns:**the number of threads

### getLargestPoolSize

public int **getLargestPoolSize**()

Returns the largest number of threads that have ever simultaneously been in the pool.

**Returns:**the number of threads

### getTaskCount

public long **getTaskCount**()

Returns the approximate total number of tasks that have ever been scheduled for execution. Because the states of tasks and threads may change dynamically during computation, the returned value is only an approximation.

**Returns:**the number of tasks

### getCompletedTaskCount

public long **getCompletedTaskCount**()

Returns the approximate total number of tasks that have completed execution. Because the states of tasks and threads may change dynamically during computation, the returned value is only an approximation, but one that does not ever decrease across successive calls.

**Returns:**the number of tasks

### beforeExecute

protected void **beforeExecute**([Thread](http://docs.google.com/java/lang/Thread.html) t,  
 [Runnable](http://docs.google.com/java/lang/Runnable.html) r)

Method invoked prior to executing the given Runnable in the given thread. This method is invoked by thread t that will execute task r, and may be used to re-initialize ThreadLocals, or to perform logging.

This implementation does nothing, but may be customized in subclasses. Note: To properly nest multiple overridings, subclasses should generally invoke super.beforeExecute at the end of this method.

**Parameters:**t - the thread that will run task r.r - the task that will be executed.

### afterExecute

protected void **afterExecute**([Runnable](http://docs.google.com/java/lang/Runnable.html) r,  
 [Throwable](http://docs.google.com/java/lang/Throwable.html) t)

Method invoked upon completion of execution of the given Runnable. This method is invoked by the thread that executed the task. If non-null, the Throwable is the uncaught RuntimeException or Error that caused execution to terminate abruptly.

**Note:** When actions are enclosed in tasks (such as [FutureTask](http://docs.google.com/java/util/concurrent/FutureTask.html)) either explicitly or via methods such as submit, these task objects catch and maintain computational exceptions, and so they do not cause abrupt termination, and the internal exceptions are *not* passed to this method.

This implementation does nothing, but may be customized in subclasses. Note: To properly nest multiple overridings, subclasses should generally invoke super.afterExecute at the beginning of this method.

**Parameters:**r - the runnable that has completed.t - the exception that caused termination, or null if execution completed normally.

### terminated

protected void **terminated**()

Method invoked when the Executor has terminated. Default implementation does nothing. Note: To properly nest multiple overridings, subclasses should generally invoke super.terminated within this method.

| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/ThreadPoolExecutor.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/java/util/concurrent/ThreadFactory.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/ThreadPoolExecutor.AbortPolicy.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/ThreadPoolExecutor.html)    [**NO FRAMES**](http://docs.google.com/ThreadPoolExecutor.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: [NESTED](#3znysh7) | FIELD | [CONSTR](#2et92p0) | [METHOD](#tyjcwt) | DETAIL: FIELD | [CONSTR](#4d34og8) | [METHOD](#lnxbz9) |

[Submit a bug or feature](http://bugs.sun.com/services/bugreport/index.jsp)

For further API reference and developer documentation, see [Java SE Developer Documentation](http://docs.google.com/webnotes/devdocs-vs-specs.html). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2006 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](http://docs.google.com/legal/license.html). Also see the [documentation redistribution policy](http://java.sun.com/docs/redist.html).