**ExecutorService Example**

Here is a simple Java ExectorService example:

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {

public void run() {

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

}

});

executorService.shutdown();

First an ExecutorService is created using the newFixedThreadPool() factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the Runnable interface is passed to the execute() method. This causes the Runnable to be executed by one of the threads in the ExecutorService.

**Task Delegation**

Here is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:

|  |
| --- |
|  |
| **A thread delegating a task to an ExecutorService for asynchronous execution.** |

Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

**ExecutorService Implementations**

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. The ExecutorService has the following implementation in the java.util.concurrent package:

* [ThreadPoolExecutor](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)
* [ScheduledThreadPoolExecutor](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

**Creating an ExecutorService**

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too. Here are a few examples of creating an ExecutorService:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

**ExecutorService Usage**

There are a few different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

I will take a look at each of these methods in the following sections.

**execute(Runnable)**

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

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).

**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.

Here is a ExecutorService submit() example:

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

public void run() {

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

}

});

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

**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. Here is an ExecutorService Callable example:

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

**invokeAny()**

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

Here is a code example:

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";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

**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.

Here is a code example:

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();

**ExecutorService Shutdown**

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown() is called, are executed.

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

# ThreadPoolExecutor

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The java.util.concurrent.ThreadPoolExecutor is an implementation of the **[ExecutorService](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html)** interface. The ThreadPoolExecutor executes the given task (Callable or Runnable) using one of its internally pooled threads.

The thread pool contained inside the ThreadPoolExecutor can contain a varying amount of threads. The number of threads in the pool is determined by these variables:

* corePoolSize
* maximumPoolSize

If less than corePoolSize threads are created in the the thread pool when a task is delegated to the thread pool, then a new thread is created, even if idle threads exist in the pool.

If the internal queue of tasks is full, and corePoolSize threads or more are running, but less than maximumPoolSize threads are running, then a new thread is created to execute the task.

Here is a diagram illustrating the ThreadPoolExecutor principles:

|  |
| --- |
|  |
| **A ThreadPoolExecutor** |

## Creating a ThreadPoolExecutor

The ThreadPoolExecutor has several constructors available. For instance:

int corePoolSize = 5;

int maxPoolSize = 10;

long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =

new ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()

);

However, unless you need to specify all these parameters explicitly for your ThreadPoolExecutor, it is often easier to use one of the factory methods in the java.util.concurrent.Executors class, as shown in the

# ScheduledExecutorService

* [ScheduledExecutorService Example](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-example)
* [ScheduledExecutorService Implementations](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-implementations)
* [Creating a ScheduledExecutorService](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#creating-a-scheduledexecutorservice)
* [ScheduledExecutorService Usage](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-usage)
  + [schedule (Callable task, long delay, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#schedule-callable)
  + [schedule (Runnable task, long delay, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#schedule-runnable)
  + [scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduleatfixedrate)
  + [scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#schedulewithfixeddelay)
* [ScheduledExecutorService Shutdown](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-shutdown)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The java.util.concurrent.ScheduledExecutorService is an **[ExecutorService](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html)** which can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution. Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to the ScheduledExecutorService.

## ScheduledExecutorService Example

Here is a simple ScheduledExecutorService example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

First a ScheduledExecutorService is created with 5 threads in. Then an anonymous implementation of the Callable interface is created and passed to the schedule() method. The two last parameters specify that the Callable should be executed after 5 seconds.

## ScheduledExecutorService Implementations

Since ScheduledExecutorService is an interface, you will have to use its implementation in the java.util.concurrent package, in order to use it. ScheduledExecutorService as the following implementation:

* ScheduledThreadPoolExecutor

## Creating a ScheduledExecutorService

How you create an ScheduledExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ScheduledExecutorService instances too. Here is an example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

## ScheduledExecutorService Usage

Once you have created a ScheduledExecutorService you use it by calling one of its methods:

* schedule (Callable task, long delay, TimeUnit timeunit)
* schedule (Runnable task, long delay, TimeUnit timeunit)
* scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
* scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

I will briefly cover each of these methods below.

### schedule (Callable task, long delay, TimeUnit timeunit)

This method schedules the given Callable for execution after the given delay.

The method returns a ScheduledFuture which you can use to either cancel the task before it has started executing, or obtain the result once it is executed.

Here is an example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

System.out.println("result = " + scheduledFuture.get());

scheduledExecutorService.shutdown();

This example outputs:

Executed!

result = Called!

### schedule (Runnable task, long delay, TimeUnit timeunit)

This method works like the method version taking a Callable as parameter, except a Runnable cannot return a value, so the ScheduledFuture.get() method returns null when the task is finished.

### scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

This method schedules a task to be executed periodically. The task is executed the first time after the initialDelay, and then recurringly every time the period expires.

If any execution of the given task throws an exception, the task is no longer executed. If no exceptions are thrown, the task will continue to be executed until the ScheduledExecutorService is shut down.

If a task takes longer to execute than the period between its scheduled executions, the next execution will start after the current execution finishes. The scheduled task will not be executed by more than one thread at a time.

### scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

This method works very much like scheduleAtFixedRate() except that the period is interpreted differently.

In the scheduleAtFixedRate() method the period is interpreted as a delay between the start of the previous execution, until the start of the next execution.

In this method, however, the period is interpreted as the delay between the **end** of the previous execution, until the start of the next. The delay is thus between finished executions, not between the beginning of executions.

## ScheduledExecutorService Shutdown

Just like an ExecutorService, the ScheduledExecutorService needs to be shut down when you are finished using it. If not, it will keep the JVM running, even when all other threads have been shut down.

You shut down a ScheduledExecutorService using the shutdown() or shutdownNow() methods which are inherited from the ExecutorService interface. See the **[ExecutorService Shutdown](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html" \l "executorservice-shutdown)** section for more