Executor Service

[*Executor Service*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) is a JDK API that simplifies running tasks in asynchronous mode. Generally speaking, Executor Service automatically provides a pool of threads and an API for assigning tasks to it

### ****Factory Methods of the****Executors****Class****

The easiest way to create *ExecutorService* is to use one of the factory methods of the *Executors* class.

For example, the following line of code will create a thread pool with 10 threads:

ExecutorService executor = Executors.newFixedThreadPool(10);

There are several other factory methods to create a predefined*ExecutorService* that meets specific use cases. To find the best method for your needs, consult [Oracle's official documentation](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html).

### ****Directly Create an****ExecutorService

Because ExecutorService is an interface, an instance of any its implementations can be used. There are several implementations to choose from in the *[java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html)* package, or you can create your own.

For example, the ThreadPoolExecutor class has a few constructors that we can use to configure an executor service and its internal pool:

ExecutorService executorService =

**new** ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,

**new** LinkedBlockingQueue<Runnable>());

You may notice that the code above is very similar to the [source code](https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/Executors.java#L133) of the factory method newSingleThreadExecutor(). For most cases, a detailed manual configuration isn't necessary.

## ****Assigning Tasks to the****ExecutorService

ExecutorService can execute Runnable and Callable tasks. To keep things simple in this article, two primitive tasks will be used. Notice that we use lambda expressions here instead of anonymous inner classes:

Runnable runnableTask = () -> {

**try** {

TimeUnit.MILLISECONDS.sleep(300);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

};

Callable<String> callableTask = () -> {

TimeUnit.MILLISECONDS.sleep(300);

**return** "Task's execution";

};

List<Callable<String>> callableTasks = **new** ArrayList<>();

callableTasks.add(callableTask);

callableTasks.add(callableTask);

callableTasks.add(callableTask);

We can assign tasks to the ExecutorService using several methods including execute(), which is inherited from the Executor interface, and also submit(), invokeAny() and invokeAll().

The **execute()** method is void and doesn't give any possibility to get the result of a task's execution or to check the task's status (is it running):

executorService.execute(runnableTask);

**submit()** submits a Callable or a Runnable task to an ExecutorService and returns a result of type Future:

Future<String> future = executorService.submit(callableTask);

**invokeAny()** assigns a collection of tasks to an ExecutorService, causing each to run, and returns the result of a successful execution of one task (if there was a successful execution):

String result = executorService.invokeAny(callableTasks);

***invokeAll()*** assigns a collection of tasks to an ExecutorService, causing each to run, and returns the result of all task executions in the form of a list of objects of type Future:

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

Before going further, we need to discuss two more items: shutting down an ExecutorService and dealing with Future return types.

## ****Shutting Down an****ExecutorService

 The ExecutorService will not be automatically destroyed when there is no task to process. It will stay alive and wait for new work to do.

On the other hand, an app could reach its end but not be stopped because a waiting ExecutorService will cause the JVM to keep running.

To properly shut down an ExecutorService, we have the shutdown() and shutdownNow() APIs.

The ***shutdown()*** method doesn't cause immediate destruction of the ExecutorService. It will make the ExecutorService stop accepting new tasks and shut down after all running threads finish their current work:

executorService.shutdown();

The **shutdownNow()** method tries to destroy the ExecutorService immediately, but it doesn't guarantee that all the running threads will be stopped at the same time:

List<Runnable> notExecutedTasks = executorService.shutDownNow();

One good way to shut down the ExecutorService (which is also [recommended by Oracle](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html)) is to use both of these methods combined with the **awaitTermination()** method:

executorService.shutdown();

**try** {

**if** (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {

executorService.shutdownNow();

}

} **catch** (InterruptedException e) {

executorService.shutdownNow();

}

With this approach, the ExecutorService will first stop taking new tasks and then wait up to a specified period of time for all tasks to be completed. If that time expires, the execution is stopped immediately.

## ****The****Future****Interface****

The submit() and invokeAll() methods return an object or a collection of objects of type Future, which allows us to get the result of a task's execution or to check the task's status (is it running)

The Future interface provides a special blocking method get(), which returns an actual result of the Callable task's execution or null in the case of a Runnable task

Future<String> future = executorService.submit(callableTask);

String result = **null**;

**try** {

result = future.get();

} **catch** (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

Calling the get() method while the task is still running will cause execution to block until the task properly executes and the result is available.