## ****What is the Executor Framework? Why use it?****

The Executor Framework contains a bunch of components that are used to efficiently manage multiple threads. It was released with the JDK 5 which is used to run the Runnable objects without creating new threads every time and also mostly re-using the already created threads.

This Executor API de-couples the execution of a task from the actual task to be executed with the help of an Executor. This is centered around the Executor interface and its sub-interface ***ExecutorService*** and the class ***ThreadPoolExecutor.***

By using this executor, only one has to implement the runnable objects and send them to the executor for execution.

In this example, the Test class implements Runnable and is parameterized to type string. It is also declared to throw Exception. Also, note, this ability to throw an exception to the executor and the executor returning this exception back to the runner is of great importance because it helps the runner know the status of task execution.

Let’s move ahead to the next section of this article and take a look at the different types of Executor frameworks in Java.

## ****Types of Executors****

There are mainly 4 types of Executors available. They are namely:

* **SingleThreadExecutor**
* **FixedThreadPool**
* **CachedThreadPool**
* **Scheduled****Executor**

**SingleThreadExecutor**

This executor has only one thread and is used to execute tasks in a sequential manner. If any thread dies due to an exception while executing the task, a new thread is created to replace the old thread and the subsequent tasks are executed in the new thread.

**FixedPoolExecutor**

This is a pool of a fixed number of threads. The tasks submitted to the executor are executed by the “n” threads and suppose if there is more task to finish, they are stored on a LinkedBlockingQueue.

**CachedThreadExecutor**

This is mainly used when there are lots of short-lived parallel tasks on the line waiting to be executed. When compared with the fixed thread pool, here, the number of threads of this executor pool is not bounded. If all the threads are busy executing the assigned tasks and when there is a new task, it will create and add a new thread to the executor. If a thread remains idle for close to sixty seconds, they are terminated and removed from the cache.

**ScheduledExecutor**

This executor is used when you have a task that needs to be run at regular intervals or if in case you wish to delay a certain task. The tasks can be scheduled in ScheduledExecutor using either of the two methods scheduleAtFixedRate or scheduleWithFixedDelay.

**Benefits of Executor Framework**

* The framework mainly separates task creation and execution. Task creation is mainly boilerplate code and is easily replaceable.
* With an executor, we have to create tasks that implement either Runnable or Callable interface and send them to the executor.
* Executor internally maintains a (configurable) thread pool to improve application performance by avoiding the continuous spawning of threads.
* Executor is responsible for executing the tasks, and running them with the necessary threads from the pool.

**1.2. Callable and Future**

Another important advantage of the Executor framework is the use of the **Callable** interface. It’s similar to the Runnable interface with two benefits:

1. It’s call() method returns a result after the thread execution is complete.

When we send a Callable object to an executor, we get a Future object’s reference. We can use this object to query the status of the thread and the result of the Callable object.

**Creating ExecutorService Instance**

ExecutorService is an interface and its implementations can execute a Runnable or Callable class in an asynchronous way. Note that invoking the run() method of a Runnable interface in a synchronous way is simply calling a method.

We can create an instance of ExecutorService in following ways:

**2.1. Using *Executors***

Executors is a utility class that provides factory methods for creating the implementations of the interface.

***//Executes only one thread***

**ExecutorService es = Executors.newSingleThreadExecutor();**

***//Internally manages thread pool of 2 threads***

**ExecutorService es = Executors.newFixedThreadPool(2);**

***//Internally manages thread pool of 10 threads to run scheduled tasks***

**ExecutorService es = Executors.newScheduledThreadPool(10);**

**2.2. Using Constructors**

We can choose an implementation class of ExecutorService interface and create its instance directly. The below statement creates a thread pool executor with a minimum thread count 10, maximum threads count 100 and 5 milliseconds keep alive time and a blocking queue to watch for tasks in future.

**ExecutorService executorService = new ThreadPoolExecutor(10, 100, 5L, TimeUnit.MILLISECONDS,**

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

**3. Submitting Tasks to ExecutorService**

Generally, tasks are created by implementing either Runnable or Callable interface. Let’s see the example of both cases.

**3.1. Executing *Runnable* Tasks**

We can execute runnables using the following methods :

* **void execute(Runnable task)** – executes the given command at some time in the future.
* **Future submit(Runnable task)** – submits a runnable task for execution and returns a Future representing that task. The Future’s get() method will return null upon successful completion.
* **Future submit(Runnable task, T result)** – Submits a runnable task for execution and returns a Future representing that task. The Future’s get() method will return the given result upon successful completion.

In given example, we are executing a task of type Runnable using both methods.

**import** java.time.LocalDateTime;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**import** java.util.concurrent.TimeUnit;

**public** **class** Main

{

**public** **static** **void** main(String[] args)

{

*//Demo task*

Runnable runnableTask = () -> {

**try** {

TimeUnit.MILLISECONDS.sleep(1000);

System.out.println("Current time :: " + LocalDateTime.now());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

};

*//Executor service instance*

ExecutorService executor = Executors.newFixedThreadPool(10);

*//1. execute task using execute() method*

executor.execute(runnableTask);

*//2. execute task using submit() method*

Future<String> result = executor.submit(runnableTask, "DONE");

**while**(result.isDone() == false)

{

**try**

{

System.out.println("The method return value : " + result.get());

**break**;

}

**catch** (InterruptedException | ExecutionException e)

{

e.printStackTrace();

}

*//Sleep for 1 second*

**try** {

Thread.sleep(1000L);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

*//Shut down the executor service*

executor.shutdownNow();

}

}

Program output.

Current time :: 2019-05-21T17:52:53.274

Current time :: 2019-05-21T17:52:53.274

The method return value : DONE

**3.2. Execute *Callable* Tasks**

We can execute callable tasks using the following methods :

* **Future submit(callableTask)** – submits a value-returning task for execution and returns a Future representing the pending results of the task.
* **List<Future> invokeAll(Collection tasks)** – executes the given tasks, returning a list of Futures holding their status and results **when all complete**. Notice that result is available only when all tasks are completed.  
  Note that a completed task could have terminated either normally or by throwing an exception.
* **List<Future> invokeAll(Collection tasks, timeOut, timeUnit)** – executes the given tasks, returning a list of Futures holding their status and results **when all complete or the timeout expires**.

In given example, we are executing a task of type Callable using both methods.

**import** java.time.LocalDateTime;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**import** java.util.concurrent.TimeUnit;

**public** **class** Main

{

**public** **static** **void** main(String[] args) **throws** ExecutionException

{

*//Demo Callable task*

Callable<String> callableTask = () -> {

TimeUnit.MILLISECONDS.sleep(1000);

**return** "Current time :: " + LocalDateTime.now();

};

*//Executor service instance*

ExecutorService executor = Executors.newFixedThreadPool(1);

List<Callable<String>> tasksList = Arrays.asList(callableTask, callableTask, callableTask);

*//1. execute tasks list using invokeAll() method*

**try**

{

List<Future<String>> results = executor.invokeAll(tasksList);

**for**(Future<String> result : results) {

System.out.println(result.get());

}

}

**catch** (InterruptedException e1)

{

e1.printStackTrace();

}

*//2. execute individual tasks using submit() method*

Future<String> result = executor.submit(callableTask);

**while**(result.isDone() == false)

{

**try**

{

System.out.println("The method return value : " + result.get());

**break**;

}

**catch** (InterruptedException | ExecutionException e)

{

e.printStackTrace();

}

*//Sleep for 1 second*

**try** {

Thread.sleep(1000L);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

*//Shut down the executor service*

executor.shutdownNow();

}

}

Program output.

Current time :: 2019-05-21T18:35:53.512

Current time :: 2019-05-21T18:35:54.513

Current time :: 2019-05-21T18:35:55.514

The method return value : Current time :: 2019-05-21T18:35:56.515

Notice that tasks have been completed with a delay of 1 second because there is only one task in the thread pool. But when you run the program, all first 3 print statements appear at the same time because even if the tasks are complete, they wait for other tasks to complete in the list.

**4. How to Shutdown *ExecutorService***

The final and most important thing that many developers miss is shutting down the ExecutorService. The ExecutorService is created and it has Thread elements.

Remember that the JVM stops only when all non-daemon threads are stopped. Here not shutting down the executor service simply prevents the JVM from stopping.

In the above examples, if we comment out the **executor.shutdownNow()** method call, then even after all tasks are executed, main thread remains active and JVM does not stop.

To tell the executor service that there is no need for the threads it has, we will have to shutdown the service.

There are three methods to invoke shutdown:

* **void shutdown()** – Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
* **List<Runnable> 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.
* **void awaitTermination()** – It blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

Use any of the above 3 methods wisely as per the requirements of the application.

**5. Best Practices**

* Always run your Java code against static analysis tools like PMD and FindBugs to look for deeper issues. They are very helpful in determining ugly situations which may arise in the future.
* Always cross-check and better plan a code review with senior guys to detect possible deadlock or livelock in code during execution. Adding a health monitor to your application to check the status of running tasks is an excellent choice in most scenarios.
* In multi-threaded programs, make a habit of catching errors too, not just exceptions. Sometimes unexpected things happen, and Java throws an error at you, apart from an exception.
* Use a back-off switch, so if something goes wrong and is non-recoverable, you don’t escalate the situation by eagerly starting another loop. Instead, you need to wait until the situation goes back to normal and then start again.
* Please note that the whole point of executors is to abstract away the specifics of execution, so ordering is not guaranteed unless explicitly stated.

**6. Conclusion**

As discussed above, ExecutorService helps in minimizing the boilerplate code which is a good thing. It also helps in better resource management by internally utilizing a thread pool.

Still, programmers should be careful to avoid some common mistakes. E.g. always shut down the executor service after tasks are completed and service is no longer needed. Otherwise, JVM will never terminate, normally.

Similarly, while creating it’s instance, be mindful of the configured thread pool capacity. Here or in any other implementation, a careless threads pool size can halt the system and bring performance down.

And finally, make a practice of using *timeout* parameters in blocking method calls. These methods can block the whole application execution if not returned in small time.