**ExecutorService in Java**

Learn to use Java **ExecutorService** to execute a [Runnable](https://howtodoinjava.com/java/multi-threading/java-runnable-vs-thread/) or [Callable](https://howtodoinjava.com/java/multi-threading/java-callable-future-example/) class in an asynchronous way. Also learn the various best practices to utilize it in most efficient manner in any Java application.

**1. What is Executor Framework?**

In simple Java applications, we do not face much challenge while working with a small number of threads. If you have to develop a program that runs a lot of concurrent tasks, this approach will present many disadvantages such as lots of boiler plate code (create and manage threads), executing thread manually and keeping track of thread execution results.

[Executor framework](https://howtodoinjava.com/java/multi-threading/executor-framework-tutorial/) (since Java 1.5) solved this problem. The framework consist of three main interfaces (and lots of child interfaces) i.e. **Executor**, **ExecutorService** and **[ThreadPoolExecutor](https://howtodoinjava.com/java/multi-threading/java-thread-pool-executor-example/)**.

**1.1. Benefits of Executor framework**

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

**1.2. Callable and Future**

Another important advantage of the Executor framework was 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.
2. 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 thread and the result of the Callable object.

**2. Creating ExecutorService instances**

ExecutorService is an interface and it’s 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. Executors class**

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. Constructors**

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

|  |
| --- |
| import java.util.concurrent.ExecutorService;  import java.util.concurrent.LinkedBlockingQueue;  import java.util.concurrent.ThreadPoolExecutor;  import java.util.concurrent.TimeUnit;    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. Execute 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.

|  |
| --- |
| ExecutorService Runnable Example |
| 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.

|  |
| --- |
| Console |
| 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.

|  |
| --- |
| ExecutorService Callable Example |
| 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.

|  |
| --- |
| Console |
| 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 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 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 above 3 methods wisely as per requirements of the application.

**5. Conclusion**

As discussed above, ExecutorService helps in minimizing the boiler plate 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 shutdown 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 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.