**Executor FrameWork**

**Executors**

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as *executors*. The following subsections describe executors in detail.

* [**Executor Interfaces**](https://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html) define the three executor object types.
* [**Thread Pools**](https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html)are the most common kind of executor implementation.
* [**Fork/Join**](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html) is a framework (new in JDK 7) for taking advantage of multiple processors.

**Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.**

## **The Executor Interface:**

* The [Executor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html) interface provides a **single method, execute**, designed to be a drop-in replacement for a common thread-creation idiom.

If **r** is a Runnable object, and **e** is an Executor object **you can replace**

**(new Thread(r)).start();** with **e.execute(r);**

* However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on [Thread Pools](https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html).)
* The executor implementations in java.util.concurrent are designed to make full use of the more advanced ExecutorService and ScheduledExecutorService interfaces, although they also work with the base Executor interface.

## **The ExecutorService Interface:**

* The [ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html" \t "_blank) interface supplements execute with a similar, but more versatile submit method. **Like execute, submit accepts Runnable objects, but also accepts**[**Callable**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)**objects, which allow the task to return a value. The submit method returns a**[**Future**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)**object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.**
* ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle [interrupts](https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html) correctly.

## **The ScheduledExecutorService Interface**

* The [ScheduledExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html" \t "_blank) interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

**What is Callable and futures?**

* **Link:** <https://stackabuse.com/concurrency-in-java-the-executor-framework/>
* **Understanding the Future Object**
  + The result of the task submitted for execution to an executor can be accessed using the java.util.concurrent.Future object returned by the executor. Future can be thought of as a promise made to the caller by the executor.

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

* + A task submitted to the executor, like above, is asynchronous i.e. the program execution does not wait for the completion of task execution to proceed to next step. Instead, whenever the task execution is completed, it is set in this Future object by the executor.
* The caller can continue executing the main program and when the result of the submitted task is needed he can call .get() on this Future object. If the task is complete the result is immediately returned to the caller or else the caller is blocked until the execution of this is completed by the executor and the result is computed.
* If the caller cannot afford to wait indefinitely before retrieving the result, this wait can be timed as well. This is achieved by the Future.get(long timeout, TimeUnit unit) method which throws a TimeoutException if the result is not returned in the stipulated timeframe. The caller can handle this exception and continue with the further execution of the program.
* If there is an exception when executing the task, the call to get method will throw an ExecutionException.
* An important thing with respect to result being returned by Future.get() method is that it is returned only if the submitted task implements java.util.concurrent.Callable. If the task implements the Runnable interface, the call to .get() will return null once the task is complete.
* Another important method is the Future.cancel(boolean mayInterruptIfRunning) method. This method is used to cancel the execution of a submitted task. If the task is already executing, the executor will attempt to interrupt the task execution if the mayInterruptIfRunning flag is passed as true.

**Callable vs Runnable:**

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. **A Runnable, however, does not return a result and cannot throw a checked exception.**