1. **Java.util.concurrent.Executor Interface.**
2. **Java.util.concurrent.Executors Class.**
3. **Java.util.concurrent. Callable<V> interfaces**
4. **Java.util.concurrent. Future<V> interfaces**
5. **Java.util.concurrent. ExecutorService Interface**
6. **java.util.concurrent.ScheduledExecutorService interface**

java.util.concurrent.Executor

An **Executor** is an object that is responsible for threads management and execution of Runnable tasks submitted from the client code. It decouples the details of thread creation, scheduling, etc. from the task submission so you can focus on developing the task's business logic without caring about the thread management details.

That means, in the simplest case, rather than creating a thread to execute a task like this:

```
Thread t = new Thread (new RunnableTask());
t.start();
```

You submit tasks to an executor like this:

```
Executor executor = anExecutorImplementation;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

The Java Concurrency API defines the following 3 base interfaces for executors: -

- **Executor**: is the super type of all executors. It defines only one method execute (Runnable).
- **ExecutorService**: is an Executor that allows tracking progress of value-returning tasks (Callable) via Futureobject, and manages the termination of threads. Its key methods include submit () and shutdown ().
- **ScheduledExecutorService**: is an ExecutorService that can schedule tasks to execute after a given delay, or to execute periodically. Its key methods are schedule (), scheduleAtFixedRate() and scheduleWithFixedDelay().

Methods of Executor Interface: -

void execute (Runnable command)	Executes the given command at some time in the future. The command may execute in a new thread, in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.

java.util.concurrent.Executors

This class supports the following kinds of methods:

- Methods that create and return an ExecutorService set up with commonly useful configuration settings.

- Methods that create and return a ScheduledExecutorService set up with commonly useful configuration settings.
- Methods that create and return a "wrapped" ExecutorService, that disables reconfiguration by making implementation-specific methods inaccessible.
- Methods that create and return a ThreadFactory that sets newly created threads to a known state.
- Methods that create and return a Callable out of other closure-like forms, so they can be used in execution methods requiring Callable.

You can create an executor by using one of several factory methods provided by the **Executors** utility class. Here's to name a few:

- **newCachedThreadPool()**: creates an expandable thread pool executor. New threads are created as needed, and previously constructed threads are reused when they are available. Idle threads are kept in the pool for one minute. This executor is suitable for applications that launch many short-lived concurrent tasks.
- **newFixedThreadPool(int n)**: creates an executor with a fixed number of threads in the pool. This executor ensures that there are no more than n concurrent threads at any time. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread becomes available. If any thread terminates due to failure during execution, it will be replaced by a new one. The threads in the pool will exist until it is explicitly shutdown. Use this executor if you and to limit the maximum number of concurrent threads.
- **newSingleThreadExecutor()**: creates an executor that executes a single task at a time. Submitted tasks are guaranteed to execute sequentially, and no more than one task will be active at any time. Consider using this executor if you want to queue tasks to be executed in order, one after another.
- **newScheduledThreadPool(int corePoolSize)**: creates an executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute concurrently.
- **newSingleThreadScheduleExecutor()**: creates a single-threaded executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute sequentially.

In case the factory methods do not meet your need, you can construct an executor directly as an instance of either ThreadPoolExecutor or ScheduledThreadPoolExecutor, which gives you additional options such as pool size, on-demand construction, keep-alive times, etc.

For creating a Fork/Join pool, construct an instance of the ForkJoinPool class.

Java Callable and Future interfaces

One of the benefits of the Java executor framework is that we can run concurrent tasks that may return a single result after processing the tasks. The Java Concurrency API achieves this with the following two interfaces **Callable** and **Future**.

Callable interfaces

In Java 5, **java.util.concurrent** was introduced. **The callable** interface was introduced in concurrency package, which is similar to the Runnable interface, but it can return any **object, and is able to throw an Exception.**

The Java Callable interface uses Generics, so it can return any type of Object. The Executor Framework offers a submit () method to execute Callable implementations in a thread pool.

Actually, the Java Executor Framework follows WorkerThread patterns, wherein a thread pool you can initiate threads by using the
`Executors.newFixedThreadPool(10);`

method.

Then you can submit a task to it.

As you may remember in Java, a runnable acts as the target of a thread, and in the runnable interface, a `public void run()` method has to be implemented where you define the task, which will be executed by threads in the thread pool.

The Executor Framework assigns work (runnable target) to threads only if there is an available thread in the pool. If all threads are in use, the work has to wait. Once a task is completed by the thread, that thread returns to the pool as an available thread.

Callable is same as Runnable but it can return any type of Object if we want to get a result or status from work (callable).

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.

The Executors class contains utility methods to convert from other common forms to Callable classes.

Callable interface has the call() method. In this method, we have to implement the logic of a task. The Callable interface is a parameterized interface, meaning we have to indicate the type of data the call() method will return.

Methods: -

V call()	Computes a result, or throws an exception if unable to do so.

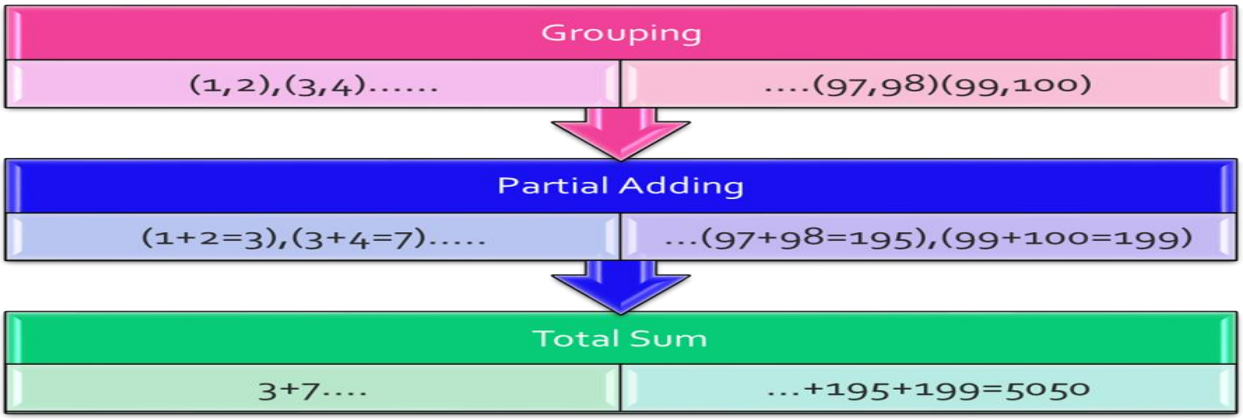
Future

Callable tasks return **java.util.concurrent.Future** objects. Java Future provides a `cancel()` method to cancel the associated Callable task. This is an overloaded version of the `get()` method, where we can specify the time to wait for the result. It's useful to avoid a current thread getting blocked for a longer time. Please note that the get method is a synchronous method. Until the callable finishes its task and returns a value, it will wait for a callable. There are also `isDone()` and `isCancelled()` methods to find out the current status of an associated Callable task.

Example: Suppose the problem is to find a sum of all numbers from 1 to 100. We can do it by looping 1 to 100 sequentially and adding them.

Another way we can do it by the *divide and Conquer* rule. Group the numbers in a way so each group has exactly two elements. Then Assign that group to a pool of threads

So each thread returns a partial sum in parallel. Then collect those partial sums and add them to get the whole sum.



Future interface has methods to obtain the result generated by a Callable object and to manage its state.

A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method.

Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.

Method Summary

Modifier and Type	Method and Description
boolean	`cancel(boolean mayInterruptIfRunning)` Attempts to cancel execution of this task.
V	`get()` Waits if necessary for the computation to complete, and then retrieves its result.
V	`get(long timeout, TimeUnit unit)` Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
boolean	`isCancelled()` Returns `true` if this task was cancelled before it completed normally.
boolean	`isDone()` Returns `true` if this task completed.

ExecutorService Interface

A java.util.concurrent.ExecutorService interface is a subinterface of Executor interface, and adds features to manage the lifecycle, both of the individual tasks and of the executor itself.

ExecutorService Methods

Method	Description
boolean awaitTermination(long timeout, TimeUnit unit)	Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)	Executes the given tasks, returning a list of Futures holding their status and results when all complete.
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.
<T> T invokeAny(Collection<? extends Callable<T>> tasks)	Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.

<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.
boolean isShutdown()	Returns true if this executor has been shut down.
boolean isTerminated()	Returns true if all tasks have completed following shut down.
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.
<T> Future<T> submit(Callable<T> task)	Submits a value-returning task for execution and returns a Future representing the pending results of the task.
Future<?> submit (Runnable task)	Submits a Runnable task for execution and returns a Future representing that task.
<T> Future<T> submit (Runnable task, T result)	Submits a Runnable task for execution and returns a Future representing that task.

java.util.concurrent.ScheduledExecutorService interface

A java.util.concurrent.ScheduledExecutorService interface is a subinterface of ExecutorService interface, and supports future and/or periodic execution of tasks.

ScheduledExecutorService Methods

Method	Description
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)	Creates and executes a ScheduledFuture that becomes enabled after the given delay.

ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)	Creates and executes a one-shot action that becomes enabled after the given delay.
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 * period, and so on.
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next.