[Main section:](#Main_Objective)

Objective Id . Objective

3 Executor Framework of java.util.concurrent package.

[Remarks:](#Notes)

[Sub Section:](#Sub_section)

- Use of interfaces Executor and ExecutorService and concept of Thread pools and use of important methods

of java.util.concurrent.Executors class.

[Referenced codebase:](#Notes) com.seed.concurrent.executor package

Demo#1

application:

JavaConcurrency\src\com\seed\concurrent\executor\TestExecutor.java

helper:

CalculateSumRunnable.java from same package

Required resources: Nill.

[Steps on how to run demos:](#Execution_steps)

Demo#1

Run the demo as ,

java TestExecutor

[Remarks:](#Remarks)

[Sub Section:](#Sub_section)

- Threads returning values with Callables and Futures . Future used to capture and get values

asynchronously from submitted Callables.

[Referenced codebase:](#Notes) com.seed.concurrent.executor package

Demo#2

application:

JavaConcurrency\src\com\seed\concurrent\executor\DemoCallableFuture.java

helper:

Above application uses

1. SquareCallable.java from same package uses following
2. SquareBean.java from same package

Required resources: Nill.

[Steps on how to run demos:](#Execution_steps)

Run the demo as,

Java DemoCallableFuture

Remarks :

[Remarks:](#Remarks)

[Sub Section:](#Sub_section)

- Understanding ScheduledExecutorService Interface.

[Referenced codebase:](#Notes) com.seed.concurrent.executor package

Demo#3

application:

JavaConcurrency\src\com\seed\concurrent\executor\DemoScheduledExecutorService.java

helper:

Required resources: Nill.

[Steps on how to run demos:](#Execution_steps)

Demo#3

Run the demo as ,

java DemoScheduledExecutorService

Remarks :

[Remarks:](#Remarks)

[Notes:](#Notes)

Purpose of Executor :

----------------------------------------

Generally in small scale multithreaded java applications we create the threads as ,

Thread t = new Thread(new Runnable()) ;

Here we find that thread instance is bound with Runnable task or in other words here Runnable task is

executed by the Thread synchronously.

But in large scale multithreaded java applications, exeuting each Runnale task synchronously

by a seperate Thread is not feasible because threads are relatively expensive to create, and

the system may have limitations on number of threads to be active simultaneously.

Hence it makes sense to separate thread creation and management from the rest of the application.

java.util.concurrent package has introduced Executors for this purpose.

Executor Interfaces

-------------------

The java.util.concurrent package defines three executor interfaces:

1. Executor : a simple interface that supports launching new tasks.

The Executor interface provides a single method, execute().

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

replace

(new Thread(r)).start(); with

e.execute(r);

2. ExecutorService : a subinterface of Executor, which adds features

that help to manage the lifecycle, both of the individual tasks and

of the executor itself.

ExecutorService can be shutdown. Some of the important methods of

this class are given below.

- 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.

For complete API of this Interface, you can refer to ,

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html

3. ScheduledExecutorService :

A subinterface of ExecutorService, supports future and/or periodic

execution of tasks.

Typically, variables that refer to executor objects are declared as one

of these three interface types, not with an executor class type.

Executor may be a simple interface, but it forms the basis for a flexible and powerful framework

for asynchronous task execution that supports a wide variety of task execution policies.

It provides a standard means of decoupling task submission from task execution,

describing tasks with Runnable.

The Executor implementations also provide lifecycle support and hooks for adding statistics gathering,

application management, and monitoring.

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads.

Worker threads are the thread objects which exists separately from the Runnable and Callable tasks

they execute. Means worker thread is not tightly bound with any particular task(i.e Runnable or Callable)

but they are often used to execute multiple tasks.

Executor creates thread pool with fixed number of Worker Threads and whenever thread is required, any

available worker thread is assigned the task (Runnable or Callable) which is executed by the worker thread

and when the task finishes this worker thread is returned back to the thread pool. And then it can be

again reassigned to another task. This means a worker thread is not tightly bound with specific Runnable task.

For working with executors we have to use java.util.concurrent.Executors class. As this class works like a factory of different type of Executor objects. Because it has several important methods for getting

executors that work on fixed size thread pools, single threads, caching thread pools, and methods to produce the same implementations in the scheduled format. In addition, the executors factory has methods to convert Runnable objects into Callable objects that return null.

For more information about this class refer to ,

http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executors.html

newFixedThreadPool

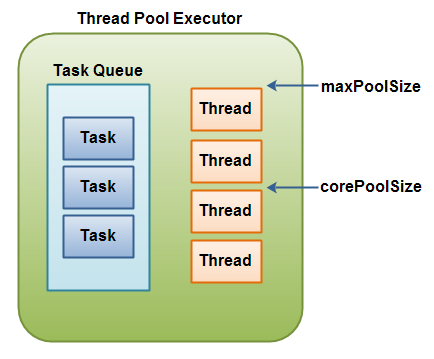
Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

At any point, at most nThreads threads will be active processing tasks.

If additional tasks are submitted when all threads are active, they will wait in the queue

until a thread is available. If any thread terminates due to a failure during execution prior

to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly shutdown.



newCachedThreadPool

Creates a thread pool that creates new threads as needed, but will reuse previously constructed

threads when they are available. These pools will typically improve the performance of programs

that execute many short-lived asynchronous tasks. Calls to execute will reuse previously

constructed threads if available. If no existing thread is available, a new thread will be created

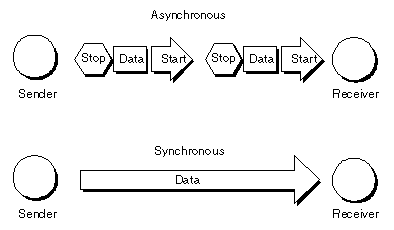
and added to the pool.

Threads that have not been used for sixty seconds are terminated and removed from the cache.

Thus, a pool that remains idle for long enough will not consume any resources.

In terms of resources, the newFixedThreadPool will keep all the threads running until they are explicitly terminated. In the newCachedThreadPool Threads that have not been used for sixty seconds are terminated and removed from the cache.

**Callables and Futures**



Runnable does not return any result.

In case we expect threads to return a computed result then we can use java.util.concurrent.Callable.

The Callable object allows to return values after completion. Java 5 introduced

java.util.concurrent.Callable interface in concurrency package which is similar to Runnable interface but it can return any Object and able to throw Exception. The Callable object uses generics to define the type of object which is returned.

Javadoc for Callable interface is,

public interface Callable<V>

A task that returns a result and may throw an exception. Implementors define a single method with no arguments called call.

The Callable interface is similar to [Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html), 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](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html) class contains utility methods to convert from other common forms to Callable classes.

### Method Detail from javadoc.

#### call

* [V](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html) call()

throws [Exception](http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html)

Computes a result, or throws an exception if unable to do so.

**Returns:**

computed result

**Throws:**

[Exception](http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html) - if unable to compute a result

When we submit a Callable object to an Executor the framework (i.e. Executor is known as framework) returns an object of type java.util.concurrent.Future. This Future object can be used to check the status of a Callable and to retrieve the result from the Callable.

Executors class provide useful methods to execute Callable in a thread pool.

Since callable tasks run in parallel, we have to wait for the returned Object. Callable tasks return java.util.concurrent.Future object.

Using Future we can find out the status of the Callable task and get the returned Object. It provides get() method that can wait for the Callable to finish and then return the result.

A Future represents the result of an asynchronous computation.

Actions taken by the asynchronous computation represented by a Future are happen-before actions subsequent to the retrieval of the result via Future.get() in another thread.

Future provides cancel() method to cancel the associated Callable task.

There is an overloaded version of get() method where we can specify the time to wait for the result,

it’s useful to avoid current thread getting blocked for longer time.

There are isDone() and isCancelled() methods to find out the current status of associated Callable task.

The FutureTask class is an implementation of Future that implements Runnable, and so may be executed by an Executor.

A cancellable asynchronous computation. This class provides a base implementation of Future, with methods to start and cancel a computation, query to see if the computation is complete, and retrieve the result of the computation. The result can only be retrieved when the computation has completed; the get method will block if the computation has not yet completed. Once the computation has completed, the computation cannot be restarted or cancelled.

FutureTask can be used to wrap a [Callable](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html) or [Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) object. Because FutureTask implements Runnable, a FutureTask can be submitted to an [Executor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) for execution. In addition to serving as a standalone class, this class provides protected functionality that may be useful when creating customized task classes. This class is useful when we want to override some of Future interface methods and don’t want to implement every method of Future interface.

ExecutorService shutdown.

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

For instance, if your application is started via a main() method and your main thread exits

your application, the application will keep running if you have an active ExexutorService in your

application. The active threads inside this ExecutorService prevents the JVM from shutting down.

To terminate the threads inside the ExecutorService you call its shutdown() method.

The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down.

All tasks submitted to the ExecutorService before shutdown() is called, are executed.

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method.

This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks.

There are no guarantees given about the executing tasks.

Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

**ScheduledExecutorService**

------------------------

This is An ExecutorService that can schedule commands to run after a given delay, or to execute periodically. This is an interface which extends ExecutorService interface.

So this interface supplements the methods of its parent ExecutorService with schedule, ScheduledExecutorService 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.

Concrete implementation of this interface is , class ScheduledThreadPoolExecutor.

How you create an ScheduledExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ScheduledExecutorService instances too.

Once you have created a ScheduledExecutorService you use it by calling one of its methods:

- schedule (Callable task, long delay, TimeUnit timeunit)

- schedule (Runnable task, long delay, TimeUnit timeunit)

- scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

- scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

Just like an ExecutorService, the ScheduledExecutorService needs to be shut down when you are finished using it. If not, it will keep the JVM running, even when all other threads have been shut down.

You shut down a ScheduledExecutorService using the shutdown() or shutdownNow() methods which are inherited from the ExecutorService interface.

In addition to the basic executor service, scheduled executors provide methods for producing repeating invocations of runnables and callables - as many people who have written asynchronous programs know, this type of class is the heart of any job execution object. Note that scheduled executor service methods return special ScheduledFuture objects which are customized to handle the delayed/repeated nature of this type of execution. Periodic application file system synchronization, file backups, application update checks, or even GUI updates are all good examples of repeated runnables this class could manage.