Executor Framework

Executor Framework (a part of Java SE)

Introduced in Java 5.

What's earlier support ?

Extends Thread
Implements Runnable

Why Executor Framework?

If you have thousands of task to be executed and if you create each thread for thousands of tasks, you will get performance overheads as creation and maintenance of each thread is an overhead.

Executor framework solves this problem.

In executor framework, you can create specified number of threads and reuse them to execute more tasks once it completes its current task.

It simplifies the design of creating multithreaded application and manages thread life cycles.

The programmer does not have to create or manage threads themselves, that’s the biggest advantage of executor framework.

Important classes / interfaces for executor framework.

1. java.util.concurrent.Executor
This interface is used to submit new task.
It has a method called “execute”.

```
public interface Executor {
 void execute(Runnable task);
}
```

2. ExecutorService
It is sub-interface of Executor.
Provides methods for
Submitting / executing Callable/Runnable tasks
Shutting down service
Executing multiple tasks etc.

3. ScheduledExecutorService
It is sub-interface of executor service which provides methods for scheduling tasks at fixed intervals or with initial delay.

4. Executors
This class provides factory methods for creating thread pool based executors.

Important factory methods(=public static method rets instance of ExecutorService) of Executors are:

4.1 newFixedThreadPool: This method returns thread pool executor whose maximum size is fixed.If all n threads are busy performing the task and additional tasks are submitted, then they will have to wait in the queue until thread is available.
4.2
newCachedThreadPool: this method returns an unbounded thread pool. It doesn’t have maximum size but if it has less number of tasks, then it will tear down unused thread. If a thread has been unused for keepAliveTime , then it will tear it down.
4.3 newSingleThreadedExecutor: this method returns an executor which is guaranteed to use the single thread.
4.4 newScheduledThreadPool: this method returns a fixed size thread pool that can schedule commands to run after a given delay, or to execute periodically.

Steps for Runnable
1. Create a thread-pool executor , using suitable factory method of Executors.

eg : For fixed no of threads
ExecutorService executor = Executors.newFixedThreadPool(10);

2. Create Runnable task

3. Use inherited method
public void execute(Runnable command)
Executes this Runnable task , in a separate thread.

4. Shutdown the service
public void shutdown()
Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

5. boolean awaitTermination(long timeout,TimeUnit unit)
                  throws InterruptedException
Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs.

6.
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.

BUT disadvantages with Runnable interface
1. Can't return result from the running task
2. Doesn't include throws Exception .

Better API
java.util.concurrent.Callable<V>

V : result type of call method
Represents a task that returns a result and may throw an exception.
Functional i/f
SAM :
public V call() throws Exception
Computes a result, or throws an exception if unable to do so.

Steps in using Callable i/f
1. Create a thread-pool executor , using suitable factory method of Executors.

eg : For fixed no of threads
ExecutorService executor = Executors.newFixedThreadPool(10);

2. Create Callable task , which returns a result.

3. To submit a task to executor service , use method of ExecutorService i/f :
public Future<T> submit(Callable<T> task)
Submits a value-returning task for execution and returns a Future representing the pending results of the task. It's a non blocking method (i.e rets immediately)

The Future's get method will return the task's result upon successful completion.

If you would like to immediately block waiting for a task, invoke get() on Future.
eg : result = exec.submit(aCallable).get();

OR
main thread can perform some other jobs in the mean time & then invoke get on Future , to actually get the results. (get : blocking call ,waits till the computation is completed n then rets result)

4. Other methods of ExecutorService i/f

public List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException

It's a blocking call.(waits till all tasks are complete)
Executes the given tasks, returning a list of Futures holding their status and results when all complete. Future.isDone() is true for each element of the returned list.

5. Shutdown the service
public void shutdown()
Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

6. boolean awaitTermination(long timeout,TimeUnit unit)
                  throws InterruptedException
Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs.

7.
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.