**Executor Framework and Methods – Quick Reference - 2022**

**Executors**

**java.util.concurrent.Executors.newFixedThreadPool(int) 🡸 For long running tasks**

**java.util.concurrent.Executors.newSingleThreadExecutor() 🡸 For Sequential task**

**java.util.concurrent.Executors.newCachedThreadPool() 🡸 For Short Lived Tasks**

**public static ExecutorService newCachedThreadPool() 🡸 For Short Lived Tasks**

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

**public static ExecutorService newFixedThreadPool(int nThreads) 🡸 For long running tasks**

**public static ExecutorService newSingleThreadExecutor() 🡸 For Sequential task**

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

**Methods in ExecutorService**

**execute(), submit(), invokeAny(), invokeAll()**

**OtherUsage: Callable, Future, CompletionService**

**void execute(Runnable runnable)**

**public void** m1(String name, **int** time) {  
 **try** {  
 System.***out***.println(name + **" Validation Executing ..."**);  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 System.***out***.println(name + **" Validation completed ..."**);  
}

**public void** check() {  
 Runnable r1 = () -> m1(**"Aadhar"**, 7);  
 Runnable r2 = () -> m1(**"Pan"**, 7);  
 Runnable r3 = () -> m1(**"Passport"**, 3);  
  
 **ExecutorService exService = Executors.*newFixedThreadPool*(3);  
 exService.execute(r1);  
 exService.execute(r2);  
 exService.execute(r3);**  
  
 exService.shutdown(); *// <== This is not a blocking method ...* System.***out***.println(**"Initiated shutdown ..."**);  
  
 **while** (!exService.isTerminated()) {  
 } *// <== To Make it blocking* System.***out***.println(**"All threads completed ..."**);  
}

**The main difference between submit() and execute() is that submit returns a Future object and from futute object , you get the result using future.get() whereas execute() does not**

**return**

**FutureTask**

A cancellable asynchronous computation. The get methods will block if the computation has not yet completed. Once the computation has completed, the computation cannot be restarted or cancelled (unless the computation is invoked using [runAndReset()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html" \l "runAndReset())). **A FutureTask can be used to wrap a Callable or Runnable object**. Because FutureTask implements Runnable, a FutureTask can be submitted to an Executor for execution. In implementation, FutureTask accepts only callable not Runnable.

**Execute with FutureTask 🡺 FutureTask can be used only with Callable not with Runnable**

**public** String m2(String name, **int** time) {  
 **try** {  
 System.***out***.println(name + **" Validation Executing ..."**);  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 **return** name + **" Validation completed ..."**;  
}

**public void** check1() {  
 //**Callable<String> c1 = () -> m2("Aadhar", 7);  
 //Callable<String> c2 = () -> m2("Pan", 7);**  
 **FutureTask<String> ft1 = new FutureTask<>(() -> m2("Aadhar", 7)); 🡸 No Runnable  
 FutureTask<String> ft2 = new FutureTask<>(() -> m2("Pan", 7)); 🡸 No Runnable**

**ExecutorService exService = Executors.*newFixedThreadPool*(2);  
 exService.execute(ft1);  
 exService.execute(ft2);**  
  
 exService.shutdown();  
 System.***out***.println(**"All threads completed ..."**);  
  
 **try** {  
 **String response1 = ft1.get();** System.***out***.println(**"Response1: "**+response1);  
 **String response2 = ft2.get();** System.***out***.println(**"Response2: "**+response2);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 } **catch** (ExecutionException e) {  
 **throw new** RuntimeException(e);  
 }  
}

**Usage of submit()**

**Future<?> submit(Runnable task)**

**<T> Future<T> submit(Callable<T> task)**

**<T> Future<T> submit(Runnable task, T result)**

**If you use submit(), the result will be obtained in the order of submission.**

**submit()** 🡺 **Provides the response in the order of submission.**

**invokeAll()** 🡺 **Provides the response in the order of submission.**

**Future<?> submit(Runnable task)**

**public void** m1(String name, **int** time) {  
 **try** {  
 System.***out***.println(name + **" Validation Executing ..."**);  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 System.***out***.println(name + **" Validation completed ..."**);  
}

**public void** check() {  
 Runnable r1 = () -> m1(**"Aadhar"**, 7);  
 Runnable r2 = () -> m1(**"Pan"**, 7);  
 **Runnable r3 = () -> m1("Passport", 3);**  
 **ExecutorService exService = Executors.*newFixedThreadPool*(3);**  
  
 **Future<Void> ft1 = (Future<Void>) exService.submit(r1);  
 Future<Void> ft2 = (Future<Void>) exService.submit(r2);  
 Future<Void> ft3 = (Future<Void>) exService.submit(r3);**  
  
 exService.shutdown(); *// <== This is not a blocking method ...* System.***out***.println(**"Initiated shutdown ..."**);  
  
 **while (!exService.isTerminated()) { }** *// <== To Make it blocking* System.***out***.println(**"All threads completed ..."**);  
}

**<T> Future<T> submit(Callable<T> task)**

**public void** check1() {  
 Callable<String> c1 = () -> m2(**"Aadhar"**, 7);  
 Callable<String> c2 = () -> m2(**"Pan"**, 7);  
  
 ExecutorService exService = Executors.*newFixedThreadPool*(2);  
  
 **Future<String> ft1 = exService.submit(c1);  
 Future<String> ft2 = exService.submit(c2);**  
 exService.shutdown();  
 System.***out***.println(**"All threads completed ..."**);  
  
 **try** {  
 String response1 = ft1.get();  
 System.***out***.println(**"Response1: "**+response1);  
 String response2 = ft2.get();  
 System.***out***.println(**"Response2: "**+response2);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 } **catch** (ExecutionException e) {  
 **throw new** RuntimeException(e);  
 }  
}

**🡺 Or it can be written like below** 🡸

List<Future> futureList = Arrays.*asList*(ft1, ft2);  
**try** {  
 **for** (Future<String> future : futureList) {  
 String response = future.get();  
 System.***out***.println(**"Response : "** + response);  
 }  
} **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
} **catch** (ExecutionException e) {  
 **throw new** RuntimeException(e);  
}

**<T>**[**List**](https://docs.oracle.com/javase/7/docs/api/java/util/List.html)**<**[**Future**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html)**<T>> invokeAll(**[**Collection**](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)**<? extends** [**Callable**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html)**<T>> tasks) throws** [**InterruptedException**](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)

**Executes the given tasks, returning a list of Futures holding their status and results when all complete.**

**public class** InvokeAll1 {  
  
 **public** String task1(String name, **int** time) {  
 System.***out***.println(name+**" executing task ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 System.***out***.println(name+**" completed task ..."**);  
 } **catch**(InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 **return** name;  
 }   
  
 **public void** test() {  
 Callable<String> c1 = () -> task1(**"John"**, 7);  
 Callable<String> c2 = () -> task1(**"John"**, 5);  
 Callable<String> c3 = () -> task1(**"John"**, 3);  
  
  
 **ExecutorService exService = Executors.*newFixedThreadPool*(3);**  
 List<Callable<String>> callableList = **new** ArrayList<>();  
  
 **callableList.add(c1);  
 callableList.add(c2);  
 callableList.add(c3);** **try** {  
 **List<Future<String>> futureList = exService.invokeAll(callableList);** **for**(Future<String> future : futureList) {  
 **String response = future.get();** System.***out***.println(**"Response : "**+response);  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }  
 exService.shutdown();  
 }  
  
 **public static void** main(String[] args) {  
 **new** InvokeAll1().test();  
 }  
}

**invokeAll()** 🡺 **Provides the response in the order of submission.**

**submit()** 🡺 **Provides the response in the order of submission.**

**OUTPUT**

John executing task ...

Hari executing task ...

Vidya executing task ...

Hari completed task ...

Vidya completed task ...

John completed task ...

Response : John

Response : Vidya

Response : Hari

**\*\* While using invokeAll(), return types should be of similar nature**

**<T> T invokeAny(**[**Collection**](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)**<? extends** [**Callable**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html)**<T>> tasks)**

**throws** [**InterruptedException**](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)**,** [**ExecutionException**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutionException.html)

**Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception). It returns which task has completed first.**

public String task(String name, int time) {  
 System.*out*.println(name+" executing task ...");  
 try {  
 TimeUnit.*SECONDS*.sleep(time);  
 System.*out*.println(name+" completed task ...");  
 } catch(InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 return name;  
 }  
  
 public void test() {  
 ExecutorService exService = Executors.*newFixedThreadPool*(2);  
 **Callable<String> c1 = () -> task("John", 7);  
 Callable<String> c2 = () -> task("Hati", 5);**  
  
 List<Callable<String>> callableList = new ArrayList<>();  
  
 **callableList.add(c1);  
 callableList.add(c2);**  
 try {  
 **String response = exService.invokeAny(callableList);** System.*out*.println("Response : "+response);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
 exService.shutdown();  
 }

**OUTPUT**

Hati executing task ...

John executing task ...

Hati completed task ...

Response : Hati

java.lang.InterruptedException: sleep interrupted

at java.base/java.lang.Thread.sleep(Native Method)

at java.base/java.lang.Thread.sleep(Thread.java:334)

at java.base/java.util.concurrent.TimeUnit.sleep(TimeUnit.java:446)

at com.ddlab.rnd.type1.InvokeAny.task(InvokeAny.java:12)

If Thread.sleep() or TimeUnit.SECONDS.sleep() is there, it always throws InterruptedException. You need to handle the exception.

**CompletionService 🡺 ExecutorCompletionService<V> (Order of Completion)**

A service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks. Producers submit tasks for execution. Consumers take completed tasks and process their results in the order they complete. **The response will come in the order of completion.**

* **ExecutorService = incoming queue + worker threads**
* **CompletionService = incoming queue + worker threads + output queue**

**public** String task(String name, **int** time) {  
 System.***out***.println(name + **" executing task ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 System.***out***.println(name + **" completed task ..."**);  
 } **catch** (InterruptedException ie) { ie.printStackTrace(); }  
 **return** name;  
 }  
  
 **public void** test() {  
 Callable<String> c1 = () -> task(**"John"**, 9);  
 Callable<String> c2 = () -> task(**"Vidya"**, 7);  
 Callable<String> c3 = () -> task(**"Hari"**, 5);  
  
 ExecutorService exService = Executors.*newFixedThreadPool*(3);  
 **CompletionService completionService = new ExecutorCompletionService(exService);**  
  
 **Future<String> future1 = completionService.submit(c1);  
 Future<String> future2 = completionService.submit(c2);  
 Future<String> future3 = completionService.submit(c3);**  
 **int** noOfFuture = 3;  
 **for** (**int** i = 0; i < noOfFuture; i++) {  
 **try** {  
 **Future<String> future = completionService.take();  
 String response = future.get();  
 System.*out*.println("Response : " + response);** } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
 exService.shutdown();  
 }

OUTPUT

John executing task ...

Hari executing task ...

Vidya executing task ...

Hari completed task ...

Response : Hari

Vidya completed task ...

Response : Vidya

John completed task ...

Response : John

**Note:** **If want in the order of completion, use CompletionService, if want in the order of submission, use normal executorService.submit.**

**take() is a blocking API**

**poll() is a non-blocking**

**Note : shutdown() method is used graceful termination whereas shutdownNow() is used for abrupt abnormal termination. Better way to use ExecutorService.shutdownNow(). For example, if your not getting response for a period of time, then use the above method.**

**try** {  
 **if** (!exService.awaitTermination(800, TimeUnit.***MILLISECONDS***)) {  
 exService.shutdownNow();  
 }  
} **catch** (InterruptedException e) {  
 exService.shutdownNow();  
}

**Example on awaitTermination(long, TimeUnit)**

**public class** TestThreads {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** ping(String url) {  
 sleep(10);  
 }  
  
 **public void** check() {  
 Instant startTime = Instant.*now*();  
 ExecutorService exService = Executors.*newCachedThreadPool*();  
 Thread t1 = **new** Thread( () -> ping("www.google.com") );  
 Thread t2 = **new** Thread( () -> ping("www.yahoo.com") );  
  
 exService.execute(t1);  
 exService.execute(t2);  
  
 exService.shutdown();  
 **try** {  
 **if** (!exService.awaitTermination(3, TimeUnit.***SECONDS***)) {  
 exService.shutdownNow();  
 }  
 } **catch** (InterruptedException e) {  
 exService.shutdownNow();  
 }  
 System.***out***.println("All the threads terminated ...");  
 Instant endTime = Instant.*now*();  
 **long** totalTime = Duration.*between*(startTime, endTime).getSeconds();  
 System.***out***.println("Total Time Taken in sec: "+totalTime);  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestThreads().check();  
 }  
}

**Difference between Executor, ExecutorService and Executors class in Java**

* **Executor is the core interface** which is an abstraction for parallel execution.
* On the other hand, **ExecutorService is an extension of Executor interface** and provides a facility for returning a Future object and terminate, or shut down the thread pool.
* **Executors is a utility class similar to Collections, which provides**[**factory methods**](http://javarevisited.blogspot.com/2015/06/difference-between-dependency-injection.html)**to create different types of thread pools e.g. fixed and cached thread pools**.

**Synchronized Collections vs Concurrent Collections in Java**

**Main reason for this slowness is locking; synchronized collections locks the whole collection** e.g. whole Map or List **while concurrent collection never locks the whole Map or List**. They achieve thread safety by using advanced and sophisticated techniques like lock stripping. For example, the **ConcurrentHashMap divides the whole map into several segments and locks only the relevant segments, which allows multiple threads to access other segments of same ConcurrentHashMap without locking**.  
  
**Similarly, CopyOnWriteArrayList allows multiple reader threads to read without synchronization and when a write happens it copies the whole ArrayList and swap with a newer one.**