Java Concurrency – 2021-2022

**Executors (According to Brian Goetz)**

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

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

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

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

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. **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**

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

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

Creates an Executor that uses a single worker thread operating off an unbounded queue, much like the Swing event thread. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.

Given this, the resource consumption will depend very much in the situation. For instance, **If you have a huge number of long running tasks I would suggest the FixedThreadPool**. **As for the CachedThreadPool, the docs say that "These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks"**.

In case of large long running tasks use **Executors.newFixedThreadPool(10)**. It is useful in case of JMS where you are getting more request to process.

In case of short lived tasks use Executors.newCachedThreadPool();

**What about Executors.newSingleThreadExecutor() ? It will execute sequentially.**

**Using Executors.*newFixedThreadPool*()**

**import** java.time.Duration;  
**import** java.time.Instant;  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** TestThreads {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** m1() {  
 **for** (**int** i = 0; i < 5; i++) {  
 System.***out***.println(Thread.*currentThread*().getName() + " running ..." + i);  
 sleep(3);  
 }  
 }

**public void** check() {  
 Instant startTime = Instant.*now*();  
 **ExecutorService exService = Executors.*newFixedThreadPool*(3);**  
 **Thread t1 = new Thread( () -> m1() );  
 Thread t2 = new Thread( () -> m1() );  
 Thread t3 = new Thread( () -> m1() );**  
 exService.execute(t1);  
 exService.execute(t2);  
 exService.execute(t3);  
  
 **exService.shutdown();  
 while (!exService.isTerminated()) {}** System.***out***.println("All threads completed ...");  
 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();  
 }  
}

Output

Total Time taken in sec: 15

**Using Executors.newSingleThreadExecutor()**

**public void** check() {  
 Instant startTime = Instant.*now*();  
 ExecutorService exService = Executors.*newSingleThreadExecutor*();  
 Thread t1 = **new** Thread( () -> m1() );  
 Thread t2 = **new** Thread( () -> m1() );  
 Thread t3 = **new** Thread( () -> m1() );  
  
 exService.execute(t1);  
 exService.execute(t2);  
 exService.execute(t3);  
  
 exService.shutdown();  
 **while** (!exService.isTerminated()) {}  
 System.***out***.println("All threads completed ...");  
 Instant endTime = Instant.*now*();  
 **long** totalTime = Duration.*between*(startTime, endTime).getSeconds();  
 System.***out***.println("Total Time Taken in sec: "+totalTime);  
}

Output

All threads completed ...

Total Time Taken in sec: 45

**Using Executors.newCachedThreadPool()**

**public void** check() {  
 Instant startTime = Instant.*now*();  
 **ExecutorService exService = Executors.*newCachedThreadPool*();**  
 Thread t1 = **new** Thread( () -> m1() );  
 Thread t2 = **new** Thread( () -> m1() );  
 Thread t3 = **new** Thread( () -> m1() );  
  
 exService.execute(t1);  
 exService.execute(t2);  
 exService.execute(t3);  
  
 exService.shutdown();  
 **while** (!exService.isTerminated()) {}  
 System.***out***.println("All threads completed ...");  
 Instant endTime = Instant.*now*();  
 **long** totalTime = Duration.*between*(startTime, endTime).getSeconds();  
 System.***out***.println("Total Time Taken in sec: "+totalTime);  
}

Output

All threads completed ...

Total Time Taken in sec: 15

**Note: In case ExecutorService.execute() method you can pass the Thread which implements Runnable and also Thread which extends Thread.**

**exservice1.shutdown()** 🡺 **Not a Blocking method**

**ExecutorService**

**java.util.concurrent.ExecutorService.shutdown()**

**java.util.concurrent.ExecutorService.shutdownNow()**

There are two methods in the executor service. Whenever you are using Executors for multi processing of tasks, you normally use shutdown() method.

What about shutdownNow() method ? According JLS : Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via Thread.interrupt, so any task that fails to respond to interrupts may never terminate. Let us consider a small example. We will check if the list size is greater than 7, it should shutdown immediately.

**public class** TestThreads {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** m1(List<String> list) {  
 **for** (**int** i = 0; i < 10; i++) {  
 list.add(Thread.*currentThread*().getName()+i);  
 System.***out***.println(Thread.*currentThread*().getName() + " running ..." + i);  
 }  
 sleep(1);  
 }  
  
 **public void** check() {  
 Instant startTime = Instant.*now*();  
 List<String> list = **new** CopyOnWriteArrayList<>();  
 ExecutorService exService = Executors.*newCachedThreadPool*();  
 Thread t1 = **new** Thread( () -> m1(list) );  
 Thread t2 = **new** Thread( () -> m1(list) );  
  
 exService.execute(t1);  
 exService.execute(t2);  
  
 exService.shutdown();  
 **while**( !exService.isTerminated() ) {  
 **if**( list.size() > 7 )  
 exService.shutdownNow();  
 }  
 System.***out***.println("My Shared List :::"+list);  
 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();  
 }  
}

The output will be as per the following.

My Shared List :::[pool-1-thread-10, pool-1-thread-20, pool-1-thread-11, pool-1-thread-21, pool-1-thread-12, pool-1-thread-22, pool-1-thread-13, pool-1-thread-23, pool-1-thread-14, pool-1-thread-24, pool-1-thread-15, pool-1-thread-25, pool-1-thread-16, pool-1-thread-26, pool-1-thread-17, pool-1-thread-27, pool-1-thread-18, pool-1-thread-28, pool-1-thread-19, pool-1-thread-29]

All the threads terminated ...

Total Time Taken in sec: 0

Exception in thread "pool-1-thread-2" java.lang.RuntimeException: java.lang.InterruptedException: sleep interrupted

at com.ddlab.rnd.TestThreads.sleep(TestThreads.java:18)

at com.ddlab.rnd.TestThreads.m1(TestThreads.java:27)

at com.ddlab.rnd.TestThreads.lambda$check$1(TestThreads.java:37)

at java.base/java.lang.Thread.run(Thread.java:833)

at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1136)

at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:635)

at java.base/java.lang.Thread.run(Thread.java:833)

Caused by: java.lang.InterruptedException: sleep interrupted

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

at java.base/java.lang.Thread.sleep(Thread.java:337)

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

In the above example we have seen how we have used **shutdownNow()** method. It is nothing but abrupt termination of thread. You can see the output that the list contains 9 elements and thread 1 completed the task whereas thread2 did not complete properly. But our intention was that the list should not contain more than 7 elements, however it contains 9 elements. What is wrong with the above program. We have also used ThreadLocal.

If you want to avoid the exception you can add the following catch statement in both the threads.

**catch( InterruptedException ie ) {**

**if( Thread.currentThread().isInterrupted() )**

**Thread.currentThread().interrupt();**

**}**

Now let use have a better program which can satisfy our requirements. Although we have used ThreadLocal, still we are unable to achieve thread-safety.

**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();  
}

In case of Java 8, use ThreadLocal in the following manner.

**private static final ThreadLocal<List<Object>> list = ThreadLocal.withInitial(ArrayList::new);**

What happened to our condition, now condition is skipped.

**Note : Always remember that ThreadLocal creates a local copy for each thread.** This is baseline.

However thread-safety can not be achieved using ThreadLocal for all kinds of Objects, best example is ArrayList.

**You have to use CopyOnWriteArrayList.**

**According to JLS**

**boolean isShutdown() : Returns true if this executor has been shut down.**

**boolean isTerminated() : Returns true if all tasks have completed following shut down. Note that isTerminated is never true unless either shutdown or shutdownNow was called first.**

**boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException**

**Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.**

Make sure that before calling all the above you have to call shutdown() or shutdownNow() method.

**Let us consider a small example on awaitTermination(). Think about the situation where we want to connect to multiple sites and we want to ping the sites. If we do not get response within 5 seconds we have to shutdown the execution.** Let us see the example.

**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();  
 }  
}

Here, you can see that even if we have given shutdownNow() method, still all the threads run. It means there is no interruption policy defined properly. If you look into the code of thread, you will find that there is sleep() method which resets the interruption flag. So it is now always true that simply providing shutdownNow() will stop all the threads. You have to design your own interruption policy.

Now the modified code for the thread class is given below.

**import** java.util.concurrent.TimeUnit;  
**public class** TaskThread **implements** Runnable {  
 **private** String **name**;  
 **private int time**;  
  
 **public** TaskThread(String name, **int** time) {  
 **this**.**name** = name;  
 **this**.**time** = time;  
 Thread.*currentThread*().setName(name);  
 }  
  
 @Override  
 **public void** run() {  
 **for** (**int** i = 0; i < 5; i++) {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" running ..."** + i);  
 TimeUnit.***SECONDS***.sleep(**time**);  
 *//Interruption/Termination policy designed below***if (Thread.*currentThread*().isInterrupted()) {  
 Thread.*currentThread*().interrupt();  
 }** } **catch** (Exception e) {  
 e.printStackTrace();  
 *// Interruption/Termination policy designed below* **Thread.*currentThread*().interrupt();** System.***out***.println(Thread.*currentThread*().getName() + **" interrupted "**);  
 }  
 }  
 }  
}

**java.util.concurrent.ExecutorService.submit(Callable<T>)**

**java.util.concurrent.ExecutorService.submit(Runnable, T)**

**java.util.concurrent.ExecutorService.submit(Runnable)**

**<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. 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, you can use constructions of the

**form result =** **exec.submit(aCallable).get();**

Note: The Executors class includes a set of methods that can convert some other common closure-like objects, for example, PrivilegedAction to Callable form so they can be submitted.

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

Submits a Runnable task for execution and returns a Future representing that task. The Future's get method will return the given result upon successful completion.

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

**Submits a Runnable task for execution and returns a Future representing that task. The Future's get method will return null upon successful completion**. Let us consider a following example.

**public class** TestThreads {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public** String m1() {  
 sleep(10);  
 **return** "someValue";  
 }  
  
 **public void** check() {  
 Instant startTime = Instant.*now*();  
 ExecutorService exService = Executors.*newCachedThreadPool*();  
 **Callable<String> callable1 = () -> m1();**  
 **Future<String> future = exService.submit(callable1);**  
 **try** {  
 String value = future.get();  
 System.***out***.println("Value: " + value);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 } **catch** (ExecutionException e) {  
 **throw new** RuntimeException(e);  
 }  
 **exService.shutdown();**  
  
 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();  
 }  
}

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

<https://stackoverflow.com/questions/6456504/what-does-result-in-executorservice-submitrunnable-task-t-result-do>

Runnable does not return anything and Future must return something so this method allows you to predefine the result of the returned future. If you don't want to return a thing you can return null and I thing the Void type exists to express that kind of things.

**Future<Void> myFuture = executor.submit(myTask, null);**

**execute()**

ExecutorService.execute() method can execute a thread which extends a Thread class or implements a Runnable interface. Code is given below.

**public class** TestThreads {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public** String m1() {  
 sleep(10);  
 **return** "someValue";  
 }  
  
 **public void** m2() {  
 sleep(5);  
 }  
  
 **public void** check() {  
 Instant startTime = Instant.*now*();  
 ExecutorService exService = Executors.*newFixedThreadPool*(2);  
  
 **exService.execute( () -> m1());  
 exService.execute( () -> m2() );** exService.shutdown();  
  
 **while** (!exService.isTerminated()) {} // ⎜A blocker  
 System.***out***.println("All threads completed ...");  
  
 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();  
 }  
}

**ExecutorService.invokeAll(Collection<? extends Callable<T>>)**

**ExecutorService.invokeAll(Collection<? extends Callable<T>>, long, TimeUnit)**

**ExecutorService.invokeAny(Collection<? extends Callable<T>>)**

**ExecutorService.invokeAny(Collection<? extends Callable<T>>, long, TimeUnit)**

**<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)**

**throws InterruptedException**

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. Note that a completed task could have terminated either normally or by throwing an exception. The results of this method are undefined if the given collection is modified while this operation is in progress.

**<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,**

**long timeout, TimeUnit unit) throws InterruptedException**

**Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first**. **Future.isDone()** is true for each element of the returned list. Upon return, tasks that have not completed are cancelled. Note that a completed task could have terminated either normally or by throwing an exception. The results of this method are undefined if the given collection is modified while this operation is in progress. Invoking this method does not return a Future, but returns the result of one of the Callable objects.

**<T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException,**

**ExecutionException**

**Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do. Upon normal or exceptional return, tasks that have not completed are cancelled**. The results of this method are undefined if the given collection is modified while this operation is in progress. Invoking this method does not return a Future, but returns the result of one of the Callable objects. It is equivalent **CompletableFuture.anyOf()**.

**<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)**

**throws InterruptedException, ExecutionException, TimeoutException**

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. Upon normal or exceptional return, tasks that have not completed are cancelled. The results of this method are undefined if the given collection is modified while this operation is in progress. Invoking this method does not return a Future, but returns the result of one of the Callable objects.

**ExecutorCompletionService**

**http://stackoverflow.com/questions/7758020/difference-between-executor-and-executorcompletionservice-in-java**

Suppose you had a set of tasks A, B, C, D, E and you want to execute each of them asynchronously in an Executor and process the results 1 by 1 as they complete. **With an Executor, you would do so like this**:

**List<Future<?>> futures = new ArrayList<Future<?>>();**

**futures.add(executorService.submit(A));**

**futures.add(executorService.submit(B));**

**futures.add(executorService.submit(C));**

**futures.add(executorService.submit(D));**

**futures.add(executorService.submit(E));**

The implementation of ExecutorCompletionService contains a queue of results. **If take or poll are not called to drain that queue, a memory leak will occur**. Some people use the Future returned by submit to process results and this is NOT correct usage.

**http://stackoverflow.com/questions/4912228/when-should-i-use-a-completionservice-over-an-executorservice**

With ExecutorService , once you have submitted the tasks to run , you need to manually code for efficiently getting the results of the tasks completed. With CompletionService , this is pretty much automated. The difference is not very evident in the code you have presented because you are submitting just one task. However , imagine you have a list of tasks to be submitted. In the example below, multiple tasks are submitted to the CompletionService. Then , instead of trying to find out which task has completed ( to get the results ) , it just asks the CompletionService instance to return the results as they become available

**CompletionService**

**If want in the order of completion, use CompletionService,**

**if want in the order of submission, use normal executorService.submit.**

**Question-1**. If you have 3 threads, all the 3 threads should run asynchronously but I want all the result in the **order of submission**. How will you do it?

The code is given below.

**public class** TestOrderOfSubmission {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public** String callableTask(String taskName, **long** time) {  
 sleep(time);  
 **return** taskName + " completed ...";  
 }  
  
 **public void** check() {  
 ExecutorService executorService = Executors.*newFixedThreadPool*(3);  
 Callable callable1 = () -> callableTask("A", 5);  
 Callable callable2 = () -> callableTask("B", 3);  
 Callable callable3 = () -> callableTask("C", 2);  
 List<Future<String>> futureList = **new** ArrayList<>();  
  
 futureList.add(executorService.submit(callable1));  
 futureList.add(executorService.submit(callable2));  
 futureList.add(executorService.submit(callable3));  
  
 futureList.forEach(  
 future -> {  
 **try** {  
 System.***out***.println("Result : " + future.get());  
 } **catch** (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 executorService.shutdown();  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestOrderOfSubmission().check();  
 }  
}

OUTPUT

Result : A completed ...

Result : B completed ...

Result : C completed ...

**Here all the results from the thread will come in the order you had submitted, it means it will come in a sequential manner.**

**Question-2**: If you have 3 threads, all the 3 threads should run asynchronously but I want all the result in the **order of completion**. How will you do it?

The above program can be written so that we can get the result in the order of submission which solves our second problem. The code is given below.

**public class** TestOrderOfCompletion {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public** String callableTask(String taskName, **long** time) {  
 sleep(time);  
 **return** taskName + " completed ...";  
 }  
  
 **public void** check() {  
 Callable callable1 = () -> callableTask("A", 5);  
 Callable callable2 = () -> callableTask("B", 3);  
 Callable callable3 = () -> callableTask("C", 2);  
  
 ExecutorService executorService = Executors.*newFixedThreadPool*(3);  
 CompletionService<String> completionService = **new** ExecutorCompletionService<>(executorService);  
  
 List<Future<String>> futureList = **new** ArrayList<>();  
  
 futureList.add(completionService.submit(callable1));  
 futureList.add(completionService.submit(callable2));  
 futureList.add(completionService.submit(callable3));  
  
 futureList.forEach(  
 a -> {  
 **try** {  
 Future<String> future = completionService.take();  
 System.***out***.println("Result : " + future.get());  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 executorService.shutdown();  
  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestOrderOfCompletion().check();  
 }  
}

**OUTPUT**

Result : C completed ...

Result : B completed ...

Result : A completed ...

**Let us take another example where 5 Indian missiles attack some cities of Pakistan, now we want to get the result which missile attack which cities in how much time. It means , basically we want the order of completion.** The code is given below.

**http://stackoverflow.com/questions/11872520/executorcompletionservice-why-do-need-one-if-we-have-invokeall**

If we use an ExecutorCompletionService we can submit a series of tasks as Callables and get the result interacting with the CompletionService as a queue. But there is also the invokeAll of ExecutorService that accepts a Collection of tasks and we get a list of Future to retrieve the results. As far as I can tell, there is no benefit in using one or over the other (except that we avoid a for loop using an invokeAll that we would have to submit the tasks to the CompletionService) and essentially they are the same idea with a slight difference.So why are there 2 different ways to submit a series of tasks? Am I correct that performance wise they are equivalent? Is there a case that one is more suitable than the other? I can't think of one.

**Ans : Using a ExecutorCompletionService.poll/take, you are receiving the Futures as they finish, in completion order** (more or less). **Using ExecutorService.invokeAll, you do not have this power; you either block until are all completed**, or you specify a timeout after which the incomplete are cancelled.

Note :

By using an ExecutorCompletionService, you can get immediately notified when each of your jobs completes. In comparison, ExecutorService.invokeAll(...) waits for all of your jobs to complete before returning the collection of Futures:

**FutureTask**

A basic example is given below.

**public class** TestThreads {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public** String m1() {  
 sleep(10);  
 **return** "someValue";  
 }  
  
 **public** String m2() {  
 sleep(5);  
 **return** "otherValue";  
 }  
  
 **public void** check() {  
 Instant startTime = Instant.*now*();  
 ExecutorService exService = Executors.*newFixedThreadPool*(2);  
  
 Callable callable1 = () -> m1();  
 Callable callable2 = () -> m2();  
  
 FutureTask<String> ft1 = **new** FutureTask<>(callable1);  
 FutureTask<String> ft2 = **new** FutureTask<>(callable2);  
  
 exService.execute(ft1);  
 exService.execute(ft2);  
 exService.shutdown();  
  
 **while** (!(ft1.isDone() && ft2.isDone())) {  
 System.***out***.println("Waiting for others to complete ...");  
 sleep(2);  
 }  
  
 System.***out***.println("All threads completed ...");  
  
 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();  
 }  
}

<http://programmingexamples.wikidot.com/futuretask>

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.

**Blocking Concept of Future.get() in java concurrency**

As we know that Future.get() will block, so question is how to use in a that we can perform some other operations. More specifically, if the thread takes some more time to perform some operation, it is not advisable to wait for some time rather we can perform some operations during that time. The code is given below.

**public class** TestThreads {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public** String m1() {  
 sleep(5);  
 **return** "someValue";  
 }  
  
 **public void** check() {  
 Instant startTime = Instant.*now*();  
 ExecutorService exService = Executors.*newFixedThreadPool*(1);  
  
 **Future<String> future = exService.submit(() -> m1());** exService.shutdown();  
  
 **while (!(future.isDone())) {  
 //wait here and do some other operation  
 System.*out*.println("I am doing something in the mean time ...");  
 sleep(2);  
 }** **try** {  
 String s = future.get();  
 System.***out***.println("Final Value --->" + s);  
 } **catch** (InterruptedException | ExecutionException e) {  
 **throw new** RuntimeException(e);  
 }  
 System.***out***.println("All threads completed ...");  
 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();  
 }  
}

Output

I am doing something in the mean time ...

I am doing something in the mean time ...

…..

Final Value --->someValue

**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.   
  
**The Future object provides the facility of asynchronous execution**, which means you don't need to wait until the execution finishes, you can just submit the task and go around, come back and check if Future object has the result, if execution is completed then it would have result which you can access by using the **Future.get() method. Just remember that this method is a**[**blocking method**](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html) i.e. it will wait until execution finish and the result is available if it's not finished already.  
  
**By using the**[**Future object**](http://javarevisited.blogspot.sg/2015/01/how-to-use-future-and-futuretask-in-Java.html)**returned by ExecutorService.submit() method, you can also cancel the execution** if you are not interested anymore. It provides cancel() method to cancel any pending execution.  
**Third one 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**.

**Executor vs ExecutorService vs Executors in Java**

1) One of the key difference between Executor and ExecutorService interface is that former is a parent interface while ExecutorService extends Executor i.e. it's a sub-interface of Executor.  
  
2) Another important difference between ExecutorService and Executor is that Executor defines execute() method which accepts an object of the Runnable interface, **while submit() method can accept objects of both**[**Runnable**](http://www.java67.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html)**and**[**Callable**](http://javarevisited.blogspot.com/2016/08/useful-difference-between-callable-and-Runnable-in-Java.html)**interfaces.**  
  
3) The third difference between Executor and **ExecutorService interface is that execute()** **method doesn't return any result**, **its return type is void but submit() method returns the result of computation via a**[**Future**](http://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.html)**object**. This is also the key difference between submit() and execute() method, which is one of the frequently asked [Java concurrency interview questions](http://javarevisited.blogspot.sg/2014/07/top-50-java-multithreading-interview-questions-answers.html).  
  
4) The fourth difference between ExecutorService and Executor interface is that apart from allowing a client to submit a task, ExecutorService also provides methods to control the thread pool e.g. terminate the thread pool by calling the shutDown() method.

5) Executors class provides [factory methods](http://javarevisited.blogspot.sg/2017/02/5-difference-between-constructor-and-factory-method-in-java.html) to create different kinds of thread pools e.g. newSingleThreadExecutor() creates a thread pool of just one thread, newFixedThreadPool(int numOfThreads) creates a thread pool of fixed number of threads and newCachedThreadPool() creates new threads when needed but reuse the existing threads if they are available.

**Thread vs Executor in Java**

1) First and foremost difference between Thread and Executor is that java.lang.Thread is a [class](http://www.java67.com/2016/08/difference-between-class-and-interface-in-java.html) in Java while java.util.concurrent.Executor is an [interface](http://www.java67.com/2014/02/what-is-actual-use-of-interface-in-java.html).  
2) The Executor concept is actually an *abstraction* over parallel computation. It allows concurrent code to be run in managed way. On the other hand, Thread is a *concrete* way to run the code in parallel.  
3) The third difference between an Executor and a Thread class is that former **decouples**a task (the code which needs to be executed in parallel) from execution, while in the case of a Thread, both task and execution are **tightly coupled**.   
4) The Executor concept allows your task is to be executed by a worker thread from the thread pool, while Thread itself execute your task.  
5) Executor provides a execute() method which accepts a Runnable task, while Thread accepts the [Runnable task](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) on its constructor.  
6) One more key difference between a Thread and an Executor is that a Thread can only execute **one Runnable task** but an Executor can execute any number of Runnable task.  
7) In the case of Thread, the task is executed by the Thread which accepts Runnable instance, but in the case of Execution the command (a Runnable implementation) may be executed in a new thread, a pooled thread or in the calling thread itself, depending upon the implementation of Executor interface.  
8) In the case of a thread, **it's developer's responsibility to create and start the thread**, but in the case of Executor, the framework will create and start threads for you. Though you can control the whole process by giving your implementation of Executor interface.

Essential difference between Callable and Runnable interface in Java  
1. Callable can return result  
2. Callable can throw checked Exception.  
**Runnable vs Callable**

Before looking at the difference between the Runnable and Callable interface, let's look at the similarities between them, they are indeed quite similar.

Callable callable = Executors.callable(Runnable task);

new Thread( () -> System.out.println("Runnable") ).start()

Remember, **Future.get() is a blocking method and blocks until execution is finished**, so you should always call this method with a timeout to avoid [deadlock](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html) or livelock in your application.  
  
**Synchronized vs Concurrent Collections**  
Synchronized collections like synchronized **HashMap**, **Hashtable**, **HashSet, Vector**, and **synchronized ArrayList** are much slower than their concurrent counterparts e.g. **ConcurrentHashMap, CopyOnWriteArrayList, and CopyOnWriteHashSet**. **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.**  
  
**Synchronized Collections vs Concurrent Collections in Java**

The synchronized collections classes, [Hashtable](http://java67.blogspot.com/2012/08/5-difference-between-hashtable-hashmap-Java-collection.html) and [Vector](http://java67.blogspot.com/2016/02/how-to-convert-vector-to-array-in-java.html), and the synchronized wrapper classes, Collections.synchronizedMap() and Collections.synchronizedList(), provides a basic conditionally thread-safe implementation of Map and List.  
  
**So what is the**[**difference between Hashtable and ConcurrentHashMap**](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html)**, both can be used in multi-threaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for longer duration.**  
  
**Since ConcurrentHashMap introduced concept of segmentation**, It doesn't mater whether how large it becomes because only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.

**Executor.execute() vs ExecutorService.submit() method**  
**ExecuterService.submit()*can return result of computation* because it has a return type of**[**Future**](http://javarevisited.blogspot.com/2015/01/how-to-use-future-and-futuretask-in-Java.html)**, but execute() method *cannot return anything* because it's return type is void.**

**1. The submit() can accept both**[**Runnable**](http://java67.blogspot.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html)**and**[**Callable**](http://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.html)**task but execute() can only accept the Runnable task.  
2. The submit() method is declared in ExecutorService interface while execute() method is declared in the Executor interface.  
3. The return type of submit() method is a Future object but return type of execute() method is void.**