Java Concurrency – 2021-2022

**Executors**

**java.util.concurrent.Executors.newFixedThreadPool(int)**

**java.util.concurrent.Executors.newSingleThreadExecutor()**

**java.util.concurrent.Executors.newCachedThreadPool()**

**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. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using ThreadPoolExecutor constructors.

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

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most n Threads 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.

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

Creates an Executor that uses a single worker thread operating off an unbounded queue. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent newFixedThreadPool(1) the returned executor is guaranteed not to be reconfigurable to use additional threads.

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.**

**According to Brian Goetz**

Rather than being instantiated through constructors, executors are generally instantiated through factory methods. The Executors class contains static factory methods for constructing a number of different kinds of Executor implementations:

**Executors.newCachedThreadPool()** Creates a thread pool that is not limited in size, but which will reuse previously created threads when they are 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 60 seconds are terminated and removed from the cache.

**Executors.newFixedThreadPool(int n)** 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.

**Executors.newSingleThreadExecutor()** 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.

I provide below the examples and the time taken report, have a look.

**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**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**public class TestTaskThread1** {  
 **public static void** main(String[] args) {  
 ExecutorService exService = Executors.*newFixedThreadPool*(3);  
 Thread t1 = **new** Thread(**new** TaskThread(**"Aadhar-Validation"**, 1));  
 Thread t2 = **new** Thread(**new** TaskThread(**"PanNo-Validation"**, 2));  
 Thread t3 = **new** Thread(**new** TaskThread(**"Passport-Validation"**, 3));  
 **exService.execute(t1);  
 exService.execute(t2);  
 exService.execute(t3);**  
 **exService.shutdown();  
 while (!exService.isTerminated()) {}  
 System.*out*.println("All threads completed ...");** }  
}

**Using Executors.newSingleThreadExecutor()**

import java.util.concurrent.TimeUnit;

public class Thread1 implements Runnable {

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

System.out.println(Thread.currentThread().getName()+" running ...");

TimeUnit.SECONDS.sleep(1);

}

} catch (Exception e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+" completed task ...");

}

}

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsTest1 {

public static void main(String[] args) {

long startTime = System.nanoTime();

ExecutorService exservice1 = Executors.newSingleThreadExecutor();

for( int i = 0 ; i < 10 ; i++ ) {

Runnable runnable1 = new Thread1();

exservice1.execute(runnable1);

}

exservice1.shutdown();

**while( !exservice1.isTerminated() ) {}**

System.out.println("All the threads terminated ...");

long endTime = System.nanoTime();

long diff = endTime - startTime;

long timeInSec = TimeUnit.SECONDS.convert(diff, TimeUnit.NANOSECONDS);

System.out.println("Total Time Taken : "+timeInSec);

}

}

Output

All the threads terminated ... (Executed sequentially)

Total Time Taken : 100

**Using Executors.newFixedThreadPool()**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsTest1 {

public static void main(String[] args) {

long startTime = System.nanoTime();

ExecutorService exservice1 = **Executors.newFixedThreadPool(10);**

for( int i = 0 ; i < 10 ; i++ ) {

Runnable runnable1 = new Thread1();

exservice1.execute(runnable1);

}

exservice1.shutdown();

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

long endTime = System.nanoTime();

long diff = endTime - startTime;

long timeInSec = TimeUnit.SECONDS.convert(diff, TimeUnit.NANOSECONDS);

System.out.println("Total Time Taken : "+timeInSec);

}

}

**Output**

All the threads terminated ...

Total Time Taken : 10

**Using Executors.newCachedThreadPool()**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsTest1 {

public static void main(String[] args) {

long startTime = System.nanoTime();

ExecutorService exservice1 = Executors.newCachedThreadPool();

for( int i = 0 ; i < 10 ; i++ ) {

Runnable runnable1 = new Thread1();

exservice1.execute(runnable1);

}

exservice1.shutdown();

**while( !exservice1.isTerminated() ) {}**

System.out.println("All the threads terminated ...");

long endTime = System.nanoTime();

long diff = endTime - startTime;

long timeInSec = TimeUnit.SECONDS.convert(diff, TimeUnit.NANOSECONDS);

System.out.println("Total Time Taken : "+timeInSec);

}

}

**Output**

All the threads terminated ...

Total Time Taken : 10

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

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

**Usage of Executor as per Java 8**

**public class** TestExecutors {  
  
 **public void** task(**int** time) {  
 System.***out***.println(Thread.*currentThread*().getName()+**" Executing task ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 System.***out***.println(Thread.*currentThread*().getName()+**" completed task ..."**);  
 } **catch**(InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 }  
  
 **public void** test() {  
 Runnable r1 = () -> task(7);  
 Runnable r2 = () -> task(5);  
 Runnable r3 = () -> task(3);  
  
 ExecutorService exService = Executors.*newFixedThreadPool*(3);  
 exService.execute(r1);  
 exService.execute(r2);  
 exService.execute(r3);  
  
 exService.shutdown();  
 System.***out***.println(**"All operations completed ..."**);  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestExecutors().test();  
 }  
}

**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.

import java.util.concurrent.TimeUnit;

public class Thread1 implements Runnable {

private MyShared shared;

public Thread1( MyShared shared ) {

this.shared = shared;

}

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

shared.getList().add("Thread1-"+i);

System.out.println(Thread.currentThread().getName()+" running and counter ... "+i);

TimeUnit.SECONDS.sleep(1);

}

}

catch (Exception e) { e.printStackTrace(); }

System.out.println(Thread.currentThread().getName()+" completed task ...");

System.out.println("List in Thread1 ::: "+shared.getList());

}

}

import java.util.concurrent.TimeUnit;

public class Thread2 extends Thread {

private MyShared shared;

public Thread2( MyShared shared ) {

this.shared = shared;

}

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

shared.getList().add("Thread2-"+i);

System.out.println(Thread.currentThread().getName()+" running and counter ... "+i);

TimeUnit.SECONDS.sleep(1);

}

}

catch (Exception e) { e.printStackTrace(); }

System.out.println(Thread.currentThread().getName()+" completed task ...");

System.out.println("List in Thread2 ::: "+shared.getList());

}

}

import java.util.ArrayList;

import java.util.List;

public class MyShared {

private List<String> list = new ArrayList<String>();

**ThreadLocal<List<String>> local = new ThreadLocal<List<String>>() {**

**@Override**

**protected List<String> initialValue() {**

**return list;**

**}**

**};**

**public List<String> getList() {**

**return list;**

**}**

}

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsTest2 {

public static void main(String[] args) {

long startTime = System.nanoTime();

MyShared shared = new MyShared();

ExecutorService exservice1 = Executors.newCachedThreadPool();

Runnable runnable1 = new Thread1(shared);

Thread th2 = new Thread2(shared);

**exservice1.execute(runnable1);**

**exservice1.execute(th2);**

**exservice1.shutdown();**

while( !exservice1.isTerminated() ) {

if( shared.getList().size() > 7 )

**exservice1.shutdownNow();**

}

System.out.println("My Shared List :::"+shared.getList());

System.out.println("All the threads terminated ...");

long endTime = System.nanoTime();

long diff = endTime - startTime;

long timeInSec = TimeUnit.SECONDS.convert(diff, TimeUnit.NANOSECONDS);

System.out.println("Total Time Taken : "+timeInSec);

}

}

The output will be as per the following.

pool-1-thread-1 running and counter ... 4

java.lang.InterruptedException: sleep interrupted

pool-1-thread-2 completed task ...

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

at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(Unknown Source)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)

at java.lang.Thread.run(Unknown Source)

java.lang.InterruptedException: sleep interrupted

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

at java.util.concurrent.TimeUnit.sleep(Unknown Source)List in Thread2 ::: [Thread1-0, Thread2-0, Thread2-1, Thread1-1, Thread2-2, Thread1-3, Thread2-3, Thread2-4, Thread1-4]

at Thread1.run(Thread1.java:24)

at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(Unknown Source)

pool-1-thread-1 completed task ...

List in Thread1 ::: [Thread1-0, Thread2-0, Thread2-1, Thread1-1, Thread2-2, Thread1-3, Thread2-3, Thread2-4, Thread1-4]

My Shared List :::[Thread1-0, Thread2-0, Thread2-1, Thread1-1, Thread2-2, Thread1-3, Thread2-3, Thread2-4, Thread1-4]

All the threads terminated ...

Total Time Taken : 4

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.**

As I have already mentioned that we have used ThreadLocal, let us see how we have wrongly used ThreadLocal. Some developer do like this just to avoid synchronization.

Let us consider the above two threads like Thread1 and Thread2, let us have a small test program using executor.

import java.util.Iterator;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsTest2 {

public static void main(String[] args) {

long startTime = System.nanoTime();

MyShared shared = new MyShared();

**ExecutorService exservice1 = Executors.newCachedThreadPool();**

Runnable runnable1 = new Thread1(shared);

Thread th2 = new Thread2(shared);

**exservice1.execute(runnable1);**

**exservice1.execute(th2);**

**exservice1.shutdown();**

while( !exservice1.isTerminated() ) {

for( Iterator<String> itr = shared.getList().iterator() ; itr.hasNext() ;) {

System.out.println("Items from List ::: "+itr.next());

}

}

System.out.println("My Shared List :::"+shared.getList());

System.out.println("All the threads terminated ...");

long endTime = System.nanoTime();

long diff = endTime - startTime;

long timeInSec = TimeUnit.SECONDS.convert(diff, TimeUnit.NANOSECONDS);

System.out.println("Total Time Taken : "+timeInSec);

}

}

If you run the above program you will get the following output.

pool-1-thread-2 running and counter ... 1

pool-1-thread-1 running and counter ... 1

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.AbstractList$Itr.checkForComodification(Unknown Source)

at java.util.AbstractList$Itr.next(Unknown Source)

at ExecutorsTest2.main(ExecutorsTest2.java:29)

pool-1-thread-2 running and counter ... 2

It means that the ArrayList is not a thread-safe class. Let us try to use Collections.synchronizedList().

You will get the same exception. Let us think for while can we modify the MyShared object in such a manner that we can achieve thread-safety.

The correct implementation of MyShared class is given below.

import java.util.ArrayList;

import java.util.List;

public class MyShared {

private List<String> list ;

ThreadLocal<List<String>> local = new ThreadLocal<List<String>>() {

@Override

protected List<String> initialValue() {

return list = new ArrayList<String>();

}

};

public List<String> getList() {

return local.get(); //**Always use like this.**

}

}

Now let us go back to the main concept where we want to check whether our list size is greater than 0 or not.

Since we had written MyShared class wrongly, we got the different result. Now let us see the complete program

which runs perfectly but we are unable to achieve the requirement.

import java.util.concurrent.TimeUnit;

public class Thread1 implements Runnable {

private MyShared shared;

public Thread1( MyShared shared ) {

this.shared = shared;

}

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

shared.getList().add("Thread1-"+i);

System.out.println(Thread.currentThread().getName()+" running and counter ... "+i);

TimeUnit.SECONDS.sleep(1);

}

}

// catch( InterruptedException ie ) {

// if( Thread.currentThread().isInterrupted() ) Thread.currentThread().interrupt();

// }

catch (Exception e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+" completed task ...");

System.out.println("List in Thread1 ::: "+shared.getList());

}

}

import java.util.concurrent.TimeUnit;

public class Thread2 extends Thread {

public Thread2() {

}

private MyShared shared;

public Thread2( MyShared shared ) {

this.shared = shared;

}

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

shared.getList().add("Thread2-"+i);

System.out.println(Thread.currentThread().getName()+" running and counter ... "+i);

TimeUnit.SECONDS.sleep(1);

}

}

// catch( InterruptedException ie ) {

// if( Thread.currentThread().isInterrupted() ) Thread.currentThread().interrupt();

// }

catch (Exception e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+" completed task ...");

System.out.println("List in Thread2 ::: "+shared.getList());

}

}

import java.util.ArrayList;

import java.util.List;

public class MyShared {

private List<String> list ;

ThreadLocal<List<String>> local = new ThreadLocal<List<String>>() {

@Override

protected List<String> initialValue()

{

return list = new ArrayList<String>();

}

};

public List<String> getList()

{

return local.get();

}

}

import java.util.Iterator;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorsTest2 {

public static void main(String[] args) {

long startTime = System.nanoTime();

MyShared shared = new MyShared();

ExecutorService exservice1 = Executors.newCachedThreadPool();

Runnable runnable1 = new Thread1(shared);

Thread th2 = new Thread2(shared);

exservice1.execute(runnable1);

exservice1.execute(th2);

exservice1.shutdown();

while( !exservice1.isTerminated() ) {

if( shared.getList().size() > 7 )

exservice1.shutdownNow();

}

System.out.println("My Shared List :::"+shared.getList());

System.out.println("All the threads terminated ...");

long endTime = System.nanoTime();

long diff = endTime - startTime;

long timeInSec = TimeUnit.SECONDS.convert(diff, TimeUnit.NANOSECONDS);

System.out.println("Total Time Taken : "+timeInSec);

}

}

Upon running the following is the output.

pool-1-thread-1 completed task ...

pool-1-thread-2 completed task ...

List in Thread1 ::: [Thread1-0, Thread1-1, Thread1-2, Thread1-3, Thread1-4, Thread1-5, Thread1-6, Thread1-7, Thread1-8, Thread1-9]

List in Thread2 ::: [Thread2-0, Thread2-1, Thread2-2, Thread2-3, Thread2-4, Thread2-5, Thread2-6, Thread2-7, Thread2-8, Thread2-9]

My Shared List :::[] // Local copy for the main thread, that is why it gives blank

All the threads terminated ...

Total Time Taken : 10

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 shutdown the execution.** Let us see the example.

import java.util.concurrent.TimeUnit;

public class PingThread1 implements Runnable {

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

System.out.println(Thread.currentThread().getName()+" running and counter ... "+i);

TimeUnit.SECONDS.sleep(1);

System.out.println("Still trying to ping the server ...");

}

System.out.println(Thread.currentThread().getName()+" completed task ...");

}

catch( InterruptedException ie ) {

System.out.println("Thread1 aborted ...");

if( Thread.currentThread().isInterrupted() ) Thread.currentThread().interrupt();

}

catch (Exception e) { e.printStackTrace(); }

}

}

import java.util.concurrent.TimeUnit;

public class PingThread2 implements Runnable {

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

System.out.println(Thread.currentThread().getName()+" running and counter ... "+i);

TimeUnit.SECONDS.sleep(1);

System.out.println("Still trying to ping the server ...");

}

System.out.println(Thread.currentThread().getName()+" completed task ...");

}

catch( InterruptedException ie ) {

System.out.println("Thread2 aborted ...");

if(Thread.currentThread().isInterrupted() ) Thread.currentThread().interrupt();

}

catch (Exception e) { e.printStackTrace(); }

}

}

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class TestPingThreads {

public static void main(String[] args) throws Exception {

ExecutorService exservice1 = Executors.newCachedThreadPool();

Runnable pingTh1 = new PingThread1();

Runnable pingTh2 = new PingThread2();

**exservice1.execute(pingTh1);**

**exservice1.execute(pingTh2);**

while(!exservice1.awaitTermination(3, TimeUnit.SECONDS) ) {

System.out.println("All threads are running ... It has crossed the time, still no response, better to stop it.");

exservice1.shutdownNow();

}

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

}

}

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

This method will wait for some time.

**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**); 🡸sleep() resets the interrupt flag  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.TimeUnit;  
  
**public class TestTaskThread1** {  
 **public static void** main(String[] args) {  
 ExecutorService exService = Executors.*newFixedThreadPool*(3);  
 Thread t1 = **new** Thread(**new** TaskThread(**"Aadhar-Validation"**, 1));  
 Thread t2 = **new** Thread(**new** TaskThread(**"PanNo-Validation"**, 2));  
 Thread t3 = **new** Thread(**new** TaskThread(**"Passport-Validation"**, 3));  
 exService.execute(t1);  
 exService.execute(t2);  
 exService.execute(t3);  
  
 exService.shutdown();  
 **try** {  
 **if** (!exService.awaitTermination(2, TimeUnit.***SECONDS***)) {  
 System.***out***.println(**"Going to shutdown immediately ..."**);  
 exService.shutdownNow();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 **while** (!exService.isTerminated()) {}  
 System.***out***.println(**"All threads completed ..."**);  
 }  
}

**OUTPUT**

pool-1-thread-1 running ...0

pool-1-thread-2 running ...0

pool-1-thread-3 running ...0

pool-1-thread-1 running ...1

Going to shutdown immediately ...

java.lang.InterruptedException: sleep interrupted

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

at java.lang.Thread.sleep(Thread.java:340)

at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

at com.ddlab.rnd.cyclicbarrier.TaskThread.run(TaskThread.java:20)

at java.lang.Thread.run(Thread.java:748)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)

at java.lang.Thread.run(Thread.java:748)

pool-1-thread-3 running ...1

pool-1-thread-1 running ...2

java.lang.InterruptedException: sleep interrupted

pool-1-thread-2 running ...1

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

at java.lang.Thread.sleep(Thread.java:340)

at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

at com.ddlab.rnd.cyclicbarrier.TaskThread.run(TaskThread.java:20)

at java.lang.Thread.run(Thread.java:748)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)

at java.lang.Thread.run(Thread.java:748)

pool-1-thread-1 running ...3

java.lang.InterruptedException: sleep interrupted

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

at java.lang.Thread.sleep(Thread.java:340)

at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

at com.ddlab.rnd.cyclicbarrier.TaskThread.run(TaskThread.java:20)

at java.lang.Thread.run(Thread.java:748)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)

at java.lang.Thread.run(Thread.java:748)

pool-1-thread-2 running ...2

pool-1-thread-1 running ...4

pool-1-thread-2 running ...3

pool-1-thread-3 running ...2

pool-1-thread-2 running ...4

pool-1-thread-3 running ...3

pool-1-thread-3 running ...4

All threads completed ...

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.

import java.util.concurrent.TimeUnit;

public class PingThread1 implements Runnable {

@Override

public void run() {

try {

for( int i = 0 ; i < 10 ; i++ ) {

System.out.println(Thread.currentThread().getName()+" running and counter ... "+i);

TimeUnit.SECONDS.sleep(1);

System.out.println("Still trying to ping the server ...");

}

System.out.println(Thread.currentThread().getName()+" completed task ...");

}

catch( InterruptedException ie ) {

System.out.println("Thread1 aborted ...");

if( Thread.currentThread().isInterrupted() ) Thread.currentThread().interrupt();

}

catch (Exception e) { e.printStackTrace(); }

}

}

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class TestMyCallable {

public static void main(String[] args) throws Exception {

ExecutorService exservice1 = Executors.newCachedThreadPool();

Callable<String> myCallabel = new MyCallable();

**Future<String> future = exservice1.submit(myCallabel);**

**exservice1.shutdown();**

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

System.out.println("Final Result ::: "+future.get());

//Create an instance of Runnable

exservice1 = Executors.newCachedThreadPool();

Runnable runnable = new PingThread1();

Future future1 = exservice1.submit(runnable);

exservice1.shutdown();

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

System.out.println("Final Result ::: "+future1.get());

exservice1 = Executors.newCachedThreadPool();

Runnable runnable1 = new PingThread1();

Future future2 = exservice1.submit(runnable1,"Just a status");

exservice1.shutdown();

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

System.out.println("Final Result ::: "+future2.get());

}

}

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class TestMyCallable {

public static void main(String[] args) throws Exception {

ExecutorService exservice1 = Executors.newCachedThreadPool();

Callable<String> myCallabel = new MyCallable();

**Future<String> future = exservice1.submit(myCallabel);**

**exservice1.shutdown();**

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

System.out.println("Final Result ::: "+future.get());

//Create an inastance of Runnable

exservice1 = Executors.newCachedThreadPool();

Runnable runnable = new PingThread1();

Future future1 = exservice1.submit(runnable);

exservice1.shutdown();

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

System.out.println("Final Result ::: "+future1.get());

exservice1 = Executors.newCachedThreadPool();

Runnable runnable1 = new PingThread1();

Future future2 = exservice1.submit(runnable1,"Just a status");

exservice1.shutdown();

while( !exservice1.isTerminated() ) {}

System.out.println("All the threads terminated ...");

System.out.println("Final Result ::: "+future2.get());

}

}

**OUTPUT**

All the threads terminated ...

Final Result ::: Successful

pool-2-thread-1 completed task ...

All the threads terminated ...

Final Result ::: null

pool-3-thread-1 completed task ...

All the threads terminated ...

Final Result ::: Just a status

Note :

If you do like this, you will get an exception.

ExecutorService exservice1 = Executors.newCachedThreadPool();

....

exservice1.shutdown();

exservice1.submit(runnable);

java.util.concurrent.RejectedExecutionException

at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(Unknown Source)

It means that once you shutdown the ExecutorService, you can not reuse to execute another task.

If you want to use it, you can do like this **exservice1 = Executors.newCachedThreadPool();**

**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 .**

<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);**

You know myFuture.get() will return null in this case but only after the task has been run, so you would use it to wait and throw any exception that thrown in the task.

An example is given below.

**import** java.util.concurrent.TimeUnit;  
**public class RunnableTask** **implements** Runnable {  
 **private int timeInSecs**;  
 **private** String **result**;  
 **private** StringBuilder **builder**;  
  
 **public** RunnableTask(**int** timeInSecs, String result, StringBuilder builder) {  
 **this**.**timeInSecs** = timeInSecs;  
 **this**.**result** = result;  
 **this**.**builder** = builder;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" executing now"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed"**);  
 **this**.**result** = **"I am done"**;  
 **builder**.append(**this**.**result**);  
 } **catch** (Exception e) { e.printStackTrace(); }  
 }  
}

**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.Future;  
**public class SubmitRunnableTaskWithT** {  
 **public static void** main(String[] args) **throws** Exception {  
 String response1 = **"Done"**;  
 String threadResponse = **null**; 🡸 Mark it  
 StringBuilder builder = **new** StringBuilder(); 🡸Mark it  
 ExecutorService executorService = Executors.*newCachedThreadPool*();  
 **Runnable runnable = new RunnableTask(3, threadResponse, builder);  
 Future<String> future = executorService.submit(runnable, response1);**  
 **System.*out*.println("Final Result : " + future.get());  
 System.*out*.println("Response from Thread : " + threadResponse);  
 System.*out*.println("Response from Thread using StringBuilder: " + builder.toString());** executorService.shutdown();  
 }  
}

**OUTPUT**

pool-1-thread-1 executing now

pool-1-thread-1 completed

Final Result : Done

Response from Thread : null

Response from Thread using StringBuilder: I am done

**Note: You know that String is immutable, so simply by passing to thread, you will never get the result from a thread where StringBuffer or StringBuilder are not immutable.**

**submit()**

It has three variations.

1. **<T> Future<T> submit(Callable<T> task)**
2. **Future<?> submit(Runnable task)**
3. **<T> Future<T> submit(Runnable task, T result)**

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

Example is given below.

**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.TimeUnit;  
**public class CallableTask** **implements** Callable<String> {  
 **private** String **taskName**;  
 **private int timeInSecs**;  
  
 **public** CallableTask(String taskName, **int** timeInSecs) {  
 **this**.**taskName** = taskName;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public String call() throws Exception** {  
 **try** {  
 System.***out***.println(**this**.**taskName** + **" executing now"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(**this**.**taskName** + **" completed"**);  
 } **catch** (Exception e) { }  
 **return this**.**taskName** + **"-Response"**;  
 }  
}

**import** java.util.concurrent.\*;  
**public class SubmitCallable1** {  
 **public static void** main(String[] args) {  
 **Callable<String> callableTask = new CallableTask("Task-1", 3);  
 ExecutorService executorService = Executors.*newSingleThreadExecutor*();  
 Future<String> future = executorService.submit(callableTask);**  
 **try** {  
 System.***out***.println(**"Final Result : "** + **future.get()**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) { e.printStackTrace(); }  
 executorService.shutdown();  
 }  
}

**OUTPUT**

Task-1 executing now

Task-1 completed

Final Result : Task-1-Response

**Callable using Java 8 lambda expression**

**public class** TestExecutors {  
  
 **public** String task(String name, **int** time) {  
 System.***out***.println(Thread.*currentThread*().getName() + **" Executing task ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed task ..."**);  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 **return** name;  
 }  
  
 **public void** test() {  
 Callable<String> callable = () -> task(**"Task-1"**, 5);  
  
 ExecutorService exService = Executors.*newFixedThreadPool*(3);  
 Future<String> future = exService.submit(callable);  
  
 exService.shutdown();  
 **try** {  
 String response = future.get();  
 System.***out***.println(**"Response : "**+response);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"All operations completed ..."**);  
}  
  
 **public static void** main(String[] args) {  
 **new** TestExecutors().test();  
 }  
}

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

Example is given below.

**import** java.util.concurrent.TimeUnit;  
**public class RunnableTask** **implements** Runnable {  
  
 **private int timeInSecs**;  
  
 **public** RunnableTask(**int** timeInSecs) {  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" executing now"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed"**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.Future;  
  
**public class** SubmitRunnable {  
 **public static void** main(String[] args) **throws** Exception {  
 ExecutorService executorService = Executors.*newCachedThreadPool*();  
 Thread t1 = **new** Thread(**new** RunnableTask(5), **"Thread-1"**);  
 Future<?> future = executorService.submit(t1);  
 System.***out***.println(**"Result = "** + future.get());  
  
 *// You can also write like this  
 // Runnable runnable = new RunnableTask(5);  
 // Future<?> future = executorService.submit(runnable);  
 // System.out.println("Result = " + future.get());* executorService.shutdown();  
 }  
}

**OUTPUT**

pool-1-thread-1 executing now

pool-1-thread-1 completed

Result = null

Question: What will happen, if I pass a String to a thread as a result, can I get response. Answer is **NO.**

**execute()**

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

**import** java.util.concurrent.TimeUnit;  
**public class RunnableTask** **implements** Runnable {  
 **private int timeInSecs**;  
 **private** String **result**;  
 **private** StringBuilder **builder**;  
  
 **public** RunnableTask(**int** timeInSecs, String result, StringBuilder builder) {  
 **this**.**timeInSecs** = timeInSecs;  
 **this**.**result** = result;  
 **this**.**builder** = builder;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" executing now"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed"**);  
 **this**.**result** = **"I am done"**;  
 **builder**.append(**this**.**result**);  
 } **catch** (Exception e) { e.printStackTrace(); }  
 }  
}

**OUTPUT**

pool-1-thread-1 executing now

pool-1-thread-2 executing now

pool-1-thread-1 completed

pool-1-thread-2 completed

All threads completed ...

**import** java.util.concurrent.TimeUnit;  
**public class ThreadTask** **extends** Thread {  
 **private** String **name**;  
 **public** ThreadTask(String name) {  
 **super**(name);🡸It has no impact in case Executor  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" executing now"**);  
 TimeUnit.***SECONDS***.sleep(5);  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed"**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**public class TestExecutor1** {  
 **public static void** main(String[] args) {  
 StringBuilder builder = **new** StringBuilder();  
 Thread t1 = **new** Thread(**new** RunnableTask(2, **null**, builder));  
 Thread t2 = **new** ThreadTask(**"NewThread"**);  
 ExecutorService executorService = Executors.*newFixedThreadPool*(2);  
 executorService.execute(t1); 🡸 Thread implementing Runnable  
 executorService.execute(t2); 🡸Thread extending Thread  
 executorService.shutdown();  
 **while** (!executorService.isTerminated()) {} 🡸A blocker  
 System.***out***.println(**"All threads completed ..."**);  
 }  
}

**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));**

//**This loop must process the tasks in the order they were submitted: A, B, C, D, E**

**for (Future<?> future:futures) {**

**? result = future.get();**

**// Some processing here**

**}**

The problem with this method is that there is no guarantee that task A will complete first. Thus, it is possible that the main thread will be blocking idly waiting for task A to complete when it could be processing the result of another task (say task B). Result processing latency could be reduced by using an ExecutorCompletionService.

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

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

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

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

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

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

//**This for loop will process the tasks in the order they are completed**, regardless of submission order

**for (int i=0; i<futures.size(); i++) {**

**? result = executorCompletionService.take().get();**

**// Some processing here**

**}**

So, in essence, **ExecutorCompletionService could be used to squeeze out a little more efficiency when the order of processing task results does not matter**.

One important thing to note though. 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**

**Question-1**. If you have 5 threads, all the 5 threads should run asynchronously but I want all the result in the **order of submission**. How will you do it?

**Question-2**: If you have 5 threads, all the 5 threads should run asynchronously but I want all the result in the **order of completion**. How will you do it?

Let us consider first problem. I have 5 threads/tasks, all are running asynchronously, but the results should be obtained in the order of submission.

The code is given below.

**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.concurrent.\*;  
**public class TestInOrder** {  
 **public static void** main(String[] args) **throws** Exception {  
 Callable a1 = **new** CallableTask(**"A1"**, 9);  
 Callable b1 = **new** CallableTask(**"B1"**, 7);  
 Callable c1 = **new** CallableTask(**"C1"**, 5);  
 Callable d1 = **new** CallableTask(**"D1"**, 3);  
 Callable e1 = **new** CallableTask(**"E1"**, 1);  
 List<Future<String>> futureList = **new** ArrayList<>();  
 **ExecutorService executorService = Executors.*newFixedThreadPool*(5);**  
  
 **futureList.add(executorService.submit(a1));  
 futureList.add(executorService.submit(b1));  
 futureList.add(executorService.submit(c1));  
 futureList.add(executorService.submit(d1));  
 futureList.add(executorService.submit(e1));****futureList.forEach(  
 future -> {  
 try {  
 System.*out*.println("Result : " + future.get());  
 } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }  
 });**  
 executorService.shutdown();  
 }  
}

**OUTPUT**

A1 started ...

B1 started ...

C1 started ...

D1 started ...

E1 started ...

**Result : A1-Completed**

**Result : B1-Completed**

**Result : C1-Completed**

**Result : D1-Completed**

**Result : E1-Completed**

**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.TimeUnit;  
**public class CallableTask** **implements** Callable<String> {  
 **private** String **taskName**;  
 **private int timeInSecs**;  
 **public** CallableTask(String taskName, **int** timeInSecs) {  
 **this**.**taskName** = taskName;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public** String call() **throws** Exception {  
 System.***out***.println(**this**.**taskName**+**" started ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 } **catch** (Exception e) { e.printStackTrace(); }  
 **return this**.**taskName** + **"-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.**

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.

**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.concurrent.\*;  
**public class TestCompletionService** {  
 **public static void** main(String[] args) {  
 Callable a1 = **new** CallableTask(**"A1"**, 9);  
 Callable b1 = **new** CallableTask(**"B1"**, 7);  
 Callable c1 = **new** CallableTask(**"C1"**, 5);  
 Callable d1 = **new** CallableTask(**"D1"**, 3);  
 Callable e1 = **new** CallableTask(**"E1"**, 1);  
  
 List<Future<String>> futureList = **new** ArrayList<>();  
 ExecutorService executorService = Executors.*newFixedThreadPool*(5);  
 **CompletionService<String> completionService = new ExecutorCompletionService<>(executorService);**  
 **futureList.add(completionService.submit(a1));  
 futureList.add(completionService.submit(b1));  
 futureList.add(completionService.submit(c1));  
 futureList.add(completionService.submit(d1));  
 futureList.add(completionService.submit(e1));**  
 **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();  
 }  
}

**OUTPUT**

A1 started ...

B1 started ...

C1 started ...

E1 started ...

D1 started ...

**Result : E1-Completed**

**Result : D1-Completed**

**Result : C1-Completed**

**Result : B1-Completed**

**Result : A1-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.

**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.TimeUnit;  
**public class Missile** **implements** Callable<String> {  
 **private** String **missileName**;  
 **private** String **attackCityName**;  
 **private int timeInSecs**;  
  
 **public** Missile(String missileName, String attackCityName, **int** timeInSecs) {  
 **this**.**missileName** = missileName;  
 **this**.**attackCityName** = attackCityName;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public** String call() **throws** Exception {  
 System.***out***.println(**this**.**missileName** + **" started firing..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return this**.**missileName** + **" hit city "** + **this**.**attackCityName**+**" in "**+**timeInSecs**+**" seconds"**;  
 }  
}

**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.concurrent.\*;  
**public class InidanMissileTest** {  
 **public static void** main(String[] args) {  
 ExecutorService executorService = Executors.*newFixedThreadPool*(5);  
 **CompletionService<String> completionService = new ExecutorCompletionService<>(executorService);**  
  
 List<Future> futureList = **new** ArrayList<>();  
  
 Callable<String> brahmos = **new** Missile(**"Brahmos"**, **"Karachi (Pakistan)"**, 11);  
 Callable<String> agnee = **new** Missile(**"Agnee"**, **"Lahore (Pakistan)"**, 13);  
 Callable<String> nag = **new** Missile(**"Nag"**, **"Islamabad (Pakistan)"**, 5);  
 Callable<String> prithvi = **new** Missile(**"Prithvi"**, **"Rawalpindi (Pakistan)"**, 7);  
 Callable<String> trishul = **new** Missile(**"Trishul"**, **"Peshawar (Pakistan)"**, 9);  
  
 **futureList.add(completionService.submit(brahmos));  
 futureList.add(completionService.submit(agnee));  
 futureList.add(completionService.submit(nag));  
 futureList.add(completionService.submit(prithvi));  
 futureList.add(completionService.submit(trishul));**  
  
 executorService.shutdown();  
  
 **for** (**int** i = 0; i < futureList.size(); i++) {  
 **try {  
 Future<String> future = completionService.take();  
 System.*out*.println("Missile attack result : " + future.get());  
 } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }** }  
 }  
}

**OUTPUT**

Agnee started firing...

Brahmos started firing...

Nag started firing...

prithvi started firing...

Trishul started firing...

**Missile attack result : Nag hit city Islamabad (Pakistan) in 5 seconds**

**Missile attack result : prithvi hit city Rawalpindi (Pakistan) in 7 seconds**

**Missile attack result : Trishul hit city Peshawar (Pakistan) in 9 seconds**

**Missile attack result : Brahmos hit city Karachi (Pakistan) in 11 seconds**

**Missile attack result : Agnee hit city Lahore (Pakistan) in 13 seconds**

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

**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.

Let us refer to the below code as an example.

**##Using invokeAll**

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.Executors;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Future;

public final class CompletionExample {

static class SleepingCallable implements Callable<String> {

final String name;

final long period;

SleepingCallable(final String name, final long period) {

this.name = name;

this.period = period;

}

public String call() {

try {

Thread.sleep(period);

} catch (InterruptedException ex) { }

return name;

}

}

public static void main(final String[] argv) {

final ExecutorService pool = Executors.newFixedThreadPool(2);

final List<? extends Callable<String>> callables = Arrays.asList(

new SleepingCallable("quick", 500),

new SleepingCallable("slow", 5000));

try {

for (final Future<String> future : pool.invokeAll(callables)) {

System.out.println(future.get());

}

} catch (ExecutionException | InterruptedException ex) { }

pool.shutdown();

}

}

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 void startMyApplication() {

ExecutorService executor = Executors.newFixedThreadPool(2);

FuturTask<Object> futureOne = new FutureTask<Object>(myFirstProcess);

FuturTask<Object> futureTwo = new FutureTask<Object>(mySecondProcess);

executor.execute(futureOne);

executor.execute(futureTwo);

**while (!(futureOne.isDone() && futureTwo.isDone()))** {

try {

// I wait until both processes are finished.

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

logger.info("Processing finished");

executor.shutdown();

// Do some processing on results

...

}

<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.

The following example illustrates the use of the Futuretask together with the Callable.

It reverses the submitted string slowly at a rate of one char per second. In the mean time, the main thread keeps polling to see if the task is completed. Example of FutureTask

import java.util.concurrent.\*;

import java.util.concurrent.ExecutorService;

public class MyStringReverser {

ExecutorService executor = Executors.newFixedThreadPool(1);

SlowStringReverser reverser = new SlowStringReverser();

void doReverse(final String target) throws InterruptedException {

**FutureTask<String> future = new FutureTask<String>(**

**new Callable<String>() {**

**public String call() {**

**return reverser.reverseString(target);**

**}**

**});**

executor.execute(future);

// try every 10 seconds

while (!future.isDone()) {

System.out.println("Task not yet completed.");

Try {

Thread.sleep(500);

} catch (InterruptedException ie) {

System.out.println("Will check after 1/2 sec.");

}

}

try {

System.out.println("Here is the result..." + future.get());

} catch (ExecutionException ex) { }

executor.shutdown();

return;

}

public static void main(String args[]) {

MyStringReverser msr = new MyStringReverser(); {

try {

msr.doReverse("foobar");

} catch (Exception e) { e.printStackTrace(); }

}

}

class SlowStringReverser {

StringBuffer orgString;

StringBuffer reversedString;

SlowStringReverser(String orgStr) {

orgString = new StringBuffer(orgStr);

}

SlowStringReverser() { }

public String reverseString(String str) {

orgString = new StringBuffer(str);

reversedString = new StringBuffer();

for (int i = (orgString.length() - 1); i >= 0; i--) {

reversedString.append(orgString.charAt(i));

System.out.println("Reversing one character per second."

+ reversedString);

try {

Thread.sleep(1000);

} catch (InterruptedException ie) { }

}

return reversedString.toString();

}

}

**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.

**package** com.ddlab.rnd.future;  
**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.TimeUnit;  
**public class MyFutureCallable** **implements** Callable<String> {  
  
 @Override  
 **public** String call() **throws** Exception {  
 String retString = **"got"**;  
 **try** {  
 TimeUnit.***SECONDS***.sleep(20);  
 System.***out***.println(**"Execution completed ... "**);  
 }  
 **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 **return** retString;  
 }  
}

**package** com.ddlab.rnd.future;  
**import** java.util.concurrent.\*;  
**public class Test1** {  
  
 **public static void** main(String[] args) {  
 Callable th1 = **new** MyFutureCallable();  
 ExecutorService executorService = Executors.*newFixedThreadPool*(1);  
 Future<String> future = executorService.submit(th1);  
 **try** {  
 **while** (!future.isDone()) {  
 *//wait here and do some other operation* System.***out***.println(**"I am doing something in the mean time ..."**);  
 }  
 String s = future.get();  
 System.***out***.println(**"Final Value --->"**+s);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"Complted ..."**);  
 executorService.shutdown();  
 }  
}

Output

I am doing something in the mean time ...

I am doing something in the mean time ...

…..

Final Value --->got

Complted ...

**Executors**

An example is given below.

import java.util.concurrent.TimeUnit;

public class CommonUtil2 {

public static void doTask1() {

for( int i = 0 ; i < 5 ; i++ ) {

System.out.println("Thread Name in Task1 :::"+Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(1);

}

catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void doTask2() {

for( int i = 0 ; i < 5 ; i++ ) {

System.out.println("Thread Name in Task2 :::"+Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(1);

}

catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

In the above program there are two methods, each method takes 5 mins. I want to execute both the methods, it takes 10 seconds.

Let us see the program below.

import java.util.concurrent.TimeUnit;

public class Test2 {

public static void doSequential() {

long startTime = System.nanoTime();

**CommonUtil2.doTask1();**

**CommonUtil2.doTask2();**

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

public static void main(String[] args) {

new Test2().doSequential();

}

}

The final output of the above program will be

Time Difference :::10137673475 nano seconds

Time Difference :::10 seconds

It means in case of sequential approach, it takes toatl 10 seconds. Let use see how it happens in case of multi threaded application. In order to execute the above two tasks, let us create two different threads, each thread will perform an independent task.

public class Thread1 implements Runnable {

@Override

public void run() {

**CommonUtil2.doTask1();**

}

}

The above program is a thread which will perform task1.

public class Thread2 implements Runnable {

@Override

public void run() {

**CommonUtil2.doTask2();**

}

}

The above program is a thread which will perform task2.

Let us write a final program to execute the threads.

import java.util.concurrent.TimeUnit;

public class Test2 {

public static void doMultithreading() {

long startTime = System.nanoTime();

Thread a = new Thread( new Thread1());

a.setName("T1");

Thread b = new Thread( new Thread2());

b.setName("T2");

a.start();

b.start();

boolean flag = true;

while( flag ) {

if( !a.isAlive() && !b.isAlive()) {

System.out.println("All threads are dead....");

flag = false;

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

}

}

public static void main(String[] args) {

new Test2().doMultithreading();

}

}

If you run the above program, the final output will be.

Time Difference :::5075613659 nano seconds

Time Difference :::5 seconds

It means now our program is faster as we are able to perform both the tasks in 5 seconds. isn't a good choice.

Now let us use java concurrency feature like ExecutorService. Let us modify the above program. Let us see the code below.

import java.lang.management.ManagementFactory;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class Test2 {

public void doMultithreading1() {

Runnable thread1 = new Thread1();

Runnable thread2 = new Thread2();

long startTime = System.nanoTime();

// ExecutorService executorService = Executors.newSingleThreadExecutor();

// ExecutorService executorService = Executors.newFixedThreadPool(2);

ExecutorService executorService = Executors.newCachedThreadPool();

executorService.execute(thread1);

executorService.execute(thread2);

executorService.shutdown();

while (!executorService.isTerminated()) { }

System.out.println("Executor completed all the tasks, let us define a new one");

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

public static void main(String[] args) {

new Test2().doMultithreading1();

}

}

The above program takes only 5 seconds to complete the activities. In this case it is very simple to use. In the above case we use only the line **"while (!executorService.isTerminated()) { } "** to track whether all the threads have completed their tasks or not.

**Example is given below.**

import java.util.concurrent.TimeUnit;

public class Test0 {

public static void main(String[] args) {

try {

long startTime = System.nanoTime();

Thread th1 = new Thread( new Thread1() );

th1.setName("T1");

Thread th2 = new Thread( new Thread2() );

th2.setName("T2");

//Always write join() method after the start() method of a thread

th1.start();

th1.join();

th2.start();

th2.join();

//**This is wrong to write**

// th1.join();

// th2.join();

boolean flag = true;

while( flag ) {

if( !th1.isAlive() && !th2.isAlive()) {

System.out.println("All threads are dead....");

flag = false;

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

}

}

catch (Exception e) {

e.printStackTrace();

}

}

}

Now let use see using the feature of **java.util.Concurren**t package to execute the above two threads on by one. In this case we will not use Thread.join() method, but we will do it sequentially.

public class Test0 {

public static void doMultithreading2() {

try {

**Runnable th1 = new Thread1() ;**

**ExecutorService service1 = Executors.newSingleThreadExecutor();**

**service1.execute(th1);**

**service1.shutdown();**

while( !service1.isTerminated() ){}

**Runnable th2 = new Thread2() ;**

**ExecutorService service2 = Executors.newSingleThreadExecutor();**

**service2.execute(th2);**

**service2.shutdown();**

**while( !service2.isTerminated() ){}**

} catch (Exception e) { e.printStackTrace(); }

}

public static void main(String[] args) {

doMultithreading2();

}

}

**ExecutorService Implementations**

The implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

**Creating an ExecutorService**

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too. Here are a few examples:

**ExecutorService executorService1 = Executors.newSingleThreadExecutor();**

**ExecutorService executorService2 = Executors.newFixedThreadPool(10);**

**ExecutorService executorService3 = Executors.newScheduledThreadPool(10);**

ExecutorService Usage, There are a few different ways to delegate tasks for execution to an ExecutorService:

**execute(Runnable)**

**submit(Runnable)**

**submit(Callable)**

**invokeAny(...)**

**invokeAll(...)**

**void execute(Runnable) :** The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example.

**ExecutorService executorService = Executors.newSingleThreadExecutor();**

**executorService.execute(new Runnable() {**

**public void run() {**

**System.out.println("Asynchronous task");**

**}**

**});**

**executorService.shutdown();**

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

**Future submit(Runnable) :The submit(Runnable) method also takes a Runnable implementation, but returns a Future object**. This Future object can be used to check if the Runnable as finished executing.

Here is a submit() example:

**Future future = executorService.submit(new Runnable() {**

**public void run() {**

**System.out.println("Asynchronous task");**

**}**

**});**

**future.get(); //returns null if the task has finished correctly.**

**Future submit(Callable) :**The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is a code example:

**Future future = executorService.submit(new Callable(){**

**public Object call() throws Exception {**

**System.out.println("Asynchronous Callable");**

**return "Callable Result";**

**}**

**});**

**System.out.println("future.get() = " + future.get());**

The above code example will output this:

Asynchronous Callable

**future.get() = Callable Result**

**invokeAny():** The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled. Here is a code example:

**Example – Type-1**  
**import** java.util.HashSet;  
**import** java.util.Set;  
**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
  
**public class** InvokeAnyTest {  
 **public static void** main(String[] args) **throws** Exception {  
 **ExecutorService executorService = Executors.*newSingleThreadExecutor*();** 🡸 Single Thread  
 Set<Callable<String>> callables = **new** HashSet<Callable<String>>();  
  
 callables.add(  
 **new** Callable<String>() {  
 **public** String call() **throws** Exception {  
 **return "Task 1"**;  
 }  
 });  
 callables.add(  
 **new** Callable<String>() {  
 **public** String call() **throws** Exception {  
 **return "Task 2"**;  
 }  
 });  
 callables.add(  
 **new** Callable<String>() {  
 **public** String call() **throws** Exception {  
 **return "Task 3"**;  
 }  
 });  
  
 **String result = executorService.invokeAny(callables);**  
 System.***out***.println(**"result = "** + result);  
 executorService.shutdown();  
 }  
}

**OUTPUT**

result = Task 3

**executorService.invokeAny() 🡸 Blocking method**

**Example Type – 2**  
**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.TimeUnit;  
**public class CallableTask** **implements** Callable<String> {  
 **private** String **taskName**;  
 **private int timeInSecs**;  
 **public** CallableTask(String taskName, **int** timeInSecs) {  
 **this**.**taskName** = taskName;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public String call() throws Exception** {  
 **try** {  
 System.***out***.println(**this**.**taskName** + **" executing now"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(**this**.**taskName** + **" completed"**);  
 } **catch** (Exception e) {  
  
 }  
 **return this**.**taskName** + **"-Completed"**;  
 }  
}

**import** java.util.ArrayList;  
**import** java.util.HashSet;  
**import** java.util.List;  
**import** java.util.Set;  
**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.ExecutionException;  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
  
**public class InvokeAnyTest1** {  
 **public static void** main(String[] args) {  
 *// ExecutorService executorService = Executors.newCachedThreadPool();* **ExecutorService executorService = Executors.*newFixedThreadPool*(3);** 🡸   
  
 List<Callable<String>> callableList = **new** ArrayList<>();  
 CallableTask task1 = **new** CallableTask(**"Task-1"**, 5);  
 CallableTask task2 = **new** CallableTask(**"Task-2"**, 3);  
 CallableTask task3 = **new** CallableTask(**"Task-3"**, 7);  
  
 callableList.add(task1);  
 callableList.add(task2);  
 callableList.add(task3);  
 **try** {  
 **String result = executorService.invokeAny(callableList);**  
 System.***out***.println(**"Result : "** + result);  
 executorService.shutdown();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) { e.printStackTrace(); }  
 }  
}

OUTPUT

Task-1 executing now

Task-2 executing now

Task-3 executing now

Task-2 completed

Result : Task-2-Completed

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

**List<Future> invokeAll():** The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable. Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference. Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

**public String call() throws Exception {**

**return "Task 1";**

**}**

**});**

**callables.add(new Callable<String>() {**

**public String call() throws Exception {**

**return "Task 2";**

**}**

**});**

**callables.add(new Callable<String>() {**

**public String call() throws Exception {**

**return "Task 3";**

**}**

**});**

List<Future<String>> futures = **executorService.invokeAll(callables);**

for(Future<String> future : futures) {

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.TimeUnit;  
  
**public class CallableTask** **implements** Callable<String> {  
 **private** String **taskName**;  
 **private int timeInSecs**;  
  
 **public** CallableTask(String taskName, **int** timeInSecs) {  
 **this**.**taskName** = taskName;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public** String call() **throws** Exception {  
 **try** {  
 System.***out***.println(**this**.**taskName** + **" executing now"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(**this**.**taskName** + **" completed"**);  
 } **catch** (Exception e) {  
  
 }  
 **return this**.**taskName** + **"-Response"**;  
 }  
}

**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.concurrent.\*;  
  
**public class InvokeAllTest1** {  
 **public static void** main(String[] args) {  
 ExecutorService executorService = Executors.*newFixedThreadPool*(3);  
  
 List<Callable<String>> callableList = **new** ArrayList<>();  
 CallableTask task1 = **new** CallableTask(**"Task-1"**, 5);  
 CallableTask task2 = **new** CallableTask(**"Task-2"**, 3);  
 CallableTask task3 = **new** CallableTask(**"Task-3"**, 7);  
  
 callableList.add(task1);  
 callableList.add(task2);  
 callableList.add(task3);  
  
 **try** {  
 **List<Future<String>> futureList = executorService.invokeAll(callableList);** futureList.forEach(  
 futureObj -> {  
 **try** {  
 System.***out***.println(**"Actual Result : "** + **futureObj.get());** } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }  
 });  
 executorService.shutdown();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"All completed ..."**);  
 }  
}

OUTPUT

Task-1 executing now

Task-2 executing now

Task-3 executing now

Task-2 completed

Task-1 completed

Task-3 completed

Actual Result : Task-1-Response

Actual Result : Task-2-Response

Actual Result : Task-3-Response

All completed ...

**Closing an ExecutorService**

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.

**Some of the self developed programs on executors are given below.**

import java.util.concurrent.TimeUnit;

public class CommonUtil2 {

public static void doTask1() {

for( int i = 0 ; i < 5 ; i++ ) {

System.out.println("Thread Name in Task1 :::"+Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(1);

}

catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void doTask2() {

for( int i = 0 ; i < 5 ; i++ ) {

System.out.println("Thread Name in Task2 :::"+Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(1);

}

catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static synchronized void doTask3() {

for( int i = 0 ; i < 5 ; i++ ) {

System.out.println("Thread Name in Task2 :::"+Thread.currentThread().getName());

try {

System.out.println("Thread Name in Task2 Before Sleep:::"+Thread.currentThread().getName());

TimeUnit.SECONDS.sleep(1);

System.out.println("Thread Name in Task2 After Sleep:::"+Thread.currentThread().getName());

}

catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Thread Name in Task2 After Try-Catch:::"+Thread.currentThread().getName());

}

}

public class Thread2 implements Runnable {

@Override

public void run() {

CommonUtil2.doTask3();

}

}

public class Thread1 implements Runnable {

@Override

public void run() {

CommonUtil2.doTask3();

}

}

}

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class Test2 {

public static void doSequential() {

long startTime = System.nanoTime();

CommonUtil2.doTask1();

CommonUtil2.doTask2();

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

public static void doMultithreading() {

long startTime = System.nanoTime();

Thread a = new Thread( new Thread1());

a.setName("T1");

Thread b = new Thread( new Thread2());

b.setName("T2");

a.start();

b.start();

boolean flag = true;

while( flag ) {

if( !a.isAlive() && !b.isAlive()) {

System.out.println("All threads are dead....");

flag = false;

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

}

}

public void doMultithreading1() {

Runnable thread1 = new Thread1();

Runnable thread2 = new Thread2();

long startTime = System.nanoTime();

// ExecutorService executorService = Executors.newSingleThreadExecutor();

// ExecutorService executorService = Executors.newFixedThreadPool(2);

ExecutorService executorService = Executors.newCachedThreadPool();

executorService.execute(thread1);

executorService.execute(thread2);

executorService.shutdown();

while (!executorService.isTerminated()) { }

System.out.println("Executor completed all the tasks, let us define a new one");

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

public static void main(String[] args) {

**final String jvmName = ManagementFactory.getRuntimeMXBean().getName();**

System.out.println(jvmName);

System.out.println("Total Processors :::"+Runtime.getRuntime().availableProcessors());

// new Test2().doSequential();

// new Test2().doMultithreading();

new Test2().doMultithreading1();

}

}

**Advantages of Executors**

**ExecutorService abstracts away many of the complexities associated with the lower-level abstractions like raw Thread. It provides mechanisms for safely starting, closing down, submitting, executing, and blocking on the successful or abrupt termination of tasks (expressed as Runnable or Callable).**

Using the Runnable interface is better than extending the Thread class. One of the main advantages of doing that is the ability of using an Executor framework instead of directly using the Thread class to create and execute new threads: constructing new threads is relatively expensive because it involves interaction with the operating system.

The Executor framework is a framework for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies.

public class Runner {

public static void main(String[] args) {

Runnable runnable = new SomeRunnable();

// create a fixed thread pool

ExecutorService pool = Executors.newFixedThreadPool(3);

// run the task 5 times using the pool

for (int i = 0; i < 5; i++) {

pool.execute(runnable);

}

pool.shutdown();

}

}

In the above example I used the ExecutorService interface which is an Executor that provides a number of useful methods to control the task execution. For creation of the concrete ExecutorService I used the Executors class and one of its static factory methods: newFixedThreadPool().

**Task scheduling:** The **ScheduledExecutorService** is the Executor that supports scheduling of tasks based on the relative time or to run task periodically. Example of scheduling a task to run after a 1000 ms:

public class Runner {

public static void main(String[] args) {

Runnable runnable = new SomeRunnable();

ScheduledExecutorService thread = Executors

.newSingleThreadScheduledExecutor();

thread.schedule(runnable, 1000, TimeUnit.MILLISECONDS);

thread.shutdown();

}

}

**Difference between Executor, ExecutorService and Executors class in Java**  
The main difference between Executor, ExecutorService, and Executors class is that **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. Once the shutdown is called, the thread pool will not accept new task but complete any pending task. It also provides a [submit()](http://javarevisited.blogspot.sg/2016/04/difference-between-ExecutorServie-submit-vs-Executor-execute-method-in-Java.html) method which extends Executor.execute() method and returns a Future.  
  
**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.**

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| boolean | [**awaitTermination**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#awaitTermination-long-java.util.concurrent.TimeUnit-)(long timeout, **[TimeUnit](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html" \o "enum in java.util.concurrent)** 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**](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)<[**Future**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<T>> | [**invokeAll**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAll-java.util.Collection-)([**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<? extends [**Callable**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)<T>> tasks)  Executes the given tasks, returning a list of Futures holding their status and results when all complete. |
| <T> [**List**](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)<[**Future**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<T>> | [**invokeAll**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAll-java.util.Collection-long-java.util.concurrent.TimeUnit-)([**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<? extends [**Callable**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)<T>> tasks, long timeout, **[TimeUnit](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html" \o "enum in java.util.concurrent)** 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**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAny-java.util.Collection-)([**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<? extends [**Callable**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)<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**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAny-java.util.Collection-long-java.util.concurrent.TimeUnit-)([**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<? extends [**Callable**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)<T>> tasks, long timeout, **[TimeUnit](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html" \o "enum in java.util.concurrent)** 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**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#isShutdown--)()  Returns true if this executor has been shut down. |
| boolean | [**isTerminated**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#isTerminated--)()  Returns true if all tasks have completed following shut down. |
| void | [**shutdown**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#shutdown--)()  Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. |
| [**List**](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)<[**Runnable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html)> | [**shutdownNow**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#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**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<T> | [**submit**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit-java.util.concurrent.Callable-)([**Callable**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)<T> task)  Submits a value-returning task for execution and returns a Future representing the pending results of the task. |
| [**Future**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<?> | [**submit**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit-java.lang.Runnable-)([**Runnable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) task)  Submits a Runnable task for execution and returns a Future representing that task. |
| <T> [**Future**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)<T> | [**submit**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit-java.lang.Runnable-T-)([**Runnable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) task, T result)  Submits a Runnable task for execution and returns a Future representing that task. |