J[ava Concurrency - Part 7 : Executors and thread pools](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.html)

 Baptiste Wicht

[2010-09-15 07:17](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.html)

[43 Comments](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.html#disqus_thread)

[Source](http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-7-executors-and-thread-pools.wp)

Let's start with a new post in the Java concurrency series.

This time we'll learn how to start cleanly new threads and to manage thread pools. In Java, if you have a Runnable like this :

Runnable runnable = **new** Runnable(){

**public** void run(){

System.out.println("Run");

}

}

You can easily run it in a new thread :

**new** Thread(runnable).start();

This is very simple and clean, but what if you've several long running tasks that you want to load in parralel and then wait for the completion of all the tasks, it's a little bit harder to code and if you want to get the return value of all the tasks it becomes really difficult to keep a good code. But like for almost any problems, Java has a solution for you, the Executors. This simple class allows you to create thread pools and thread factories.

A thread pool is represented by an instance of the class ExecutorService. With an ExecutorService, you can submit task that will be completed in the future. Here are the type of thread pools you can create with the Executors class :

* **Single Thread Executor** : A thread pool with only one thread. So all the submitted task will be executed sequentially. Method :*Executors.newSingleThreadExecutor()*
* **Cached Thread Pool** : A thread pool that create as many threads it needs to execute the task in parralel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool. Method :*Executors.newCachedThreadPool()*
* **Fixed Thread Pool** : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends. Method : *Executors.newFixedThreadPool()*
* **Scheduled Thread Pool** : A thread pool made to schedule future task. Method : *Executors.newScheduledThreadPool()*
* **Single Thread Scheduled Pool** : A thread pool with only one thread to schedule future task. Method :*Executors.newSingleThreadScheduledExecutor()*

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

Once you have a thread pool, you can submit task to it using the different submit methods. You can submit a Runnable or a Callable to the thread pool. The method return a Future representing the future state of the task. If you submitted a Runnable, the Future object return null once the task finished.

By example, if you have this Callable :

**private** **final** **class** **StringTask** **implements** Callable<String> {

**public** String call(){

*//Long operations*

**return** "Run";

}

}

If you want to execute that task 10 times using 4 threads, you can use that code :

ExecutorService pool = Executors.newFixedThreadPool(4);

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

pool.submit(**new** StringTask());

}

But you must shutdown the thread pool in order to terminate all the threads of the pool :

pool.shutdown();

If you don't do that, the JVM risk to not shutdown because there is still threads not terminated. You can also force the shutdown of the pool using shutdownNow, with that the currently running tasks will be interrupted and the tasks not started will not be started at all.

But with that example, you cannot get the result of the task. So let's get the Future objects of the tasks :

ExecutorService pool = Executors.newFixedThreadPool(4);

List<Future<String>> futures = **new** ArrayList<Future<String>>(10);

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

futures.add(pool.submit(**new** StringTask()));

}

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

String result = future.get();

*//Compute the result*

}

pool.shutdown();

But this code is a bit complicated. And there is a disadvantage. If the first task takes a long time to compute and all the other tasks ends before the first, the current thread cannot compute the result before the first task ends. Once again, Java has the solution for you, CompletionService.

A CompletionService is a service that make easier to wait for result of submitted task to an executor. The implementation is ExecutorCompletionService who's based on an ExecutorService to work. So let's try :

ExecutorService threadPool = Executors.newFixedThreadPool(4);

CompletionService<String> pool = **new** ExecutorCompletionService<String>(threadPool);

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

pool.submit(**new** StringTask());

}

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

String result = pool.take().get();

*//Compute the result*

}

threadPool.shutdown();

As soon as a task is completed, it is put in an internal java.util.concurrent.BlockingQueue (a highly efficient queue for Producer/Consumer problems and communication between threads).

From that queue, you can get the results of the finished tasks with take. If no task is yet available, take will wait until something is available.  
In this case we just print the result (the name of the current threat executing the Callable).

This is all you need to know to use a CompletionService. It is really simple. There is a lot of cool stuff in the JDK and in the java.util.concurrent package. Make sure to browse through the docs from time to time before inventing your own solution.

With that, you have the result in the order they are completed and you don't have to keep a collection of Future.

Here we are, you have the tools in hand to launch tasks in parralel using performing thread pools. Using Executors, ExecutorService and CompletionService you can create complex algorithm using several taks. With that tools, it's really easy to change the number of threads performing in parralel or adding more tasks without changing a lot of code.

I hope that this post will help you to write better concurrent code.

\*\*\*\*\*\*------IMP

This is where **ExecutorCompletionService** steps in. It is a thin wrapper around [ExecutorService](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) that "remembers" all submitted tasks and allows you to wait for the first *completed*, as opposed to first *submitted* task. In a way ExecutorCompletionService keeps a handle to all intermediate Future objects and once any of them finishes, it's returned. Crucial API method is [CompletionService.take()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CompletionService.html" \l "take()) that blocks and waits for *any* underlying Future to complete. Here is the submit step with ExecutorCompletionService:

## Task Delegation

Here is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:

|  |
| --- |
| A thread delegating a task to an ExecutorService for asynchronous execution. |
| **A thread delegating a task to an ExecutorService for asynchronous execution.** |

Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

## ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. TheExecutorService has the following implementation in the java.util.concurrent package:

* [**ThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)
* [**ScheduledThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

## Creating an ExecutorService

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

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(...)

I will take a look at each of these methods in the following sections.

### execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

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 aCallable for that (explained in the following sections).

### submit(Runnable)

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

Here is a ExecutorService 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.

### 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 an ExecutorService Callable 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:

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";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

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.

### 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 aFuture 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();

## ExecutorService Shutdown

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

For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

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

# Java Fork and Join using ForkJoinPool

Table of Contents

* [Fork and Join Explained](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#fork-and-join-explained)
  + [Fork](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#fork)
  + [Join](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#join)
* [The ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#the-forkjoinpool)
  + [Creating a ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#creating-a-forkjoinpool)
  + [Submitting Tasks to the ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#submittting-tasks-to-forkjoinpool)
* [RecursiveAction](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#recursive-action)
* [RecursiveTask](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#recursive-task)
* [ForkJoinPool Critique](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#forkjoinpool-critique)

Last updated: 2015-02-03

The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the [**Java ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html)but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Tasks can keep splitting their work into smaller subtasks for as long as it makes to split up the task. It may sound a bit abstract, so in this fork and join tutorial I will explain how the ForkJoinPool works, and how splitting tasks up work.

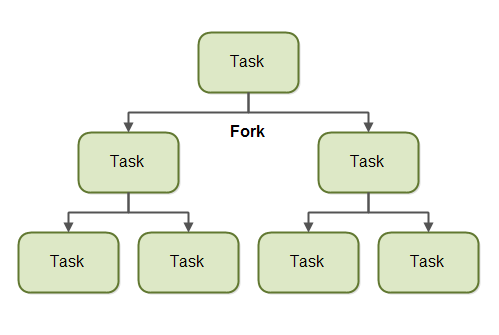
## Fork and Join Explained

Before we look at the ForkJoinPool I want to explain how the fork and join principle works in general.

The fork and join principle consists of two steps which are performed recursively. These two steps are the fork step and the join step.

### Fork

A task that uses the fork and join principle can *fork* (split) itself into smaller subtasks which can be executed concurrently. This is illustrated in the diagram below:



By splitting itself up into subtasks, each subtask can be executed in parallel by different CPUs, or different threads on the same CPU.

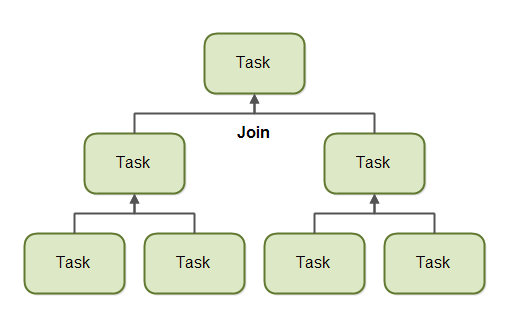
A task only splits itself up into subtasks if the work the task was given is large enough for this to make sense. There is an overhead to splitting up a task into subtasks, so for small amounts of work this overhead may be greater than the speedup achieved by executing subtasks concurrently.

The limit for when it makes sense to fork a task into subtasks is also called a threshold. It is up to each task to decide on a sensible threshold. It depends very much on the kind of work being done.

### Join

When a task has split itself up into subtasks, the task waits until the subtasks have finished executing.

Once the subtasks have finished executing, the task may *join* (merge) all the results into one result. This is illustrated in the diagram below:



Of course, not all types of tasks may return a result. If the tasks do not return a result then a task just waits for its subtasks to complete. No result merging takes place then.

## The ForkJoinPool

The ForkJoinPool is a special thread pool which is designed to work well with fork-and-join task splitting. TheForkJoinPool located in the java.util.concurrent package, so the full class name isjava.util.concurrent.ForkJoinPool.

### Creating a ForkJoinPool

You create a ForkJoinPool using its constructor. As a parameter to the ForkJoinPool constructor you pass the indicated level of parallelism you desire. The parallelism level indicates how many threads or CPUs you want to work concurrently on on tasks passed to the ForkJoinPool. Here is a ForkJoinPool creation example:

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

This example creates a ForkJoinPool with a parallelism level of 4.

### Submitting Tasks to the ForkJoinPool

You submit tasks to a ForkJoinPool similarly to how you submit tasks to an ExecutorService. You can submit two types of tasks. A task that does not return any result (an "action"), and a task which does return a result (a "task"). These two types of tasks are represented by the RecursiveAction and RecursiveTaskclasses. How to use both of these tasks and how to submit them will be covered in the following sections.

## RecursiveAction

A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.

A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.

You implement a RecursiveAction by subclassing it. Here is a RecursiveAction example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

private long workLoad = 0;

public MyRecursiveAction(long workLoad) {

this.workLoad = workLoad;

}

@Override

protected void compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

subtasks.addAll(createSubtasks());

for(RecursiveAction subtask : subtasks){

subtask.fork();

}

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

}

}

private List<MyRecursiveAction> createSubtasks() {

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);

MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is very simplified. The MyRecursiveAction simply takes a fictive workLoad as parameter to its constructor. If the workLoad is above a certain threshold, the work is split into subtasks which are also scheduled for execution (via the .fork() method of the subtasks. If the workLoad is below a certain threshold then the work is carried out by the MyRecursiveAction itself.

You can schedule a MyRecursiveAction for execution like this:

MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

forkJoinPool.invoke(myRecursiveAction);

## RecursiveTask

A RecursiveTask is a task that returns a result. It may split its work up into smaller tasks, and merge the result of these smaller tasks into a collective result. The splitting and merging may take place on several levels. Here is a RecursiveTask example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Long> {

private long workLoad = 0;

public MyRecursiveTask(long workLoad) {

this.workLoad = workLoad;

}

protected Long compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

subtasks.addAll(createSubtasks());

for(MyRecursiveTask subtask : subtasks){

subtask.fork();

}

long result = 0;

for(MyRecursiveTask subtask : subtasks) {

result += subtask.join();

}

return result;

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

return workLoad \* 3;

}

}

private List<MyRecursiveTask> createSubtasks() {

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);

MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is similar to the RecursiveAction example except it returns a result. The classMyRecursiveTask extends RecursiveTask<Long> which means that the result returned from the task is aLong .

The MyRecursiveTask example also breaks the work down into subtasks, and schedules these subtasks for execution using their fork() method.

Additionally, this example then receives the result returned by each subtask by calling the join() method of each subtask. The subtask results are merged into a bigger result which is then returned. This kind of joining / mergining of subtask results may occur recursively for several levels of recursion.

You can schedule a RecursiveTask like this:

MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);

Notice how you get the final result out from the ForkJoinPool.invoke() method call.

## ForkJoinPool Critique

It seems not everyone is equally happy with the new ForkJoinPool in Java 7. While searching for experiences with, and opinions about, the ForkJoinPool, I came across the following critique:

[**A Java Fork-Join Calamity**](http://coopsoft.com/ar/CalamityArticle.html)

It is well worth a read before you plan to use the ForkJoinPool in your own projects.

Java 8 introduced a lot of cool features, whereas lambdas and streams caught much of the attention.

What you may have missed is the CompletableFuture.

### You probably already know about Futures

A Future represents the pending result of an asynchronous computation. It offers a method — get — that returns the result of the computation when it's done.

The problem is that a call to get is blocking until the computation is done. This is quite restrictive and can quickly make the asynchronous computation pointless.

Sure — you can keep coding all scenarios into the job you're sending to the executor, but why should you have to worry about all the plumbing around the logic you really care about?

### This is where CompletableFuture saves the day

Beside implementing the Future interface, CompletableFuture also implements the CompletionStage interface.

A CompletionStage is a promise. It promises that the computation eventually will be done.

The great thing about the CompletionStage is that it offers a vast selection of methods that let you attach callbacks that will be executed on completion.

This way we can build systems in a non-blocking fashion.

Ok, enough chatting, let's start coding!

### The simplest asynchronous computation

Let's start with the absolute basics — creating a simple asynchronous computation.

CompletableFuture.supplyAsync(this::sendMsg);

It's as easy as that.

supplyAsync takes a Supplier containing the code we want to execute asynchronously — in our case the sendMsg method.

If you've worked a bit with Futures in the past, you may wonder where the Executor went. If you want to, you can still define it as a second argument. However, if you leave it out it will be submitted to the ForkJoinPool.commonPool().

# Java 8 CompletableFuture Tutorial

[Rajeev Kumar Singh](https://www.callicoder.com/about)• [Java](https://www.callicoder.com/category/java)• Jul 18, 2017 • 18 mins



Java 8 came up with tons of new features and enhancements like [Lambda expressions](https://www.callicoder.com/java-lambda-expression-tutorial/), [Streams](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html), [CompletableFutures](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) etc. In this post I’ll give you a detailed explanation of CompletableFuture and all its methods using simple examples.

## What’s a CompletableFuture?

CompletableFuture is used for asynchronous programming in Java. Asynchronous programming is a means of writing non-blocking code by running a task on a separate thread than the main application thread and notifying the main thread about its progress, completion or failure.

This way, your main thread does not block/wait for the completion of the task and it can execute other tasks in parallel.

Having this kind of parallelism greatly improves the performance of your programs.

## Future vs CompletableFuture

CompletableFuture is an extension to [Java’s Future API](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) which was introduced in Java 5.

A Future is used as a reference to the result of an asynchronous computation. It provides an isDone()method to check whether the computation is done or not, and a get() method to retrieve the result of the computation when it is done.

You can learn more about Future from my [Callable and Future Tutorial](https://www.callicoder.com/java-callable-and-future-tutorial/).

Future API was a good step towards asynchronous programming in Java but it lacked some important and useful features -

### Limitations of Future

1. **It cannot be manually completed :**

Let’s say that you’ve written a function to fetch the latest price of an e-commerce product from a remote API. Since this API call is time-consuming, you’re running it in a separate thread and returning a Future from your function.

Now, let’s say that If the remote API service is down, then you want to complete the Future manually by the last cached price of the product.

Can you do this with Future? No!

1. **You cannot perform further action on a Future’s result without blocking:**

Future does not notify you of its completion. It provides a get() method which **blocks** until the result is available.

You don’t have the ability to attach a callback function to the Future and have it get called automatically when the Future’s result is available.

1. **Multiple Futures cannot be chained together :**

Sometimes you need to execute a long-running computation and when the computation is done, you need to send its result to another long-running computation, and so on.

You can not create such asynchronous workflow with Futures.

1. **You can not combine multiple Futures together :**

Let’s say that you have 10 different Futures that you want to run in parallel and then run some function after all of them completes. You can’t do this as well with Future.

1. **No Exception Handling :**

Future API does not have any exception handling construct.

Whoa! So many limitations right? Well, That’s why we have CompletableFuture. You can achieve all of the above with CompletableFuture.

CompletableFuture implements Future and CompletionStage interfaces and provides a huge set of convenience methods for creating, chaining and combining multiple Futures. It also has a very comprehensive exception handling support.

## Creating a CompletableFuture

### 1. The trivial example -

You can create a CompletableFuture simply by using the following no-arg constructor -

CompletableFuture<String> completableFuture = new CompletableFuture<String>();

This is the simplest CompletableFuture that you can have. All the clients who want to get the result of this CompletableFuture can call CompletableFuture.get() method -

String result = completableFuture.get()

The get() method blocks until the Future is complete. So, the above call will block forever because the Future is never completed.

You can use CompletableFuture.complete() method to manually complete a Future -

completableFuture.complete("Future's Result")

All the clients waiting for this Future will get the specified result. And, Subsequent calls to completableFuture.complete() will be ignored.

### 2. Running asynchronous computation using runAsync() -

If you want to run some background task asynchronously and don’t want to return anything from the task, then you can use CompletableFuture.runAsync() method. It takes a [Runnable](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) object and returns CompletableFuture<Void>.

// Run a task specified by a Runnable Object asynchronously.

CompletableFuture<Void> future = CompletableFuture.runAsync(new Runnable() {

@Override

public void run() {

// Simulate a long-running Job

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("I'll run in a separate thread than the main thread.");

}

});

// Block and wait for the future to complete

future.get()

You can also pass the Runnable object in the form of a [lambda expression](https://www.callicoder.com/java-lambda-expression-tutorial/) -

// Using Lambda Expression

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

// Simulate a long-running Job

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

System.out.println("I'll run in a separate thread than the main thread.");

});

In this post, I’ll use lambda expressions very frequently, and you should use it too if you’re not already using it in your Java code.

### 3. Run a task asynchronously and return the result using supplyAsync() -

CompletableFuture.runAsync() is useful for tasks that don’t return anything. But what if you want to return some result from your background task?

Well, CompletableFuture.supplyAsync() is your companion. It takes a [Supplier<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) and returns CompletableFuture<T> where T is the type of the value obtained by calling the given supplier -

// Run a task specified by a Supplier object asynchronously

CompletableFuture<String> future = CompletableFuture.supplyAsync(new Supplier<String>() {

@Override

public String get() {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Result of the asynchronous computation";

}

});

// Block and get the result of the Future

String result = future.get();

System.out.println(result);

A [Supplier<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) is a simple functional interface which represents a supplier of results. It has a single get() method where you can write your background task and return the result.

Once again, you can use Java 8’s lambda expression to make the above code more concise -

// Using Lambda Expression

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Result of the asynchronous computation";

});

#### **A note about Executor and Thread Pool -**

You might be wondering that - Well, I know that the runAsync() and supplyAsync() methods execute their tasks in a separate thread. But, we never created a thread right?

Yes! CompletableFuture executes these tasks in a thread obtained from the global [ForkJoinPool.commonPool()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html" \l "commonPool--).

But hey, you can also create a Thread Pool and pass it to runAsync() and supplyAsync() methods to let them execute their tasks in a thread obtained from your thread pool.

All the methods in the CompletableFuture API has two variants - One which accepts an [Executor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html) as an argument and one which doesn’t -

// Variations of runAsync() and supplyAsync() methods

static CompletableFuture<Void> runAsync(Runnable runnable)

static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)

Here’s how you can create a thread pool and pass it to one of these methods -

Executor executor = Executors.newFixedThreadPool(10);

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Result of the asynchronous computation";

}, executor);

## Transforming and acting on a CompletableFuture

The CompletableFuture.get() method is blocking. It waits until the Future is completed and returns the result after its completion.

But, that’s not what we want right? For building asynchronous systems we should be able to attach a callback to the CompletableFuture which should automatically get called when the Future completes.

That way, we won’t need to wait for the result, and we can write the logic that needs to be executed after the completion of the Future inside our callback function.

You can attach a callback to the CompletableFuture using thenApply(), thenAccept() and thenRun()methods -

### 1. thenApply()

You can use thenApply() method to process and transform the result of a CompletableFuture when it arrives. It takes a [Function<R,T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) as an argument. [Function<R,T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) is a simple functional interface representing a function that accepts an argument of type T and produces a result of type R -

// Create a CompletableFuture

CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Rajeev";

});

// Attach a callback to the Future using thenApply()

CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(name -> {

return "Hello " + name;

});

// Block and get the result of the future.

System.out.println(greetingFuture.get()); // Hello Rajeev

You can also write a **sequence of transformations** on the CompletableFuture by attaching a series of thenApply() callback methods. The result of one thenApply() method is passed to the next in the series -

CompletableFuture<String> welcomeText = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Rajeev";

}).thenApply(name -> {

return "Hello " + name;

}).thenApply(greeting -> {

return greeting + ", Welcome to the CalliCoder Blog";

});

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

// Prints - Hello Rajeev, Welcome to the CalliCoder Blog

### 2. thenAccept() and thenRun()

If you don’t want to return anything from your callback function and just want to run some piece of code after the completion of the Future, then you can use thenAccept() and thenRun() methods. These methods are consumers and are often used as the last callback in the callback chain.

CompletableFuture.thenAccept() takes a [Consumer<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) and returns CompletableFuture<Void>. It has access to the result of the CompletableFuture on which it is attached.

// thenAccept() example

CompletableFuture.supplyAsync(() -> {

return ProductService.getProductDetail(productId);

}).thenAccept(product -> {

System.out.println("Got product detail from remote service " + product.getName())

});

While thenAccept() has access to the result of the CompletableFuture on which it is attached, thenRun() doesn’t even have access to the Future’s result. It takes a Runnable and returns CompletableFuture<Void> -

// thenRun() example

CompletableFuture.supplyAsync(() -> {

// Run some computation

}).thenRun(() -> {

// Computation Finished.

});

#### A note about async callback methods -

All the callback methods provided by CompletableFuture have two async variants -

// thenApply() variants

<U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)

<U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn)

<U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)

These async callback variations help you further parallelize your computations by executing the callback tasks in a separate thread.

Consider the following example -

CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Some Result"

}).thenApply(result -> {

/\*

Executed in the same thread where the supplyAsync() task is executed

or in the main thread If the supplyAsync() task completes immediately (Remove sleep() call to verify)

\*/

return "Processed Result"

})

In the above case, the task inside thenApply() is executed in the same thread where the supplyAsync() task is executed, or in the main thread if the supplyAsync() task completes immediately (try removing sleep() call to verify).

To have more control over the thread that executes the callback task, you can use async callbacks. If you use thenApplyAsync() callback, then it will be executed in a different thread obtained from ForkJoinPool.commonPool() -

CompletableFuture.supplyAsync(() -> {

return "Some Result"

}).thenApplyAsync(result -> {

// Executed in a different thread from ForkJoinPool.commonPool()

return "Processed Result"

})

Moreover, If you pass an Executor to the thenApplyAsync() callback then the task will be executed in a thread obtained from the Executor’s thread pool.

Executor executor = Executors.newFixedThreadPool(2);

CompletableFuture.supplyAsync(() -> {

return "Some result"

}).thenApplyAsync(result -> {

// Executed in a thread obtained from the executor

return "Processed Result"

}, executor);

## Combining two CompletableFutures together

### 1. Combine two dependent futures using thenCompose() -

Let’s say that you want to fetch the details of a user from a remote API service and once the user’s detail is available, you want to fetch his Credit rating from another service.

Consider the following implementations of getUserDetail() and getCreditRating() methods -

CompletableFuture<User> getUsersDetail(String userId) {

return CompletableFuture.supplyAsync(() -> {

UserService.getUserDetails(userId);

});

}

CompletableFuture<Double> getCreditRating(User user) {

return CompletableFuture.supplyAsync(() -> {

CreditRatingService.getCreditRating(user);

});

}

Now, Let’s understand what will happen if we use thenApply() to achieve the desired result -

CompletableFuture<CompletableFuture<Double>> result = getUserDetail(userId)

.thenApply(user -> getCreditRating(user));

In earlier examples, the Supplier function passed to thenApply() callback would return a simple value but in this case, it is returning a CompletableFuture. Therefore, the final result in the above case is a nested CompletableFuture.

If you want the final result to be a top-level Future, use thenCompose() method instead -

CompletableFuture<Double> result = getUserDetail(userId)

.thenCompose(user -> getCreditRating(user));

So, Rule of thumb here - If your callback function returns a CompletableFuture, and you want a flattened result from the CompletableFuture chain (which in most cases you would), then use *thenCompose()*.

### 2. Combine two independent futures using thenCombine() -

While thenCompose() is used to combine two Futures where one future is dependent on the other, thenCombine() is used when you want two Futures to run independently and do something after both are complete.

System.out.println("Retrieving weight.");

CompletableFuture<Double> weightInKgFuture = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return 65.0;

});

System.out.println("Retrieving height.");

CompletableFuture<Double> heightInCmFuture = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return 177.8;

});

System.out.println("Calculating BMI.");

CompletableFuture<Double> combinedFuture = weightInKgFuture

.thenCombine(heightInCmFuture, (weightInKg, heightInCm) -> {

Double heightInMeter = heightInCm/100;

return weightInKg/(heightInMeter\*heightInMeter);

});

System.out.println("Your BMI is - " + combinedFuture.get());

The callback function passed to thenCombine() will be called when both the Futures are complete.

## Combining multiple CompletableFutures together

We used thenCompose() and thenCombine() to combine two CompletableFutures together. Now, what if you want to combine an arbitrary number of CompletableFutures? Well, you can use the following methods to combine any number of CompletableFutures -

static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)

static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

### 1. CompletableFuture.allOf()

CompletableFuture.allOf is used in scenarios when you have a List of independent futures that you want to run in parallel and do something after all of them are complete.

Let’s say that you want to download the contents of 100 different web pages of a website. You can do this operation sequentially but this will take a lot of time. So, you have written a function which takes a web page link, and returns a CompletableFuture, i.e. It downloads the web page’s content asynchronously -

CompletableFuture<String> downloadWebPage(String pageLink) {

return CompletableFuture.supplyAsync(() -> {

// Code to download and return the web page's content

});

}

Now, when all the web pages are downloaded, you want to count the number of web pages that contain a keyword - ‘CompletableFuture’. Let’s use CompletableFuture.allOf() to achieve this -

List<String> webPageLinks = Arrays.asList(...) // A list of 100 web page links

// Download contents of all the web pages asynchronously

List<CompletableFuture<String>> pageContentFutures = webPageLinks.stream()

.map(webPageLink -> downloadWebPage(webPageLink))

.collect(Collectors.toList());

// Create a combined Future using allOf()

CompletableFuture<Void> allFutures = CompletableFuture.allOf(

pageContentFutures.toArray(new CompletableFuture[pageContentFutures.size()])

);

The problem with CompletableFuture.allOf() is that it returns CompletableFuture<Void>. But we can get the results of all the wrapped CompletableFutures by writing few additional lines of code -

// When all the Futures are completed, call `future.join()` to get their results and collect the results in a list -

CompletableFuture<List<String>> allPageContentsFuture = allFutures.thenApply(v -> {

return pageContentFutures.stream()

.map(pageContentFuture -> pageContentFuture.join())

.collect(Collectors.toList());

});

Take a moment to understand the above code snippet. Since we’re calling future.join() when all the futures are complete, we’re not blocking anywhere :-)

The join() method is similar to get(). The only difference is that it throws an unchecked exception if the underlying CompletableFuture completes exceptionally.

Let’s now count the number of web pages that contain our keyword -

// Count the number of web pages having the "CompletableFuture" keyword.

CompletableFuture<Long> countFuture = allPageContentsFuture.thenApply(pageContents -> {

return pageContents.stream()

.filter(pageContent -> pageContent.contains("CompletableFuture"))

.count();

});

System.out.println("Number of Web Pages having CompletableFuture keyword - " +

countFuture.get());

### 2. CompletableFuture.anyOf()

CompletableFuture.anyOf() as the name suggests, returns a new CompletableFuture which is completed when any of the given CompletableFutures complete, with the same result.

Consider the following example -

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Result of Future 1";

});

CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Result of Future 2";

});

CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> {

try {

TimeUnit.SECONDS.sleep(3);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

return "Result of Future 3";

});

CompletableFuture<Object> anyOfFuture = CompletableFuture.anyOf(future1, future2, future3);

System.out.println(anyOfFuture.get()); // Result of Future 2

In the above example, the anyOfFuture is completed when any of the three CompletableFutures complete. Since future2 has the least amount of sleep time, it will complete first, and the final result will be - Result of Future 2.

CompletableFuture.anyOf() takes a varargs of Futures and returns CompletableFuture<Object>. The problem with CompletableFuture.anyOf() is that if you have CompletableFutures that return results of different types, then you won’t know the type of your final CompletableFuture.

## CompletableFuture Exception Handling

We explored How to create CompletableFuture, transform them, and combine multiple CompletableFutures. Now let’s understand what to do when anything goes wrong.

Let’s first understand how errors are propagated in a callback chain. Consider the following CompletableFuture callback chain -

CompletableFuture.supplyAsync(() -> {

// Code which might throw an exception

return "Some result";

}).thenApply(result -> {

return "processed result";

}).thenApply(result -> {

return "result after further processing";

}).thenAccept(result -> {

// do something with the final result

});

If an error occurs in the original supplyAsync() task, then none of the thenApply() callbacks will be called and future will be resolved with the exception occurred. If an error occurs in first thenApply()callback then 2nd and 3rd callbacks won’t be called and the future will be resolved with the exception occurred, and so on.

### 1. Handle exceptions using exceptionally() callback

The exceptionally() callback gives you a chance to recover from errors generated from the original Future. You can log the exception here and return a default value.

Integer age = -1;

CompletableFuture<String> maturityFuture = CompletableFuture.supplyAsync(() -> {

if(age < 0) {

throw new IllegalArgumentException("Age can not be negative");

}

if(age > 18) {

return "Adult";

} else {

return "Child";

}

}).exceptionally(ex -> {

System.out.println("Oops! We have an exception - " + ex.getMessage());

return "Unknown!";

});

System.out.println("Maturity : " + maturityFuture.get());

Note that, the error will not be propagated further in the callback chain if you handle it once.

### 2. Handle exceptions using the generic handle() method

The API also provides a more generic method - handle() to recover from exceptions. It is called whether or not an exception occurs.

Integer age = -1;

CompletableFuture<String> maturityFuture = CompletableFuture.supplyAsync(() -> {

if(age < 0) {

throw new IllegalArgumentException("Age can not be negative");

}

if(age > 18) {

return "Adult";

} else {

return "Child";

}

}).handle((res, ex) -> {

if(ex != null) {

System.out.println("Oops! We have an exception - " + ex.getMessage());

return "Unknown!";

}

return res;

});

System.out.println("Maturity : " + maturityFuture.get());

If an exception occurs, then the res argument will be null, otherwise, the ex argument will be null.

## Conclusion

**Congratulations folks!** In this tutorial, we explored the most useful and important concepts of CompletableFuture API.

Thank you for reading. I hope this blog post was helpful to you. Let me know your views, questions, comments in the comment section below.