**CompletableFute Methods – Usage – Precise Organized – 2022**

**Difference between supplyAsync() and runAsync()**

[**runAsync**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#runAsync-java.lang.Runnable-)**()** 🡺 **Runnable** as input parameter, output as **CompletableFuture<Void>**. Returns nothing

Runnable r = () -> *m1*();  
CompletableFuture<Void> cf = CompletableFuture.*runAsync*(r);  
cf.join();

**CompletableFuture.*runAsync*(() -> *m1*()).join(); 🡸 Recommended to use like this**

[**suppyAsync**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#supplyAsync-java.util.function.Supplier-)**()** 🡺 Supplier as argument, output as **CompletableFuture<U>** with result value, which means it does not take any input parameters but it returns result as output.

Runnable r = () -> *m11*("abcd", "pqrs");  
String value = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs")).join();  
System.*out*.println(value);

You can not write like below, it give compilation issue.

~~Runnable r = () ->~~ *~~m11~~*~~("abcd", "pqrs");  
String value = CompletableFuture.~~*~~supplyAsync~~*~~(r).join();~~

Simply execute 🡺 runAsync()

Returns Result 🡺 supplyAsync()

# What is the difference between get() and join() in CompletableFutre ?

V get() 🡺 throws InterruptedException, ExecutionException.

join() 🡺 method doesn't throw checked exceptions.

**Cheat Sheet**

The computation performed by a stage may be expressed as a Function, Consumer, or Runnable (using methods with names including **apply, accept, or run**, respectively)

**stage.thenApply(x -> square(x)) .thenAccept(x -> System.out.print(x)).thenRun(() -> System.out.println());**

**Accept: thenAccept(), acceptEither(), thenAccceptBoth() 🡺 returns nothing**

**Apply: thenApply(), applyToEither() 🡺 It returns**

**Run: thenRun()**, **runAfterEither()**, **runAfterBoth() 🡺 returns nothing**

**Accept: thenAccept(), acceptEither(), thenAccceptBoth() 🡺 returns nothing**

**thenAccept()**

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));  
**cf1.thenAccept(result -> *m1*(result)).join();**

**acceptEither()**

Between two tasks, which completes first. This is used when you want either of the tasks which completes the earliest. Anyone who gives money quickly, I will cast my vote.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));  
CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m22*("asdf"));  
cf1.acceptEither(cf2, (result) -> *m3*(result)).join();

**thenAccceptBoth()**

This method is useful when you perform two parallel tasks and take both the results for processing.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));  
CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m22*("asdf"));  
cf1.thenAcceptBoth(cf2, (result1, result2) -> *m4*(result1, result2)).join();

**Run: thenRun()**, **runAfterEither()**, **runAfterBoth() 🡺 returns nothing**

**thenRun**

This method is useful when you perform an another task asynchronously after completion of main task.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));

cf1.thenRun( () -> m5()).join();

**runAfterEither()**

Which completes first and then run another function to perform.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));  
CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m22*("asdf"));  
cf1.runAfterEither(cf2, () -> *m4*(null, null)).join();

**runAfterBoth()**

This method is useful when you want to perform a third task after completion two parallel tasks **without any responses**.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));  
CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m22*("asdf"));  
cf1.runAfterBoth(cf2, () -> *m4*(null, null)).join();

**apply: thenApply(), applyToEither() 🡺 It returns**

**thenApply()**

**In case of .thenApply() method, result is derived from the previous thread execution and added/manipulated result with manipulation will be returned**.

CompletableFuture<String> cf = cf1.thenApply( (result) -> getVal(result));

String response = cf.join();

**applyToEither()**

CompletableFuture<String> cf = cf1.applyToEither(cf2, (result) -> getVal(result));

String response = cf.join();

**thenCombine() 🡺 returns**

.thenCombine() is used to combine the results of two independent CompletableFuture stages.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));  
CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m22*("asdf"));  
String response = cf1.thenCombine(cf2, (result1,result2) -> *m4*(result1, result2)).join();

**thenCompose() 🡺returns**

**.**thenCompose is used for chaining dependent asynchronous computations.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m11*("abcd", "pqrs"));  
String response = cf1.thenCompose((result1) -> CompletableFuture.*supplyAsync*( () -> *m3*(result1))).join();

**anyOf()**

CompletableFuture<Object> cf = CompletableFuture.anyOf(cf1, cf2, cf3, cf4);

Object response = cf.join();

System.out.println("Who completed first : "+response);

**allOf()**

CompletableFuture<Void> cf = CompletableFuture.allOf(cf1, cf2, cf3, cf4);

cf.join();

String result1 = cf1.join();

String result2 = cf2.join();

Integer result4 = cf4.join();

**handle() 🡺 returns**

public static String task2(String input) {  
 try {  
 System.*out*.println("Executing Task 2 ...");  
 TimeUnit.*SECONDS*.sleep(3);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 if(input == null) throw new IllegalArgumentException("Invalid value");  
 return "Task-2";  
}

public static String getDefaultValue(String result, Throwable error) {  
 if(error != null)  
 return "default value";  
 else  
 return result;  
}

public static void check1() {  
 CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task2*("some value"));  
 String response = cf1.handle((result, error) -> *getDefaultValue*(result,error)).join();  
 System.*out*.println("Result : "+response);  
}

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task2*("some value"));

The result will be Result : Task-2

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task2*(null));

The result will be Result : default value

**whenComplete() 🡺 returns**

**This method is useful when you want to perform certain intermediate operation before returning the value.** As per the above methods, it will be written like this.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task2*(null));  
String response = cf1.whenComplete((result, error) -> *getDefaultValue*(result,error)).join();  
System.*out*.println("Result : "+response);

**exceptionally()**

CompletableFuture<String> cf = cf1.exceptionally( ex -> {

System.out.println("Exception Details: "+ex.getMessage());

return "Good Result";

});

String result = cf.join();

**exceptionally()** is used to get the fallback where as **handle()** is used to take some necessary action. Both return the values.

**Difference Between thenApply() and thenCompose()**

We can use **thenApply()** method to work with a result of the previous call. However, a key point to remember is that the return type will be combined of all calls. So this method is useful when we want to transform the result of a CompletableFuture call.

The thenCompose() method is similar to thenApply() in that both return a new Completion Stage. However, **thenCompose() uses the previous stage as the argument**. It will flatten and return a Future with the result directly, rather than a nested future as we observed in thenApply()

**thenApply** is used if you have a synchronous mapping function.

String result = cf1.thenApply(r1 -> m4(r1)).join();  
System.*out*.println("Final Result: "+result);

It is like this get the result and apply anywhere.

**thenCompose** is used if you have an asynchronous mapping function (i.e. one that returns a CompletableFuture). It will then return a future with the result directly, rather than a nested future.

CompletableFuture<Integer> future =

CompletableFuture.supplyAsync(() -> 1)

.thenCompose(x -> CompletableFuture.supplyAsync(() -> x+1));