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

**Accept, Run, Apply, Combine, Compose, anyOf, allOf, handle(), exceptionally**

**Accept, Apply , Run, Combine, Compose, anyOf, allOf, handle(), exceptionally**

Category of **Accept: thenAccept(), acceptEither(), thenAcceptBoth()**

**acceptEither (CompletionStage<? extends T> other, Consumer<? super T> action)**

🡺 CompletableFuture<Void>

**public** **static** String m1() {

**try** {

System.***out***.println("Collecting Aadhar Information ...");

TimeUnit.***SECONDS***.sleep(7);

} **catch** (InterruptedException ie) {

ie.printStackTrace();

}

**return** "m1";

}

**public** **static** String m2() {

**try** {

System.***out***.println("Collecting Pan Information ...");

TimeUnit.***SECONDS***.sleep(5);

} **catch** (InterruptedException ie) {

ie.printStackTrace();

}

**return** "m2";

}

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m2*());

CompletableFuture<Void> cf = cf1.acceptEither(cf2, (result) -> {

System.***out***.println("Result: "+ result);

});

cf.join();

}

It can be written like this using anyOf()

CompletableFuture<Object> cf = CompletableFuture.*anyOf*(cf1, cf2);  
Object response = cf.join();  
System.*out*.println("Result : "+response);

**public** **static** **void** main(String[] args) {

*check*();

}

**thenAcceptBoth (CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action) 🡺 CompletableFuture<Void>**

This method is useful when you perform two parallel tasks and take both the results for processing. It is of Void type, it does not return anything.

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m2*());

CompletableFuture<Void> cf = cf1.thenAcceptBoth(cf2, (m1Result1, m2Result2) -> {

System.***out***.println("Result1: "+ m1Result1);

System.***out***.println("Result2: "+ m2Result2);

});

cf.join();

}

The above code can be written using allOf()

CompletableFuture<Void> cf = CompletableFuture.*allOf*(cf1, cf2);  
cf.join();  
String result1 = cf1.join();  
String result2 = cf2.join();  
  
System.*out*.println("Result1 : "+result1);  
System.*out*.println("Result2 : "+result2);

**thenAccept (Consumer<? super T> action) 🡺 CompletableFuture<Void>**

**It does not return any result**, simply executes a function. Get the response from a method and pass to another method which does not provide result. An example will be get Aadhar No by providing some input and then execute another method by passing Aadhar No and Address. The second method does not provide result.

**public** **static** String m4(String arg) {

**try** {

System.***out***.println("some ops");

TimeUnit.***SECONDS***.sleep(2);

} **catch** (InterruptedException ie) {

ie.printStackTrace();

}

**return** "m4";

}

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<Void> cf = cf1.thenAccept((result) -> *m4*(result));

cf.join();

}

The above can be written like this.

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());  
String result = cf1.join();  
*m4*(result);

However, the first method takes less time as compared to the above highlighted code.

Category of **Apply**: **thenApply(), applyToEither()**

**applyToEither (CompletionStage<? extends T> other, Function<? super T,U> fn)**

🡺 **CompletableFuture<U>**

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m2*());

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

**return** result;

});

String response = cf.join();

System.***out***.println("Response: " + response);

}

The above can be written like this.

CompletableFuture<Object> cf = CompletableFuture.*anyOf*(cf1,cf2);  
String response = (String) cf.join();  
System.*out*.println("Response: " + response);

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

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

**public static void** update() {  
 CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *getAadhar*(**"input"**));  
 **CompletableFuture<String> cf2 = cf1  
 .thenApply((aadharResult) -> *getPancard*(aadharResult))  
 .thenApply((panResult) -> *getPassport*(panResult));**  
 String result = **null**;  
 **try** {  
 **result = cf2.get();** System.***out***.println(**"Result : "** + result);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }  
}

Category of **Run: thenRun(), runAfterEither(), runAfterBoth()**

# thenRun(Runnable action) 🡺 CompletableFuture<Void>

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

Always remember, **.thenRun() does not return any value**. It is of type void. It will be useful if you do an audit operation when you want to store the values in database in each .thenRun().

**CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(() -> task1());**

**CompletableFuture<Void> cfVoid = cf1.thenRun(() -> task2());**

//If you do not write the following line ie. cfVoid.join(),

//task2() will not be executed completely, only one line will be executed.

**public** **static** **void** check() {

CompletableFuture<Void> cf = CompletableFuture.*runAsync*(() -> *m1*()).thenRun(() -> *m2*());

Void blank = cf.join();

}

**runAfterEither(CompletionStage<?> other, Runnable action) 🡺 CompletableFuture<Void>**

Which completes first and then run another function to perform.

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m2*());

CompletableFuture<Void> cf = cf1.runAfterEither(cf2, () -> *m3*());

cf.join();

}

It can be also written like this.

CompletableFuture.*anyOf*(cf1,cf2).thenRun(() -> *m3*()).join(); // Takes more time

# runAfterBoth (CompletionStage<?> other, Runnable action)🡺 CompletableFuture<Void>

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

**public** **static** String m3() {

**try** {

System.***out***.println("Collecting Passport Information ...");

TimeUnit.***SECONDS***.sleep(5);

} **catch** (InterruptedException ie) {

ie.printStackTrace();

}

**return** "m2";

}

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m2*());

CompletableFuture<Void> cf = cf1.runAfterBoth(cf2, () -> *m3*());

cf.join();

}

It can also be written like this.

CompletableFuture.*allOf*(cf1,cf2).thenRun(() -> *m3*()).join();

**thenCombine (CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn) 🡺 <U,V> CompletableFuture<V>**

**public** **static** String performThridTask(String s1, String s2) {

System.***out***.println("Running Thrid Operation ..");

**return** s1+"-combined-"+s2;

}

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *m2*());

CompletableFuture<String> cf = cf1.thenCombine(cf2, (m1Result, m2Result) -> {

**return** *performThridTask*(m1Result, m2Result);

});

String response = cf.join();

System.***out***.println("Response: "+response);

}

Alternative way of writing

CompletableFuture.*allOf*(cf1,cf2).join();  
String result1 = cf1.join();  
String result2 = cf2.join();  
String response = *performThridTask*(result1, result2);  
System.*out*.println(response);

**Multiple combine**

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task1*());  
CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *task2*(**"input1"**));  
**CompletableFuture<Void> cf01 = cf1.thenCombine(cf2, (r1, r2) -> *task3*(r1));  
CompletableFuture<Integer> cf02 = cf01.thenCombine(cf01, (r1, r2) -> *task4*(1));**  
Integer val = cf02.join();  
System.***out***.println(**"Value : "**+val);

**thenCompose()(Function<? super T,? extends CompletionStage<U>> fn)**

**🡺 CompletableFuture<U>**

**public** **static** String m4(String arg) {

**try** {

System.***out***.println("some ops");

TimeUnit.***SECONDS***.sleep(2);

} **catch** (InterruptedException ie) {

ie.printStackTrace();

}

**return** "m4";

}

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf = cf1.thenCompose((m1Result) -> CompletableFuture.*supplyAsync*(() -> *m4*(m1Result)));

String response = cf.join();

System.***out***.println("Response: "+response);

}

**anyOf(CompletableFuture<?>... cfs)** 🡺 [CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html)<[Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)>

**Which task completes first.**

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task1*());  
CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *task2*(**"input1"**));  
CompletableFuture<Void> cf3 = CompletableFuture.*supplyAsync*(() -> *task3*(**"input2"**));  
CompletableFuture<Integer> cf4 = CompletableFuture.*supplyAsync*(() -> *task4*(4));  
  
CompletableFuture<Object> cf = **CompletableFuture.*anyOf*(cf1, cf2, cf3, cf4);**  
Object response = cf.join();  
System.***out***.println(**"Who completed first : "**+response);

**allOf(CompletableFuture<?>... cfs)** 🡺**CompletableFuture<Void>**

**public static void** check() {  
 CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task1*());  
 CompletableFuture<String> cf2 = CompletableFuture.*supplyAsync*(() -> *task2*(**"input1"**));  
 CompletableFuture<Void> cf3 = CompletableFuture.*supplyAsync*(() -> *task3*(**"input2"**));  
 CompletableFuture<Integer> cf4 = CompletableFuture.*supplyAsync*(() -> *task4*(4));  
  
 **CompletableFuture<Void> cf = CompletableFuture.*allOf*(cf1, cf2, cf3, cf4);  
 cf.join();**  
  
 String result1 = cf1.join();  
 String result2 = cf2.join();  
 Integer result4 = cf4.join();  
 System.***out***.println(result1 + **" : "** + result2 + **" : "** + result4);}

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

**handle (BiFunction<? super T,Throwable,? extends U> fn) 🡺 CompletableFuture<U>**

**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 void** check() {  
 CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task2*(**null**));  
 CompletableFuture<String> cf = cf1.handle((result, error) -> {  
 System.***out***.println(**"Result : "**+result);  
 System.***out***.println(**"Error : "**+error);  
 **return "some business result"**;  
 });  
 String result = cf.join();  
 System.***out***.println(**"Result : "**+result);  
}

**OUTPUT**

Executing Task 2 ...

Result : null

Error : java.util.concurrent.CompletionException: java.lang.IllegalArgumentException: Invalid value

Result : some business result

**exceptionally(Function<Throwable,? extends T> fn) 🡺 CompletableFuture<T>**

**public static void** check() {  
 CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *task2*(**null**));  
 CompletableFuture<String> cf = **cf1.exceptionally( ex -> {  
 System.*out*.println("Exception Details: "+ex.getMessage());  
 return "Good Result";  
 });** String result = cf.join();  
 System.***out***.println(**"Result : "**+result);  
}

OUTPUT

Task2 started running ...

finalResult = Some exception, so some Default Value

Task2 started running ...

finalResult = some default value as there is an exception

**whenComplete (BiConsumer<? super T,? super Throwable> action)**

🡺 **CompletableFuture<T>**

**This method is useful when you want to perform certain intermediate operation before returning the value.** For example querying a certain result, update the table for audit entry. This is a useful function where you want to log both good result or bad result like exception. This function returns either result or exception value.

**public** **static** **void** check() {

CompletableFuture<String> cf1 = CompletableFuture.*supplyAsync*(() -> *m1*());

CompletableFuture<String> cf = cf1.whenComplete((result, error) -> {

System.***out***.println(".... Intermediate operation ....");

System.***out***.println("Result: "+ result);

System.***out***.println("Error: "+ error);

});

String response = cf.join();

System.***out***.println("Response: "+ response);

}

**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.

CompletableFuture<Integer> future =

CompletableFuture.supplyAsync(() -> 1)

.thenApply(x -> x+1);

**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));

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

[runAsync](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#runAsync-java.lang.Runnable-) takes Runnable as input parameter and returns CompletableFuture<Void>, which means it does not return any result.

CompletableFuture<Void> run = CompletableFuture.runAsync(()-> System.out.println("hello"));

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

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

System.out.println("Hello");

return "result";

});

System.out.println(supply.get()); //result

**Conclusion :** So if you want the result to be returned, then choose supplyAsync or if you just want to run an async action, then choose runAsync.

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

The only difference is how methods throw exceptions. get() is declared in Future interface as

V get() throws InterruptedException, ExecutionException.

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

public T join() Instead it throws unchecked CompletionException. So you do not need a try-catch block and instead you can fully harness exceptionally() method.

**Cheat Sheet**

**Accept: thenAccept(), acceptEither(), thenAccceptBoth()**

**thenAccept()s**

CompletableFuture<Void> cf = cf1.thenAccept( (r) -> m3(r));

cf.join();

**acceptEither()**

CompletableFuture<Void> cf = cf1.acceptEither(cf2, (result) -> m3(result));

cf.join();

**thenAccceptBoth()**

CompletableFuture<Void> cf = cf1.thenAcceptBoth(cf2, (m1R, m2R) -> m4(m1R,m2R));

cf.join();

**apply: thenApply(), applyToEither()**

**thenApply()**

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();

**Run: thenRun()**, **runAfterEither()**, **runAfterBoth()**

**thenRun**

CompletableFuture<Void> cf = cf1.thenRun( () -> m5());

cf.join();

**runAfterEither()**

CompletableFuture<Void> cf = cf1.runAfterEither( cf2,() -> m5());

cf.join();

**runAfterBoth()**

CompletableFuture<Void> cf = cf1.runAfterBoth( cf2,() -> m5());

cf.join();

**thenCombine()**

CompletableFuture<String> cf = cf1.thenCombine(cf2, (m1R, m2R) -> m6(m1R, m2R));

String response = cf.join();

**thenCompose()**

CompletableFuture<String> cf = cf1.thenCompose( (m1R) -> CompletableFuture.supplyAsync(() -> getVal(m1R)));

String response = cf.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()**

CompletableFuture<String> cf = cf1.handle( (result, error) -> {

System.out.println(result);

System.out.println(error);

return result;

});

String response = cf.join();

**also it can be**

CompletableFuture<String> cf = cf1.handle( (result, error) -> m7(result,error));

String response = cf.join();

**whenComplete()**

CompletableFuture<String> cf = cf1.whenComplete( (result, error) -> {

System.out.println(result);

System.out.println(error);

});

String response = cf.join();

**Also it can be**

CompletableFuture<String> cf = cf1.whenComplete( (result, error) -> m7(result,error));

String response = cf.join();

**exceptionally()**

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

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

return "Good Result";

});

String result = cf.join();