CompletableFuture in Java 8 – 2021-2022

# Class CompletableFuture<T>

A [Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) that may be explicitly completed (setting its value and status), and may be used as a [CompletionStage](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html), supporting dependent functions and actions that trigger upon its completion.

When two or more threads attempt to [complete](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#complete-T-), [completeExceptionally](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#completeExceptionally-java.lang.Throwable-), or [cancel](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#cancel-boolean-) a CompletableFuture, only one of them succeeds. In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface [CompletionStage](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html) with the following policies:

* Actions supplied for dependent completions of *non-async* methods may be performed by the thread that completes the current CompletableFuture, or by any other caller of a completion method.
* All *async* methods without an explicit Executor argument are performed using the [ForkJoinPool.commonPool()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool--) (unless it does not support a parallelism level of at least two, in which case, a new Thread is created to run each task). To simplify monitoring, debugging, and tracking, all generated asynchronous tasks are instances of the marker interface [CompletableFuture.AsynchronousCompletionTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.AsynchronousCompletionTask.html).
* All CompletionStage methods are implemented independently of other public methods, so the behavior of one method is not impacted by overrides of others in subclasses.

CompletableFuture also implements [Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) with the following policies:

* Since (unlike [FutureTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html)) this class has no direct control over the computation that causes it to be completed, cancellation is treated as just another form of exceptional completion. Method [cancel](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#cancel-boolean-) has the same effect as completeExceptionally(new CancellationException()). Method [isCompletedExceptionally()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#isCompletedExceptionally--) can be used to determine if a CompletableFuture completed in any exceptional fashion.
* In case of exceptional completion with a CompletionException, methods [get()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#get--) and [get(long, TimeUnit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#get-long-java.util.concurrent.TimeUnit-) throw an [ExecutionException](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutionException.html) with the same cause as held in the corresponding CompletionException. To simplify usage in most contexts, this class also defines methods [join()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#join--) and [getNow(T)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#getNow-T-) that instead throw the CompletionException directly in these cases.

# 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 exceptions are both *checked* exceptions which means they need to be handled in your code. As you can see in your code an automatic code generator in your IDE asked if to creat try-catch block on your behalf.

try {

CompletableFuture.allOf(fanoutRequestList).get()

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

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 when using the disscused List<String> processfunction

CompletableFuture<List<String>> cf = CompletableFuture

.supplyAsync(this::process)

.exceptionally(this::getFallbackListOfStrings) // Here you can catch e.g. {@code join}'s CompletionException

.thenAccept(this::processFurther);

Example-1

**class** Task1 **implements** Runnable {

@Override

**public** **void** run() {

System.***out***.println("Running Thread Name : " + Thread.*currentThread*().getName());

**try** {

TimeUnit.***SECONDS***.sleep(3);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Thread operations over ...");

}

}

**public** **class** Test3 {

**public** **static** **void** showThreadOperation1() {

CompletableFuture<Void> completableFuture = CompletableFuture.*runAsync*(**new** Task1());

completableFuture.join(); // Wait till the completion

System.***out***.println("Operations over...");

}

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

*showThreadOperation1*();

}

}

**Execute a thread in Java 8**

Runnable task =

() -> {

String threadName = Thread.*currentThread*().getName();

System.***out***.println("Hello " + threadName);

};

Thread thread = **new** Thread(task);

thread.start();

System.***out***.println("Done!");

**How to user Executor perfectly**

ExecutorService executor = Executors.*newSingleThreadExecutor*();

executor.submit(

() -> {

String threadName = Thread.*currentThread*().getName();

System.***out***.println("Hello " + threadName);

});

**try** {

System.***out***.println("attempt to shutdown executor");

executor.shutdown();

executor.awaitTermination(5, TimeUnit.***SECONDS***);

} **catch** (InterruptedException e) {

System.***err***.println("tasks interrupted");

} **finally** {

**if** (!executor.isTerminated()) {

System.***err***.println("cancel non-finished tasks");

}

executor.shutdownNow();

System.***out***.println("shutdown finished");

}

**OUTPUT**

attempt to shutdown executor

Hello pool-1-thread-1

shutdown finished

**Executor with Callable in Java 8**

ExecutorService executor = Executors.*newWorkStealingPool*();

List<Callable<String>> callables = Arrays.*asList*(() -> "task1", () -> "task2", () -> "task3");

executor

.invokeAll(callables)

.stream()

.map(

future -> {

**try** {

**return** future.get();

} **catch** (Exception e) {

**throw** **new** IllegalStateException(e);

}

})

.forEach(System.***out***::println);

OUTPUT

task1

task2

task3

**Chaining of Completable Future**

**private** **static** **void** performTask(String stage) {

System.***out***.println("---------");

System.***out***.printf(

"stage: %s, time before task: %s, thread: %s%n",

stage, LocalTime.*now*(), Thread.*currentThread*().getName());

**try** {

// simulating long task

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.printf(

"stage: %s, time after task: %s, thread: %s%n",

stage, LocalTime.*now*(), Thread.*currentThread*().getName());

}

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

**CompletionStage<Void> cf =**

**CompletableFuture.*runAsync*(() -> *performTask*("first stage"));**

**cf = cf.thenRun(() -> *performTask*("second stage"));**

**cf = cf.thenRunAsync(() -> *performTask*("third stage"));**

**((CompletableFuture) cf).join();//waits until task is completed**

System.***out***.println("main exiting");

}

OUTPUT

---------

stage: first stage, time before task: 23:34:58.550, thread: ForkJoinPool.commonPool-worker-1

stage: first stage, time after task: 23:34:59.554, thread: ForkJoinPool.commonPool-worker-1

---------

stage: second stage, time before task: 23:34:59.555, thread: ForkJoinPool.commonPool-worker-1

stage: second stage, time after task: 23:35:00.557, thread: ForkJoinPool.commonPool-worker-1

---------

stage: third stage, time before task: 23:35:00.558, thread: ForkJoinPool.commonPool-worker-1

stage: third stage, time after task: 23:35:01.561, thread: ForkJoinPool.commonPool-worker-1

main exiting

**Simple example on CompletableFuture**

**public** **class** Test4 {

**public** **static** String performTask1() {

**try** {

TimeUnit.***SECONDS***.sleep(3);

System.***out***.println("Task execution going on ...");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**return** "Task1";

}

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

**CompletableFuture<Void> cf =**

**CompletableFuture.*runAsync*(**

**() -> {**

**try {**

**TimeUnit.*SECONDS*.sleep(3);**

**System.*out*.println("Execution over ...");**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**});**

System.***out***.println("Other tasks are going on ...");

Void blank = cf.get(); // This will block

System.***out***.println("Another task is going on ...");

CompletableFuture<String> cf1 = CompletableFuture.*completedFuture*(*performTask1*());

**if** (cf1.isDone()) {

System.***out***.println("Value- " + cf1.get());

}

}

**OUTPUT**

Other tasks are going on ...

Execution over ...

Another task is going on ...

Task execution going on ...

Value- Task1

# CompletableFuture with Executors

**public** **class** Test4 {

**public** **static** Future<String> calculateAsync() **throws** InterruptedException {

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

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

**exService.submit(**

**() -> {**

**Thread.*sleep*(500);**

**completableFuture.complete("Hello");**

**return null;**

**});**

**exService.shutdown();**

**return** completableFuture;

}

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

Future<String> completableFuture = *calculateAsync*();

String result = completableFuture.get();

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

}

}

OUTPUT

Result :::Hello

# Chain of task one by one using CompletableFuture

**public** **class** Chain {

**public** **static** String task1() {

**return** "Task-1";

}

**public** **static** String task2(String input) {

**return** input + " Task-2";

}

**public** **static** String task3(String input) {

**return** input + " Task-3";

}

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

**CompletableFuture<String> cf =**

**CompletableFuture.*supplyAsync*(() -> *task1*())**

**.thenApply(data -> *task2*(data))**

**.thenApply(data -> *task3*(data));**

String result = cf.get();

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

}

}

OUTPUT

Result after chained task :::Task-1 Task-2 Task-3

# *thenCompose* method to chain two *Futures* sequentially

**CompletableFuture<String> completableFuture  = CompletableFuture.supplyAsync(() -> "Hello")**

**.thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " World"));**

**assertEquals("Hello World", completableFuture.get());**

# Another example on .thenCompose()

**public** **class** Chain {

**public** **static** String task1() {

**return** "Task-1";

}

**public** **static** String task2(String input) {

**return** input + " Task-2";

}

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

CompletableFuture<String> completableFuture =

**CompletableFuture.*supplyAsync*(() -> *task1*())**

**.thenCompose(s -> CompletableFuture.*supplyAsync*(() -> *task2*(s)));**

System.***out***.println("Now result :::"+completableFuture.get());

}

}

The thenCompose method together with thenApply implement basic building blocks of the monadic pattern. They closely relate to the map and flatMap methods of Stream and Optional classes also available in Java 8.

Both methods receive a function and apply it to the computation result, but the thenCompose(flatMap) method **receives a function that returns another object of the same type**. This functional structure allows composing the instances of these classes as building blocks.

If you want to execute two independent Futures and do something with their results, use the thenCombine method that accepts a Future and a Function with two arguments to process both results:

**CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> "Hello")**

**.thenCombine(CompletableFuture.supplyAsync(**

**() -> " World"), (s1, s2) -> s1 + s2));**

**assertEquals("Hello World", completableFuture.get());**

**Complete Example is given below.**

**public** **static** CompletableFuture<String> task1() {

**return** CompletableFuture.*supplyAsync*(

() -> {

**return** "task-1";

});

}

**public** **static** CompletableFuture<String> task2() {

**return** CompletableFuture.*supplyAsync*(

() -> {

**return** "task-2";

});

}

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

**CompletableFuture<String> task1Future = *task1*();**

**CompletableFuture<String> task2Future = *task2*();**

CompletableFuture<String> combineFuture =

**task1Future.thenCombine(**

**task2Future,**

**(a, b) -> {**

**return a + "-" + b;**

**});**

System.***out***.println("Combined Future : " + combineFuture.get());

}

#### Exception handling

There are several options to handle exceptions while working with CompletableFuture. We are going to see how to use the methods **exceptionally() and handle().**

# Exceptionally

**CompletableFuture<Integer> goodFuture =**

**CompletableFuture.*supplyAsync*(**

**() -> {**

**return 10 / 2;**

**});**

**CompletableFuture<Integer> exceptionalFuture =**

**CompletableFuture.*supplyAsync*(**

**() -> {**

**return 10 / 0;**

**});**

**CompletableFuture<Integer> fallback1 = goodFuture.exceptionally(x -> 0);**

**CompletableFuture<Integer> fallback2 = exceptionalFuture.exceptionally(x -> 0);**

System.***out***.println("First Fallback : " + fallback1.get());

System.***out***.println("Second Fallback : " + fallback2.get());

# Handle()

CompletableFuture<Integer> completableFutureHandleOk =

CompletableFuture.*supplyAsync*(

() -> {

// big task

**return** 10 / 0; // exception division by zero

});

CompletableFuture<Integer> handleOkError =

completableFutureHandleOk.handle(

(ok, ex) -> {

**if** (ok != **null**) **return** ok;

**else** {

ex.printStackTrace();

**return** **null**;

}

});

System.***out***.println( "ok or error ? " + handleOkError.get() );

OUTPUT

java.util.concurrent.CompletionException: java.lang.ArithmeticException: / by zero

at java.util.concurrent.CompletableFuture.encodeThrowable(Unknown Source)

at java.util.concurrent.CompletableFuture.completeThrowable(Unknown Source)

at java.util.concurrent.CompletableFuture$AsyncSupply.run(Unknown Source)

at java.util.concurrent.CompletableFuture$AsyncSupply.exec(Unknown Source)

at java.util.concurrent.ForkJoinTask.doExec(Unknown Source)

at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(Unknown Source)

at java.util.concurrent.ForkJoinPool.runWorker(Unknown Source)

at java.util.concurrent.ForkJoinWorkerThread.run(Unknown Source)

Caused by: java.lang.ArithmeticException: / by zero

at com.ddlab.rnd.type1.Chain.lambda$2(Chain.java:26)

... 6 more

ok or error ? null

**StackOverflow questions – Difference between thenCombine and thenCompose**

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));

The updated Javadocs in Java 9 will probably help understand it better:

## [thenApply](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletionStage.html#thenApply-java.util.function.Function-)

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

Returns a new [CompletionStage](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletionStage.html) that, when this stage completes normally, is executed with this stage's result as the argument to the supplied function.

This method is analogous to [Optional.map](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#map-java.util.function.Function-) and [Stream.map](https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html#map-java.util.function.Function-).

See the [CompletionStage](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletionStage.html) documentation for rules covering exceptional completion.

## [thenCompose](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletionStage.html#thenCompose-java.util.function.Function-)

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

Returns a new [CompletionStage](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletionStage.html) that is completed with the same value as the CompletionStagereturned by the given function.

When this stage completes normally, the given function is invoked with this stage's result as the argument, returning another CompletionStage. When that stage completes normally, theCompletionStage returned by this method is completed with the same value.

To ensure progress, the supplied function must arrange eventual completion of its result.

This method is analogous to [Optional.flatMap](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#flatMap-java.util.function.Function-) and [Stream.flatMap](https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html#flatMap-java.util.function.Function-).

See the [CompletionStage](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletionStage.html) documentation for rules covering exceptional completion.

Another example

CompletableFuture<String> future = CompletableFuture.*completedFuture*("foo");

future.thenApply(str -> {

System.***out***.println("Stage 1: " + str);

**return** "bar";

});

future.thenApply(str -> {

System.***out***.println("Stage 2: " + str);

**throw** **new** RuntimeException();

});

future.thenApply(str -> {

System.***out***.println("Stage 3: " + str);

**return** "abc";

});

future.exceptionally(e -> {

System.***out***.println("Exceptionally");

**return** "I am the blank";

});

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

OUTPUT

Stage 1: foo

Stage 2: foo

Stage 3: foo

Now Result :::foo

### Combining 2 CompletableFutures together

In case of Future, there is no way to create asynchronous workflow i.e. long running computation. But *CompletableFuture* provides us with 2 methods to achieve this functionality:

**i) thenCompose()**

It is a method of combining 2 dependent futures together. This method takes a function that returns a *CompletableFuture*instance. The argument of this function is the result of the previous computation step. This allows us to use this value inside the next *CompletableFuture*‘s lambda.

**CompletableFuture<String> completableFuture =**

**CompletableFuture.*supplyAsync*(() -> "Hello")**

**.thenCompose(**

**value -> CompletableFuture.*supplyAsync*(() -> value + " Knolders! Its thenCompose"));**

**completableFuture.thenAccept(System.*out*::println); // Hello Knolders! Its thenCompose**

**ii) thenCombine()**

It is a method of combining 2 independent futures together and do something  with there result after both of them are complete. Combining is accomplished by taking 2 successful CompletionStages and having the results from both used as parameters to a BiFunction to produce another result.  For example:

**CompletableFuture<String> completableFuture =**

**CompletableFuture.*supplyAsync*(() -> "Hello")**

**.thenCombine(**

**CompletableFuture.*supplyAsync*(() -> " Knolders! Its thenCombine"),**

**(value1, value2) -> value1 + value2);**

**completableFuture.thenAccept(System.*out*::println); // Hello Knolders! Its thenCombine**

**iii) thenAcceptBoth()**

It is used when you want to perform some operation with two independent Future’s result but don’t need to pass any resulting value down a Future chain. For example:

**CompletableFuture<Void> completableFuture =**

**CompletableFuture.*supplyAsync*(() -> "Hello")**

**.thenAcceptBoth(**

**CompletableFuture.*supplyAsync*(() -> " Knolders! Its thenAcceptBoth"),**

**(value1, value2) ->**

**System.*out*.println(value1 + value2)); // Hello Knolders! Its thenAcceptBoth**

### ****Combining multiple CompletableFutures together****

What if there comes a scenario where you want to combine 100 different Futures that you want to run in parallel and then run some function after all of them completes. Future does not provide us any way in order to achieve this functionality but CompletableFuture does. There are two methods of implementing this:

**i) CompletableFuture.allOf()**

CompletableFuture.allOf()  static method 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.

|  |  |
| --- | --- |
|  | private static void allOf() { |
|  | CompletableFuture<String> completableFuture1 |
|  | = CompletableFuture.supplyAsync(() -> "Hello"); |
|  | CompletableFuture<String> completableFuture2 |
|  | = CompletableFuture.supplyAsync(() -> "Knolders!"); |
|  | CompletableFuture<String> completableFuture3 |
|  | = CompletableFuture.supplyAsync(() -> "Its allOf"); |
|  |  |
|  | CompletableFuture<Void> combinedFuture |
|  | = CompletableFuture.allOf(completableFuture1, completableFuture2, completableFuture3); |
|  | } |

**Limitation:**

The limitation of this method is that the return type is CompletableFuture i.e. it does not return the combined results of all *Futures*. Instead you have to manually get results from *Futures*.

|  |  |
| --- | --- |
|  | combinedFuture.get(); |
|  | assertTrue(future1.isDone()); |
|  | assertTrue(future2.isDone()); |
|  | assertTrue(future3.isDone()); |

Fortunately, *CompletableFuture.join()* method and Java 8 Streams API helps to resolve this issue:

|  |  |
| --- | --- |
|  | combinedFuture.thenApply(v -> |
|  | Stream.of(completableFuture1, completableFuture2, completableFuture3). |
|  | map(CompletableFuture::join). |
|  | collect(Collectors.toList())); |
|  | String combined = Stream.of(completableFuture1, completableFuture2, completableFuture3) |
|  | .map(CompletableFuture::join) |
|  | .collect(Collectors.joining(" ")); |
|  | System.out.println(combined); // Hello Knolders! Its allOf |

**Note :**

The *CompletableFuture.join()* method is similar to the *CompletableFuture.get()* method, but it throws an unchecked exception in case the *Future* does not complete normally. This makes it possible to use it as a method reference in the *Stream.map()* method.

**ii) 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. *CompletableFuture.anyOf()* takes a varargs of Futures and returns *CompletableFuture.*

|  |  |
| --- | --- |
|  | private static void anyOf() throws Exception { |
|  | 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); |
|  | anyOfFuture.thenAccept(System.out::println); // Result of Future 2 |
|  | } |

**Limitation:**The problem with CompletableFuture.anyOf() is that if you have CompletableFuture that return results of different types, then you won’t know the type of your final CompletableFuture.

5. Exception Handling

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

|  |  |
| --- | --- |
|  | private static void chainOfThenApply() { |
|  | 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.

So, there are 2 ways in order to handle this scenario:

**i) Handle exceptions using exceptionally() callback**

*exceptionally()* gives us a chance to recover by returning a default value or taking an alternative function that will be executed if preceding calculation fails with an exception.

|  |  |
| --- | --- |
|  | private static void exception() { |
|  | Integer age = -1; |
|  | CompletableFuture<String> exceptionFuture = 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!"; |
|  | }); |
|  | exceptionFuture.thenAccept(System.out::println); //Unknown! |
|  | } |

**ii) 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. If an exception occurs, then the result argument will be null, otherwise, the ex argument will be null.

|  |  |
| --- | --- |
|  | private static void exceptionUsingHandle() { |
|  | Integer age = -1; |
|  | CompletableFuture<String> exceptionFuture = CompletableFuture.supplyAsync(() -> { |
|  | if (age < 0) { |
|  | throw new IllegalArgumentException("Age can not be negative"); |
|  | } |
|  | if (age > 18) { |
|  | return "Adult"; |
|  | } else { |
|  | return "Child"; |
|  | } |
|  | }).handle((result, ex) -> { |
|  | if (ex != null) { |
|  | System.out.println("Oops! We have an exception - " + ex.getMessage()); |
|  | return "Unknown!"; |
|  | } |
|  | return result; |
|  | }); |
|  | exceptionFuture.thenAccept(System.out::println); // Unknown! |
|  | } |