Java 8 CompletableFuture Example

java.util.concurrent.CompletableFuture is a Future in java 8 which is the derived class of java.util.concurrent.CompletionStage. CompletableFuture can be completed setting value and status explicitly. There are different methods in CompletableFuture that can be used to handle task. Here in this page we will provide the example of some methods like supplyAsync, thenApply, join, thenAccept, whenComplete and getNow.

CompletableFuture.supplyAsync

supplyAsync accepts a Supplier as an argument and complete its job asynchronously. The result of supplier is run by a task from ForkJoinPool.commonPool() as default. We can also pass our Executor. Finally supplyAsync method returns CompletableFuture on which we can apply other methods.

CompletableFuture.thenApply

thenApply method accepts a function as an argument. Once the calling CompletableFuture completes , then on the result of this stage, that function is applied by thenApply method and returns a CompletableFuture.

CompletableFuture.join

join method returns the result after completion or throws CompletionException. This method waits for the completion of calling completion stage.   
**CompletableFutureOneExample.java**

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.CompletableFuture;

public class CompletableFutureOneExample {

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

List<Integer> list = Arrays.asList(10,20,30,40);

list.stream().map(data->CompletableFuture.supplyAsync(()->getNumber(data))).

map(compFuture->compFuture.thenApply(n->n\*n)).map(t->t.join())

.forEach(s->System.out.println(s));

}

private static int getNumber(int a){

return a\*a;

}

}

Find the output.

10000

160000

810000

2560000

CompletableFuture.thenAccept

thenAccept method accepts Consumer as an argument. On the completion of any completion stage, thenAccept method applies Consumer on the result and returns CompletableFuture.   
**CompletableFutureTwoExample.java**

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.CompletableFuture;

public class CompletableFutureTwoExample {

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

List<String> list = Arrays.asList("A","B","C","D");

list.stream().map(data->CompletableFuture.supplyAsync(()->"Processing:"+data)).

map(compFuture->compFuture.thenAccept(s->System.out.println(s))).map(t->t.join())

.count();

}

}

Find the output.

Processing:A

Processing:B

Processing:C

Processing:D

CompletableFuture.whenComplete

whenComplete method uses BiConsumer as an argument. Once the calling completion stage completes, whenComplete method applies completion stage result on BiConsumer. BiConsumer takes first argument as result and second argument as error if any.   
**CompletableFutureThreeExample.java**

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.CompletableFuture;

public class CompletableFutureThreeExample {

public static void main(String[] args) {

List<String> list = Arrays.asList("A","B","C","D");

list.stream().map(s->CompletableFuture.supplyAsync(()->s+s))

.map(f->f.whenComplete((result,error)->System.out.println(result+" Error:"+error))).count();

}

}

Find the output.

AA Error:null

BB Error:null

CC Error:null

DD Error:null

CompletableFuture.getNow

getNow is a method that if calling completion stage is not completed then the value passed to getNow will be set to result.   
**CompletableFutureFourExample.java**

import java.util.Arrays;

import java.util.List;

import java.util.concurrent.CompletableFuture;

public class CompletableFutureFourExample {

public static void main(String[] args) {

List<String> list = Arrays.asList("A","B","C","D");

list.stream().map(s->CompletableFuture.supplyAsync(()->s+s))

.map(f->f.getNow("Not Done")).forEach(s->System.out.println(s));

}

}

The output of the above program will be random. At the time of calling getNow, if thread is completed, then that result is returned else the default value set by getNow will be returned.

Not Done

Not Done

CC

Not Done

## **1. Introduction**

This article is a guide to the functionality and use cases of the CompletableFutureclass – introduced as a Java 8 Concurrency API improvement.

## **2. Asynchronous Computation in Java**

Asynchronous computation is difficult to reason about. Usually we want to think of any computation as a series of steps. But in case of asynchronous computation, **actions represented as callbacks tend to be either scattered across the code or deeply nested inside each other**. Things get even worse when we need to handle errors that might occur during one of the steps.

The Future interface was added in Java 5 to serve as a result of an asynchronous computation, but it did not have any methods to combine these computations or handle possible errors.

**In Java 8, the CompletableFuture class was introduced.** Along with the Futureinterface, it also implemented the CompletionStage interface. This interface defines the contract for an asynchronous computation step that can be combined with other steps.

CompletableFuture is at the same time a building block and a framework with **about 50 different methods for composing, combining, executing asynchronous computation steps and handling errors**.

Such a large API can be overwhelming, but these mostly fall in several clear and distinct use cases.

## **3. Using**CompletableFuture**as a Simple**Future

First of all, the CompletableFuture class implements the Future interface, so you can **use it as a Future implementation, but with additional completion logic**.

For example, you can create an instance of this class with a no-arg constructor to represent some future result, hand it out to the consumers and complete it at some time in the future using the complete method. The consumers may use the get method to block the current thread until this result will be provided.

In the example below we have a method that creates a CompletableFutureinstance, then spins off some computation in another thread and returns the Future immediately.

When the computation is done, the method completes the Future by providing the result to the complete method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public Future<String> calculateAsync() throws InterruptedException {      CompletableFuture<String> completableFuture        = new CompletableFuture<>();        Executors.newCachedThreadPool().submit(() -> {          Thread.sleep(500);          completableFuture.complete("Hello");          return null;      });        return completableFuture;  } |

To spin off the computation, we use the Executor API which is described in the article [“Introduction to Thread Pools in Java”](http://www.baeldung.com/thread-pool-java-and-guava), but this method of creating and completing a CompletableFuture can be used together with any concurrency mechanism or API including raw threads.

Notice that **the calculateAsync method returns a Future instance**.

We simply call the method, receive the Future instance and call the get method on it when we’re ready to block for the result.

Also notice that the get method throws some checked exceptions, namely ExecutionException (encapsulating an exception that occurred during a computation) and InterruptedException (an exception signifying that a thread executing a method was interrupted):

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Future<String> completableFuture = calculateAsync();    // ...    String result = completableFuture.get();  assertEquals("Hello", result); |

**If you already know the result of a computation**, you can use the static completedFuture method with an argument that represents a result of this computation. Then the get method of the Future will never block, immediately returning this result instead.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Future<String> completableFuture =    CompletableFuture.completedFuture("Hello");    // ...    String result = completableFuture.get();  assertEquals("Hello", result); |

As an alternative scenario, you may want to **cancel the execution of a Future**.

Suppose we didn’t manage to find a result and decided to cancel an asynchronous execution altogether. This can be done with the Future‘s cancelmethod. This method receives a boolean argument mayInterruptIfRunning, but in the case of CompletableFuture it has no effect, as interrupts are not used to control processing for CompletableFuture.

Here’s a modified version of the asynchronous method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public Future<String> calculateAsyncWithCancellation() throws InterruptedException {      CompletableFuture<String> completableFuture = new CompletableFuture<>();        Executors.newCachedThreadPool().submit(() -> {          Thread.sleep(500);          completableFuture.cancel(false);          return null;      });        return completableFuture;  } |

When we block on the result using the Future.get() method, it throws CancellationException if the future is canceled:

|  |  |
| --- | --- |
| 1  2 | Future<String> future = calculateAsyncWithCancellation();  future.get(); // CancellationException |

## **4.**CompletableFuture**with Encapsulated Computation Logic**

The code above allows us to pick any mechanism of concurrent execution, but what if we want to skip this boilerplate and simply execute some code asynchronously?

Static methods runAsync and supplyAsync allow us to create a CompletableFutureinstance out of Runnable and Supplier functional types correspondingly.

Both Runnable and Supplier are functional interfaces that allow passing their instances as lambda expressions thanks to the new Java 8 feature.

The Runnable interface is the same old interface that is used in threads and it does not allow to return a value.

The Supplier interface is a generic functional interface with a single method that has no arguments and returns a value of a parameterized type.

This allows to **provide an instance of the Supplier as a lambda expression that does the calculation and returns the result**. This is as simple as:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CompletableFuture<String> future    = CompletableFuture.supplyAsync(() -> "Hello");    // ...    assertEquals("Hello", future.get()); |

## **5. Processing Results of Asynchronous Computations**

The most generic way to process the result of a computation is to feed it to a function. The thenApply method does exactly that: accepts a Function instance, uses it to process the result and returns a Future that holds a value returned by a function:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CompletableFuture<String> completableFuture    = CompletableFuture.supplyAsync(() -> "Hello");    CompletableFuture<String> future = completableFuture    .thenApply(s -> s + " World");    assertEquals("Hello World", future.get()); |

If you don’t need to return a value down the Future chain, you can use an instance of the Consumer functional interface. Its single method takes a parameter and returns void.

There’s a method for this use case in the CompletableFuture — the thenAcceptmethod receives a Consumer and passes it the result of the computation. The final future.get() call returns an instance of the Void type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CompletableFuture<String> completableFuture    = CompletableFuture.supplyAsync(() -> "Hello");    CompletableFuture<Void> future = completableFuture    .thenAccept(s -> System.out.println("Computation returned: " + s));    future.get(); |

At last, if you neither need the value of the computation nor want to return some value at the end of the chain, then you can pass a Runnable lambda to the thenRun method. In the following example, after the future.get() method is called, we simply print a line in the console:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CompletableFuture<String> completableFuture    = CompletableFuture.supplyAsync(() -> "Hello");    CompletableFuture<Void> future = completableFuture    .thenRun(() -> System.out.println("Computation finished."));    future.get(); |

## **6. Combining Futures**

The best part of the CompletableFuture API is the **ability to combine CompletableFuture instances in a chain of computation steps**.

The result of this chaining is itself a CompletableFuture that allows further chaining and combining. This approach is ubiquitous in functional languages and is often referred to as a monadic design pattern.

**In the following example we use the thenCompose method to chain two Futures sequentially.**

Notice that this method takes a function that returns a CompletableFutureinstance. 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CompletableFuture<String> completableFuture    = CompletableFuture.supplyAsync(() -> "Hello")      .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " World"));    assertEquals("Hello World", completableFuture.get()); |

The thenCompose method together with thenApply implement basic building blocks of the monadic pattern. They closely relate to the map and flatMapmethods 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CompletableFuture<String> completableFuture    = CompletableFuture.supplyAsync(() -> "Hello")      .thenCombine(CompletableFuture.supplyAsync(        () -> " World"), (s1, s2) -> s1 + s2));    assertEquals("Hello World", completableFuture.get()); |

A simpler case is when you want to do something with two Futures‘ results, but don’t need to pass any resulting value down a Future chain. The thenAcceptBothmethod is there to help:

|  |  |
| --- | --- |
| 1  2  3 | CompletableFuture future = CompletableFuture.supplyAsync(() -> "Hello")    .thenAcceptBoth(CompletableFuture.supplyAsync(() -> " World"),      (s1, s2) -> System.out.println(s1 + s2)); |

## **7. Running Multiple**Futures**in Parallel**

When we need to execute multiple Futures in parallel, we usually want to wait for all of them to execute and then process their combined results.

The CompletableFuture.allOf static method allows to wait for completion of all of the Futures provided as a var-arg:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | CompletableFuture<String> future1    = CompletableFuture.supplyAsync(() -> "Hello");  CompletableFuture<String> future2    = CompletableFuture.supplyAsync(() -> "Beautiful");  CompletableFuture<String> future3    = CompletableFuture.supplyAsync(() -> "World");    CompletableFuture<Void> combinedFuture    = CompletableFuture.allOf(future1, future2, future3);    // ...    combinedFuture.get();    assertTrue(future1.isDone());  assertTrue(future2.isDone());  assertTrue(future3.isDone()); |

Notice that the return type of the CompletableFuture.allOf() is a CompletableFuture<Void>. The limitation of this method is that it does not return the combined results of all Futures. Instead you have to manually get results from Futures. Fortunately, CompletableFuture.join() method and Java 8 Streams API makes it simple:

|  |  |
| --- | --- |
| 1  2  3  4  5 | String combined = Stream.of(future1, future2, future3)    .map(CompletableFuture::join)    .collect(Collectors.joining(" "));    assertEquals("Hello Beautiful World", combined); |

The CompletableFuture.join() method is similar to the 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.

## **8. Handling Errors**

For error-handling in a chain of asynchronous computation steps, throw/catchidiom had to be adapted in a similar fashion.

Instead of catching an exception in a syntactic block, the CompletableFuture class allows you to handle it in a special handle method. This method receives two parameters: a result of a computation (if it finished successfully) and the exception thrown (if some computation step did not complete normally).

In the following example we use the handle method to provide a default value when the asynchronous computation of a greeting was finished with an error because no name was provided:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | String name = null;    // ...    CompletableFuture<String> completableFuture    =  CompletableFuture.supplyAsync(() -> {        if (name == null) {            throw new RuntimeException("Computation error!");        }        return "Hello, " + name;    })}).handle((s, t) -> s != null ? s : "Hello, Stranger!");    assertEquals("Hello, Stranger!", completableFuture.get()); |

As an alternative scenario, suppose we want to manually complete the Future with a value, as in the first example, but also to have the ability to complete it with an exception. The completeExceptionally method is intended for that. The completableFuture.get() method in the following example throws an ExecutionException with a RuntimeException as its cause:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CompletableFuture<String> completableFuture = new CompletableFuture<>();    // ...    completableFuture.completeExceptionally(    new RuntimeException("Calculation failed!"));    // ...    completableFuture.get(); // ExecutionException |

In the example above we could have handled the exception with the handlemethod asynchronously, but with the get method we can use a more typical approach of a synchronous exception processing.

## **9. Async Methods**

Most methods of the fluent API in CompletableFuture class have two additional variants with the Async postfix. These methods are usually intended for **running a corresponding step of execution in another thread**.

The methods without the Async postfix run the next execution stage using a calling thread. The Async method without the Executor argument runs a step using the common fork/join pool implementation of Executor that is accessed with the ForkJoinPool.commonPool() method. The Async method with an Executorargument runs a step using the passed Executor.

Here’s a modified example that processes the result of a computation with a Function instance. The only visible difference is the thenApplyAsync method. But under the hood the application of a function is wrapped into a ForkJoinTaskinstance (for more information on the fork/join framework, see the article [“Guide to the Fork/Join Framework in Java”](http://www.baeldung.com/java-fork-join)). This allows to parallelize your computation even more and use system resources more efficiently.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CompletableFuture<String> completableFuture    = CompletableFuture.supplyAsync(() -> "Hello");    CompletableFuture<String> future = completableFuture    .thenApplyAsync(s -> s + " World");    assertEquals("Hello World", future.get()); |