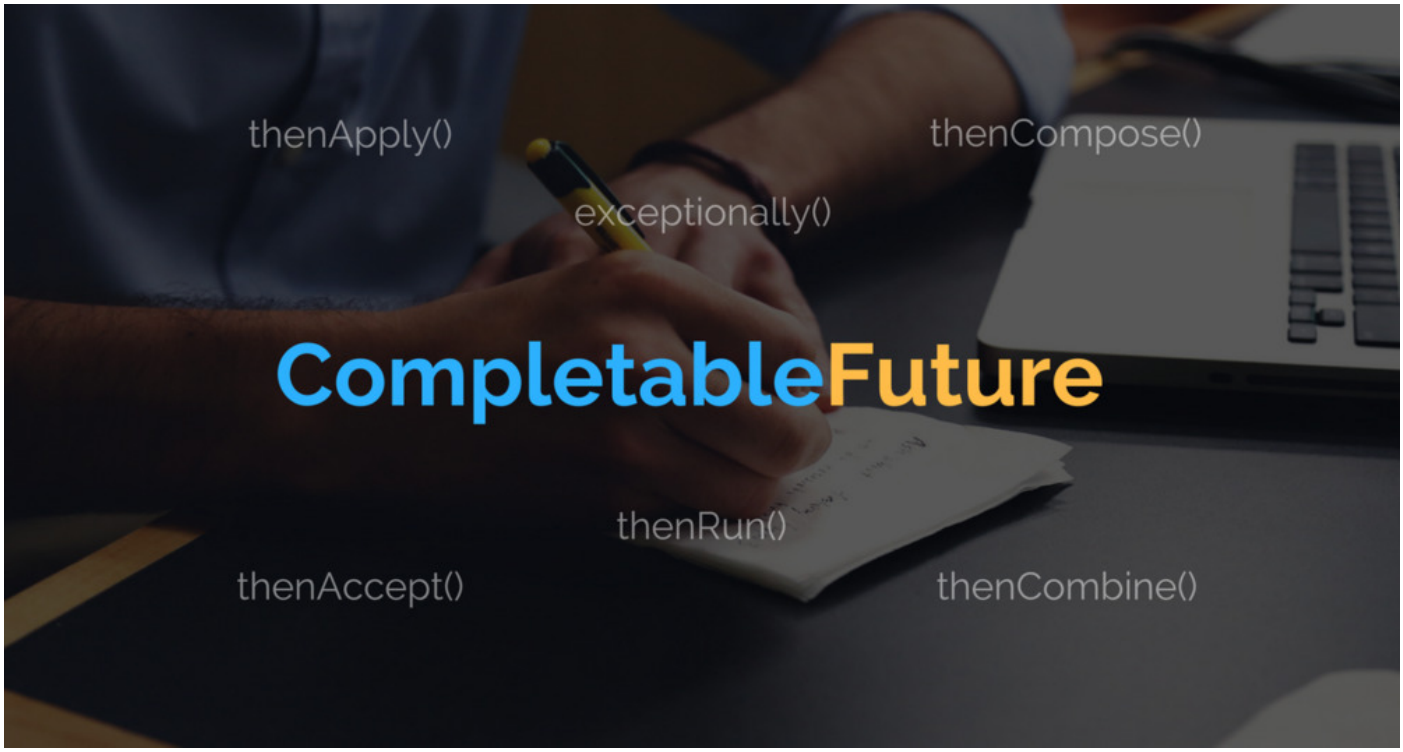
Java CompletableFuture Tutorial with Examples

Rajeev Singh • Java • Jul 18, 2017 • 15 mins read



Java 8 came up with tons of new features and enhancements like Lambda expressions, Streams, CompletableFutures 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.

Also Read: Java Concurrency and Multithreading Basics

CompletableFuture is an extension to Java's Future API 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.

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!

2. **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.

3. **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.

4. **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.

5. **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 -

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 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 -

```
// Using Lambda Expression
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
    // Simulate a long-running Job
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
```

```
});
```

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> 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> 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 -**

Yes! CompletableFuture executes these tasks in a thread obtained from the global ForkJoinPool.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 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<T,R> as an argument. Function<T,R> 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(() -> {
```

```
        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> 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())
});
```

```
// 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()` -

```
}).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(() -> {
        return UserService.getUserDetails(userId);
    });
}

CompletableFuture<Double> getCreditRating(User user) {
    return CompletableFuture.supplyAsync(() -> {
        return 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);
});
```

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 -

```
    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(() -> {
```

```
            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) {
```

```
            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.