# Java CompletableFuture

[CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html) is a class introduced in Java 8 that allows us to write asynchronous, non-blocking code. It is a powerful tool that can help us write code that is more efficient and responsive.

Here’s an example of how to use [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank) in Java:

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "Hello, world!";  
});  
  
future.thenAccept(result -> System.out.println(result));

In the example above, we’re creating a [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank) that will supply a result in the future. We're passing a lambda expression that simulates a long-running operation by sleeping for 5 seconds. After the operation is complete, it will return the string "Hello, world!".

We’re then calling the thenAccept() method on the [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank)object to specify what to do when the operation is complete. In this case, we're passing a lambda expression that simply prints the result to the console.

When we run this code, it will print “Hello, world!” to the console after a delay of 5 seconds.

Here’s another example that shows how to use [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank)to chain multiple asynchronous operations together:

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 10)  
 .thenApplyAsync(result -> result \* 2)  
 .thenApplyAsync(result -> result + 5);  
  
future.thenAccept(result -> System.out.println(result));

In this example, we’re creating a [CompletableFuture](https://www.java67.com/2022/02/completablefuture-in-java-with-example.html" \t "_blank)that starts by supplying the value 10. We're then using the thenApplyAsync() method to chain two additional operations together: multiplying the result by 2, and then adding 5 to the result.

Finally, we’re calling thenAccept() to specify what to do when the entire operation is complete. In this case, we're simply printing the final result to the console.

When we run this code, it will print “25” to the console.

## Future vs CompletableFuture

[Future](https://javarevisited.blogspot.com/2015/01/how-to-use-future-and-futuretask-in-Java.html) and CompletableFuture are both abstractions for representing a result that will be available in the future, but there are some important differences between them.

1. Blocking vs non-blocking**:** One of the key differences between Future and CompletableFuture is that Future is a blocking API, whereas CompletableFuture is non-blocking. With a Future object, you must call the get() method to retrieve the result, but this method blocks until the result is available. In contrast, with a CompletableFuture object, you can use various non-blocking methods to retrieve the result, such as thenApply(), thenAccept(), or join().
2. Composition: CompletableFuture provides a more powerful composition API than Future. With Future, it is difficult to chain multiple asynchronous operations together or to combine the results of multiple operations. CompletableFuture, on the other hand, provides methods such as thenCompose(), thenCombine(), and allOf() that make it easy to compose multiple asynchronous operations and to handle their results in a non-blocking way.
3. Exception Handling: CompletableFuture provides better exception handling than Future. With Future, you can only check if the computation completed successfully or not. If an exception occurs during the computation, you have to catch it explicitly. In contrast, with CompletableFuture, you can handle exceptions in a more declarative way using methods like exceptionally() and handle().
4. Completion: With a Future object, there is no way to explicitly complete the future. Once you submit a task to an executor service and get a Future object in return, you can only wait for the task to complete. With CompletableFuture, you have more control over the completion of the future. You can complete it explicitly by calling complete(), completeExceptionally(), or cancel() methods.

In summary, CompletableFuture provides a more flexible and powerful API for working with asynchronous computations than [Future](https://javarevisited.blogspot.com/2015/01/how-to-use-future-and-futuretask-in-Java.html). It offers non-blocking methods, composition methods, better exception handling, and explicit completion methods, which makes it easier to write robust and scalable concurrent code.

**Run multiple futures in parallel using CompletableFuture**

To run multiple futures in parallel using CompletableFuture, you can use the CompletableFuture.allOf() method. Here's an example:

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return "Result 1";  
});  
  
CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return "Result 2";  
});  
  
CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return "Result 3";  
});  
  
CompletableFuture<Void> allFutures = CompletableFuture.allOf(future1, future2, future3);  
  
allFutures.thenRun(() -> {  
 // All futures completed  
 String result1 = future1.join();  
 String result2 = future2.join();  
 String result3 = future3.join();  
 System.out.println(result1 + ", " + result2 + ", " + result3);  
});

In this example, we’re creating three CompletableFuture objects, each representing a long-running operation that returns a String result. We're then using the CompletableFuture.allOf() method to combine all three futures into a single future that completes when all three are complete.

We’re then calling the thenRun() method on the allFutures object to specify what to do when all three futures have completed. In this case, we're using the join() method to get the results of each future, and then printing them to the console.

When we run this code, it will start all three futures in parallel and then wait for them to complete before printing the results. The order of the results may vary depending on which future completes first, but all three results will be printed to the console once they are all available.

**Handle errors — CompletableFuture**

CompletableFuture provides a number of methods for handling errors that can occur during asynchronous operations. Here are a few examples:

1. Handling an exception in a single CompletableFuture:

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {  
 int result = 10 / 0; // Causes an ArithmeticException  
 return result;  
});  
  
future.exceptionally(ex -> {  
 System.out.println("Exception occurred: " + ex.getMessage());  
 return 0; // Default value to return if there's an exception  
}).thenAccept(result -> {  
 System.out.println("Result: " + result);  
});

In this example, we’re creating a CompletableFuture that will throw an ArithmeticException because we're dividing by zero. We're then calling the exceptionally() method to specify what to do if there's an exception. In this case, we're printing an error message to the console and returning a default value of 0.

We’re then calling the thenAccept() method to specify what to do when the operation is complete, whether there was an exception or not. In this case, we're simply printing the result to the console.

When we run this code, it will print “Exception occurred: / by zero” to the console followed by “Result: 0”.

2. Handling errors in multiple CompletableFuture objects:

CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return 10;  
});  
  
CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {  
 int result = 10 / 0; // Causes an ArithmeticException  
 return result;  
});  
  
CompletableFuture<Integer> future3 = CompletableFuture.supplyAsync(() -> {  
 // Some long-running operation  
 return 20;  
});  
  
CompletableFuture<Void> allFutures = CompletableFuture.allOf(future1, future2, future3);  
  
allFutures.exceptionally(ex -> {  
 System.out.println("Exception occurred: " + ex.getMessage());  
 return null; // Default value to return if there's an exception  
}).thenRun(() -> {  
 // All futures completed  
 int result1 = future1.join();  
 int result2 = future2.join();  
 int result3 = future3.join();  
 System.out.println(result1 + ", " + result2 + ", " + result3);  
});

In this example, we’re creating three CompletableFuture objects, two of which will throw an exception. We're then using the CompletableFuture.allOf() method to combine all three futures into a single future that completes when all three are complete.

We’re then calling the exceptionally() method on the allFutures object to specify what to do if there's an exception. In this case, we're printing an error message to the console and returning a default value of null.

We’re then calling the thenRun() method to specify what to do when all three futures have completed. In this case, we're using the join() method to get the results of each future, and then printing them to the console.

When we run this code, it will print “Exception occurred: / by zero” to the console followed by “10, 0, 20”. Note that the second future returned a default value of 0 because of the exception, which is why we see “0” in the output.

**Async Methods of CompletableFuture**

CompletableFuture provides a set of asynchronous methods to execute multiple tasks concurrently and process the results as soon as they become available. These methods allow you to create a chain of dependent tasks and execute them in parallel, improving the performance of your application.

Here are some examples of the async methods available in CompletableFuture:

1. thenApplyAsync(): This method is used to process the result of a task asynchronously and return a new CompletableFuture with the transformed result. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");  
  
CompletableFuture<Integer> transformedFuture = future.thenApplyAsync(s -> {  
 System.out.println("Thread: " + Thread.currentThread().getName());  
 return s.length();  
});  
  
transformedFuture.thenAccept(length -> {  
 System.out.println("Thread: " + Thread.currentThread().getName());  
 System.out.println("Length of Hello: " + length);  
});

In this example, we’re creating a CompletableFuture that returns the string "Hello". We're then using the thenApplyAsync() method to transform the result to its length using a separate thread. The thenAccept() method is then used to print the length of the string.

The output of this code will vary depending on the number of available threads in the ForkJoinPool, but it will look something like this:

Thread: ForkJoinPool.commonPool-worker-1  
Thread: ForkJoinPool.commonPool-worker-1  
Length of Hello: 5

2. thenAcceptAsync(): This method is used to consume the result of a task asynchronously, without returning a value. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

thenAcceptAsync(): This method is used to consume the result of a task asynchronously, without returning a value. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

n this example, we’re creating a CompletableFuture that returns the string "Hello". We're then using the thenAcceptAsync() method to print the string in uppercase using a separate thread.

The output of this code will vary depending on the number of available threads in the ForkJoinPool, but it will look something like this:

Thread: ForkJoinPool.commonPool-worker-1  
HELLO

3. runAsync(): This method is used to execute a task asynchronously, without returning a value. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Here's an example:

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {  
 System.out.println("Thread: " + Thread.currentThread().getName());  
 System.out.println("Hello from async task");  
});

In this example, we’re using the runAsync() method to execute a task asynchronously that prints a message to the console using a separate thread.

The output of this code will vary depending on the number of available threads in the ForkJoinPool, but it will look something like this:

Thread: ForkJoinPool.commonPool-worker-1  
Hello from async task

4. thenComposeAsync() is a method in CompletableFuture that allows you to chain multiple asynchronous tasks together in a non-blocking way. This method is used when you have one CompletableFuture object that returns another CompletableFuture object as its result, and you want to execute the second task after the first one has completed.

The thenComposeAsync() method takes a Function object as its argument, which takes the result of the first CompletableFuture object as its input and returns another CompletableFuture object as its result. The second task is executed when the first one completes, and its result is passed to the next stage of the pipeline.

Here is an example of using thenComposeAsync():

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");  
  
CompletableFuture<String> future2 = future1.thenComposeAsync(s -> CompletableFuture.supplyAsync(() -> s + " World"));  
  
future2.thenAccept(result -> System.out.println(result));

In this example, we create a CompletableFuture that returns the string "Hello". We then use the thenComposeAsync() method to chain another CompletableFuture to the first one. The second CompletableFuture takes the result of the first one and adds the string " World" to it. Finally, we use the thenAccept() method to print the result of the second CompletableFuture.

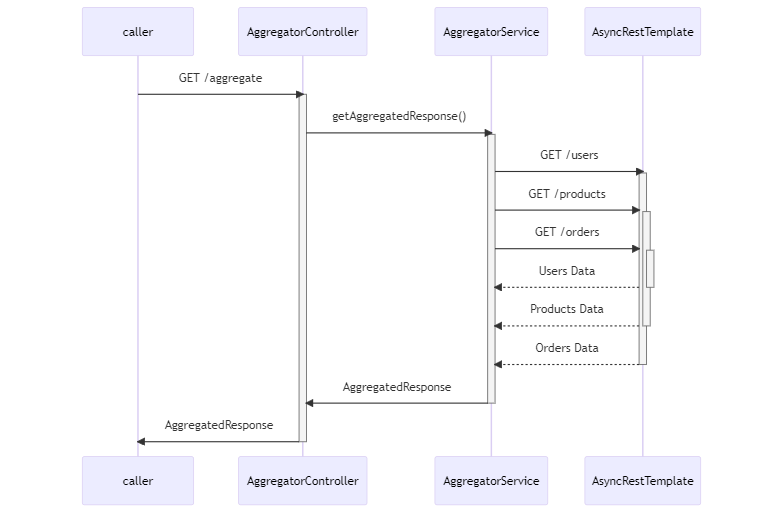
The output of this code will be:

Hello World

In this example, thenComposeAsync() creates a dependent CompletableFuture that takes the result of the first one and applies a transformation to it. The thenAccept() method is then used to consume the result of the dependent CompletableFuture. Note that both tasks are executed asynchronously, so the main thread doesn't block while they are running.

**Example**

We have 3 services asynchronously using RestTemplate and aggregates all the responses before sending the response to the caller. Here’s how you can do it:



sequence diagram showing parallel calls

1. Create a service that uses RestTemplate to call the three endpoints asynchronously. You can use Spring’s AsyncRestTemplate to achieve this. Here's an example of how you can create a service that calls three endpoints and aggregates their responses:

@Service  
public class AggregatorService {  
 @Autowired  
 private AsyncRestTemplate restTemplate;  
  
 public CompletableFuture<AggregatedResponse> getAggregatedResponse() {  
 CompletableFuture<User[]> usersFuture = CompletableFuture.supplyAsync(() -> {  
 return restTemplate.getForObject("http://localhost:8080/users", User[].class);  
 });  
   
 CompletableFuture<Product[]> productsFuture = CompletableFuture.supplyAsync(() -> {  
 return restTemplate.getForObject("http://localhost:8080/products", Product[].class);  
 });  
   
 CompletableFuture<Order[]> ordersFuture = CompletableFuture.supplyAsync(() -> {  
 return restTemplate.getForObject("http://localhost:8080/orders", Order[].class);  
 });  
   
 return CompletableFuture.allOf(usersFuture, productsFuture, ordersFuture)  
 .thenApply(v -> new AggregatedResponse(usersFuture.join(), productsFuture.join(), ordersFuture.join()));  
 }  
}

In the example above, we’re calling three endpoints /users, /products, and /orders asynchronously using AsyncRestTemplate. We're using CompletableFuture to handle the asynchronous calls and aggregate their responses. Once all the responses are received, we're returning an AggregatedResponse object containing the data from all the endpoints.

2. Create a REST endpoint that calls the AggregatorService and returns the aggregated response. Here's an example of how you can create a REST endpoint that calls the AggregatorService:

@RestController  
public class AggregatorController {  
 @Autowired  
 private AggregatorService aggregatorService;  
  
 @GetMapping("/aggregate")  
 public CompletableFuture<AggregatedResponse> getAggregatedResponse() {  
 return aggregatorService.getAggregatedResponse();  
 }  
}

AggregatedResponse class

public class AggregatedResponse {  
 private User[] users;  
 private Product[] products;  
 private Order[] orders;  
  
 //getters and setters  
}

Thanks, before you go:

<https://stackoverflow.com/questions/46130969/difference-between-java8-thencompose-and-thencomposeasync>

https://www.baeldung.com/java-completablefuture