* [Books](https://reflectoring.io/java-completablefuture/)
* [Simplify](https://reflectoring.io/simplify)
* [Contribute](https://reflectoring.io/java-completablefuture/)
* [Services](https://reflectoring.io/java-completablefuture/)
* [Categories](https://reflectoring.io/java-completablefuture/)

Top of Form

Bottom of Form

**Improving Performance with Java’s CompletableFuture**

* February 21, 2022
  + [Java](https://reflectoring.io/categories/java)

Table Of Contents

In this article, we will learn how to use CompletableFuture to increase the performance of our application. We’ll start with looking at the Future interface and its limitations and then will discuss how we can instead use the CompletableFuture class to overcome these limitations.

We will do this by building a simple application that tries to categorize a list of bank Transactions using a remote service. Let’s begin our journey!

**What Is a Future?**

**Future is a Java interface that was introduced in Java 5 to represent a value that will be available in the future**. The advantages of using a Future are enormous because we could do some very intensive computation asynchronously without blocking the current thread that in the meantime can do some other useful job.

We can think of it as going to the restaurant. During the time that the chef is preparing our dinner, we can do other things, like talking to friends or drinking a glass of wine and once the chef has finished the preparation, we can finally eat. Another advantage is that using the Future interface is much more developer-friendly than working directly with threads.

**CompletableFuture vs. Future**

In this section we will look at some limitations of the Future interface and how we can solve these by using the CompletableFuture class.

**Defining a Timeout**

The Future interface provides only the get() method to retrieve the result of the computation, but **if the computation takes too long we don’t have any way to complete it by returning a value that we can assign**.

To understand better, let’s look at some code:

class Demo {

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

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String> stringFuture = executor.submit(() -> neverEndingComputation());

System.out.println("The result is: " + stringFuture.get());

}

}

We have created an instance of ExecutorService that we will use to submit a task that never ends - we call it neverEndingComputation().

After that we want to print the value of the stringFuture variable on the console by invoking the get() method. This method waits if necessary for the computation to complete, and then retrieves its result. But because we are calling neverEndingComputation() that never ends, the result will never be printed on the console, and we don’t have any way to complete it manually by passing a value.

Now let’s see how to overcome this limitation by using the class CompletableFuture. We will use the same scenario, but in this case, we will provide our value by using the method complete() of the CompletableFuture class.

class Demo {

public static void main(String[] args) {

CompletableFuture<String> stringCompletableFuture = CompletableFuture.supplyAsync(() -> neverEndingComputation());

stringCompletableFuture.complete("Completed");

System.out.println("Is the stringCompletableFuture done ? " + stringCompletableFuture.isDone());

}

}

Here we are creating a CompletableFuture of type String by calling the method supplyAsync() which takes a Supplier as an argument.

In the end, we are testing if stringCompletableFuture really has a value by using the method isDone() which returns true if completed in any fashion: normally, exceptionally, or via cancellation. The output of the main() method is:

Is the stringCompletableFuture done ? true

**Combining Asynchronous Operations**

Let’s imagine that we need to call two remote APIs, firstApiCall() and secondApiCall(). The result of the first API will be the input for the second API. By using the Future interface there is no way to combine these two operations asynchronously:

class Demo {

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

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String> firstApiCallResult = executor.submit(

() -> firstApiCall(someValue)

);

String stringResult = firstApiCallResult.get();

Future<String> secondApiCallResult = executor.submit(

() -> secondApiCall(stringResult)

);

}

}

In the code example above, we call the first API by submitting a task on the ExecutorService that returns a Future. We need to pass this value to the second API, but the only way to retrieve the value is by using the get() of the Future method that we have discussed earlier, and by using it we block the main thread. Now we have to wait until the first API returns the result before doing anything else.

By using the CompletableFuture class we don’t need to block the main thread anymore, but we can asynchronously combine more operations:

class Demo {

public static void main(String[] args) {

var finalResult = CompletableFuture.supplyAsync(

() -> firstApiCall(someValue)

)

.thenApply(firstApiResult -> secondApiCall(firstApiResult));

}

}

We are using the method supplyAsync() of the CompletableFuture class which returns a new CompletableFuture that is asynchronously completed by a task running in the ForkJoinPool.commonPool() with the value obtained by calling the given Supplier. After that we are taking the result of the firstApiCall() and using the method thenApply(), we pass it to the other API invoking secondApiCall().

**Reacting to Completion Without Blocking the Thread**

Using the Future interface we don’t have a way to react to the completion of an operation asynchronously. The only way to get the value is by using the get() method which blocks the thread until the result is returned:

class Demo {

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

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String> stringFuture = executor.submit(() -> "hello future");

String uppercase = stringFuture.get().toUpperCase();

System.out.println("The result is: " + uppercase);

}

}

The code above creates a Future by returning a String value. Then we transform it to uppercase by firstly calling the get() method and right after the toUpperCase() method of the String class.

Using CompletableFuture we can now create a pipeline of asynchronous operations. Let’s see a simple example of how to do it:

class Demo {

public static void main(String[] args) {

CompletableFuture.supplyAsync(() -> "hello completable future")

.thenApply(String::toUpperCase)

.thenAccept(System.out::println);

}

}

In the example above we can notice how simple is to create such a pipeline. First, we are calling the supplyAsync() method which takes a Supplier and returns a new CompletableFuture. Then we are then transforming the result to an uppercase string by calling thenApply() method. In the end, we just print the value on the console using thenAccept() that takes a Consumer as the argument.

If we step back for a moment, we realize that working with CompletableFuture is very similar to Java Streams.

**Performance Gains with CompletableFuture**

In this section we will build a simple application that takes a list of bank transactions and calls an external service to categorize each transaction based on the description. We will simulate the call of the external service by using a method that adds some delay before returning the category of the transaction. In the next sections we will incrementally change the implementation of our client application to improve the performance by using CompletableFuture.

**Synchronous Implementation**

Let’s start implementing our categorization service that declares a method called categorizeTransaction :

public class CategorizationService {

public static Category categorizeTransaction(Transaction transaction) {

delay();

return new Category("Category\_" + transaction.getId());

}

public static void delay() {

try {

Thread.sleep(1000L);

} catch (InterruptedException e) {

throw new RuntimeException(e);

}

}

}

public class Category {

private final String category;

public Category(String category) {

this.category = category;

}

@Override

public String toString() {

return "Category{" +

"category='" + category + '\'' +

'}';

}

}

public class Transaction {

private String id;

private String description;

public Transaction(String id, String description) {

this.id = id;

this.description = description;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

}

In the code above we have a class called Transaction that has an id and a description field.

We will pass an instance of this class to the static method categorizeTransaction(Transaction transaction) of our CategorizationService which will return an instance of the class Category.

Before returning the result, the categorizeTransaction() method waits for one second and then returns a Category object that has field of type String called description. The description field will be just the concatenation of the String "Category\_" with the id field from the Transaction class.

To test this implementation we will build a client application that tries to categorize three transactions, as follows :

public class Demo {

public static void main(String[] args) {

long start = System.currentTimeMillis();

var categories = Stream.of(

new Transaction("1", "description 1"),

new Transaction("2", "description 2"),

new Transaction("3", "description 3"))

.map(CategorizationService::categorizeTransaction)

.collect(Collectors.toList());

long end = System.currentTimeMillis();

System.out.printf("The operation took %s ms%n", end - start);

System.out.println("Categories are: " + categories);

}

}

After running the code, it prints on the console the total time taken to categorize the three transactions, and on my machine it is saying :

The operation took 3039 ms

Categories are: [Category{category='Category\_1'},

Category{category='Category\_2'},

Category{category='Category\_3'}]

The program takes 3 seconds to complete because we are categorizing each transaction in sequence and the time needed to categorize one transaction is one second. In the next section, we will try to refactor our client application using a parallel stream.

**Parallel Stream Implementation**

Using a parallel stream, our client application will look like this:

public class Demo {

public static void main(String[] args) {

long start = System.currentTimeMillis();

var categories = Stream.of(

new Transaction("1", "description 1"),

new Transaction("2", "description 2"),

new Transaction("3", "description 3"))

.parallel()

.map(CategorizationService::categorizeTransaction)

.collect(Collectors.toList());

long end = System.currentTimeMillis();

System.out.printf("The operation took %s ms%n", end - start);

System.out.println("Categories are: " + categories);

}

}

It’s almost identical to before, apart from that here we are using the parallel() method to parallelize the computation. If we run this program now, it will print the following output:

The operation took 1037 ms

Categories are: [Category{category='Category\_1'},

Category{category='Category\_2'},

Category{category='Category\_3'}]

The difference is huge! Now our application runs almost three times faster, but this is not the whole story.

This solution can scale until we reach the limit of the number of processors. After that the performance doesn’t change because internally the parallel stream uses a Thread pool that has a fixed number of threads that is equal to Runtime.getRuntime().availableProcessors().

In my machine, I have 8 processors, so if we run the code above with ten transactions it should take at least 2 seconds:

The operation took 2030 ms

Categories are: [Category{category='Category\_1'},

Category{category='Category\_2'},

Category{category='Category\_3'},

Category{category='Category\_4'},

Category{category='Category\_5'},

Category{category='Category\_6'},

Category{category='Category\_7'},

Category{category='Category\_8'},

Category{category='Category\_9'},

Category{category='Category\_10'}]

We see that the operation took 2030 ms, as predicted. Can we do something to increase the performance of our application even more? YES!

**Increasing Performance Using CompletableFuture**

Now will refactor our client application to take advantage of CompletableFuture:

public class Demo {

public static void main(String[] args) {

Executor executor = Executors.newFixedThreadPool(10);

long start = System.currentTimeMillis();

var futureCategories = Stream.of(

new Transaction("1", "description 1"),

new Transaction("2", "description 2"),

new Transaction("3", "description 3"),

new Transaction("4", "description 4"),

new Transaction("5", "description 5"),

new Transaction("6", "description 6"),

new Transaction("7", "description 7"),

new Transaction("8", "description 8"),

new Transaction("9", "description 9"),

new Transaction("10", "description 10")

)

.map(transaction -> CompletableFuture.supplyAsync(

() -> CategorizationService.categorizeTransaction(transaction), executor)

)

.collect(toList());

var categories = futureCategories.stream()

.map(CompletableFuture::join)

.collect(toList());

long end = System.currentTimeMillis();

System.out.printf("The operation took %s ms%n", end - start);

System.out.println("Categories are: " + categories);

}

}

Our client application is trying to call the categorization service by using the method supplyAsync() that takes as arguments a Supplier and an Executor. Here we can now pass a custom Executor with a pool of ten threads to make the computation finish even faster than before.

With 10 threads, we expect that the operation should take around one second. Indeed, the output confirms the expected result :

The operation took 1040 ms

Categories are: [Category{category='Category\_1'},

Category{category='Category\_2'},

Category{category='Category\_3'},

Category{category='Category\_4'},

Category{category='Category\_5'},

Category{category='Category\_6'},

Category{category='Category\_7'},

Category{category='Category\_8'},

Category{category='Category\_9'},

Category{category='Category\_10'}]

**Conclusion**

In this article, we learned how to use the Future interface in Java and its limitations. We learned how to overcome these limitations by using the CompletableFuture class. After that, we analyzed a demo application, and step by step using the potential offered by CompletableFuture we refactored it for better performance.

Written By:

**Bledi Nukaj**

Bledi is a senior software Engineer and Architect, specialized in Java and Spring to deliver high-quality Enterprise applications, on-premise or cloud. Experienced and keenly interested in the field of Cloud Architecture.

**Simplify! Newsletter**

*"You can't just****keep****it simple.****Make****it simple, then it's easy."*

Join more than 6,000 software engineers to get **exclusive productivity and growth tips** directly to your inbox.

[Check out the Wall of Love!](https://reflectoring.io/simplify#wall-of-love)

Top of Form

Subscribe

Bottom of Form

No spam. Your email address is safe with us. Your data will be used according to the [privacy policy](https://reflectoring.io/privacy).

Join more than 1200 [happy readers](https://stratospheric.dev/#testimonials?utm_source=reflectoring&utm_content=sidebar).

**Save $12.00** by joining the [Stratospheric newsletter](https://stratospheric.dev/#mailing-list?utm_source=reflectoring&utm_content=sidebar).

**[Learn Spring Boot & AWS](https://leanpub.com/stratospheric?utm_source=reflectoring&utm_content=sidebar)**

Join more than 6,500 [happy readers](https://reflectoring.io/book/#what-readers-say).

     more than 150 reviews on [Amazon](https://www.amazon.com/gp/product/1839211962/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1839211962&linkCode=as2&tag=reflectorin0c-20&linkId=559e54b6599c4213252259df28d1d3e3) and [Goodreads](https://www.goodreads.com/book/show/49238827-get-your-hands-dirty-on-clean-architecture).

**Save $10** by joining the [Simplify! newsletter](https://reflectoring.io/simplify).

[**Get Your Hands Dirty**](https://gum.co/gyhdoca)

**Recent Posts**

* [Node](https://reflectoring.io/categories/node)

[**Publish and Receive Messages with Google Pub/Sub in Node.js**](https://reflectoring.io/google-pub-sub-in-node-js/)

* [Olaoluwa Ajibade](https://reflectoring.io/authors/ajibade)
* January 12, 2024

Google Cloud Pub/Sub employs a publish-subscribe model, streamlining modern software communication and reshaping how information is shared. Publishers send messages to topics, and subscribers interested in those messages can retrieve them flexibly and asynchronously.

[**Read More**](https://reflectoring.io/google-pub-sub-in-node-js/)

* [Kotlin](https://reflectoring.io/categories/kotlin)

[**High-Order Functions vs. Inline Functions in Kotlin**](https://reflectoring.io/kotlin-high-order-vs-inline-functions/)

* [Ezra Kanake](https://reflectoring.io/authors/ezra)
* January 10, 2024

One of Kotlin’s standout features is its robust support for functions, including high-order functions and inline functions. In this blog post, we will delve into the world of functions in Kotlin, exploring the differences between high-order functions and inline functions and understanding when to leverage each for optimal code efficiency.

[**Read More**](https://reflectoring.io/kotlin-high-order-vs-inline-functions/)

* [Kotlin](https://reflectoring.io/categories/kotlin)

[**Design Patterns in Kotlin**](https://reflectoring.io/kotlin-design-patterns/)

* [Ezra Kanake](https://reflectoring.io/authors/ezra)
* January 2, 2024

A design pattern is a general repeatable solution to a commonly occurring problem in software design. In this blog post, we will delve into various design patterns and explore how they can be effectively implemented in Kotlin.

[**Read More**](https://reflectoring.io/kotlin-design-patterns/)

Where the HOW meets the WHY.

**Content**

* [Spring Boot](https://reflectoring.io/categories/spring-boot/)
* [Java](https://reflectoring.io/categories/java/)
* [Node](https://reflectoring.io/categories/node/)
* [AWS](https://reflectoring.io/categories/aws/)
* [Software Craft](https://reflectoring.io/categories/software-craft/)
* [Simplify!](https://reflectoring.io/categories/simplify/)
* [Meta](https://reflectoring.io/categories/meta/)
* [Book Reviews](https://reflectoring.io/categories/book-notes/)

**Products**

* [Get Your Hands Dirty on Clean Architecture](https://reflectoring.io/book)
* [Simplify! Newsletter](https://reflectoring.io/simplify)
* [Stratospheric](https://stratospheric.dev/?utm_source=reflectoring&utm_content=sidebar)

**Contribute**

* [Become an Author](https://reflectoring.io/contribute/become-an-author)
* [Writing Guide](https://reflectoring.io/contribute/writing-guide)
* [Author Workflow](https://reflectoring.io/contribute/author-workflow)
* [Author Payment](https://reflectoring.io/contribute/author-payment)

**Reflectoring**

* [About](https://reflectoring.io/about)
* [Atom Feed](https://reflectoring.io/feed.xml)
* [Advertise](https://reflectoring.io/advertise)
* [Book me](https://reflectoring.io/book-me)
* [Privacy](https://reflectoring.io/privacy)

Built upon [Geeky Hugo](https://statichunt.com/themes/hugo-geeky) theme by [Statichunt](https://statichunt.com/)