



Shallow vs. Deep Copy in Java

by Marcus Biel MVB · Apr. 04, 17 · Java Zone

Download Microservices for Java Developers: A hands-on introduction to frameworks and containers. Brought to you in partnership with Red Hat.

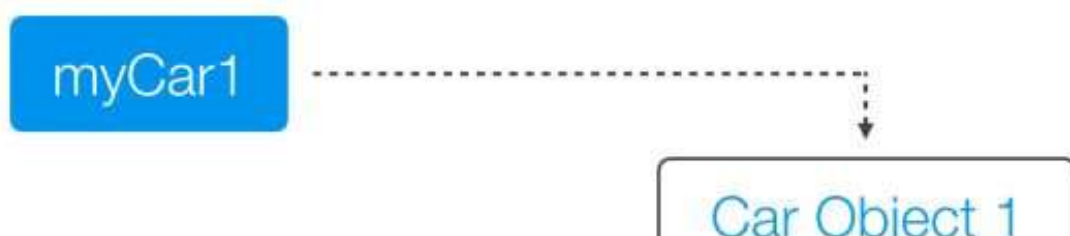
In this article from my free Java 8 Course, I will be discussing the difference between a **Deep** and a **Shallow Copy**. You can download the slides and the article as PDF here.

Shallow vs Deep Copy in Java



What Is a Copy?

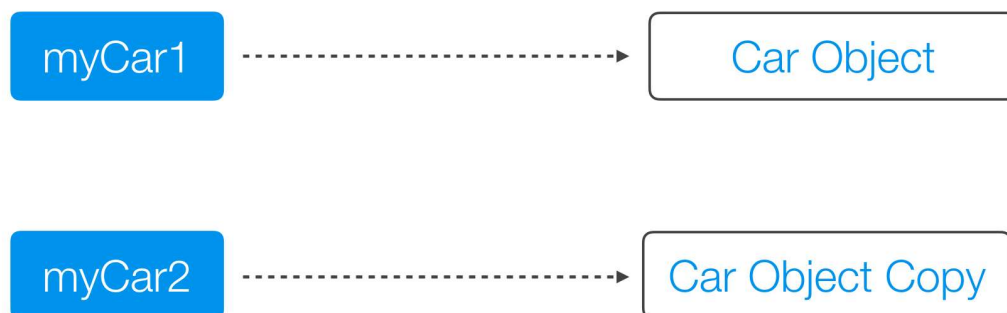
To begin, I'd like to highlight what a copy in Java is. First, let's differentiate between a reference copy and an object copy. A **reference copy**, as the name implies, creates a copy of a reference variable pointing to an object. If we have a Car object, with a *myCar* variable pointing to it and we make a reference copy, we will now have two *myCar* variables, but still one object.





Example 1

An **object copy** creates a copy of the object itself. So if we again copied our *car* object, we would create a copy of the object itself, as well as a second reference variable referencing that copied object.



Example 2

What Is an Object?

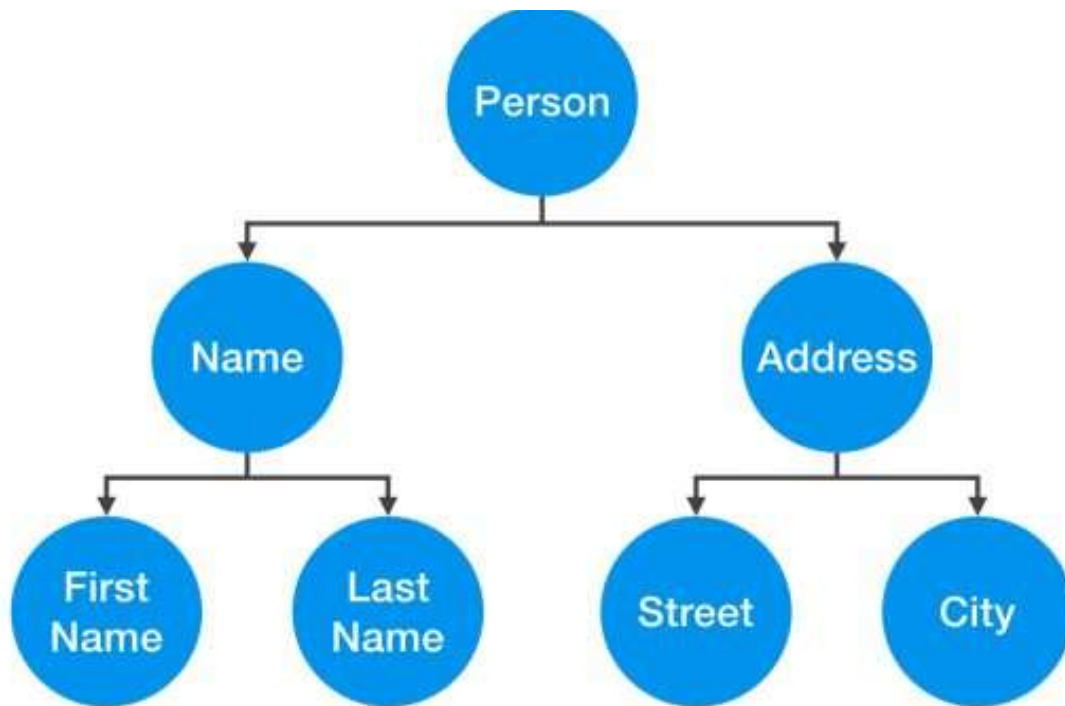
Both a Deep Copy and a Shallow Copy are types of object copies, but what really is an object? Often, when we talk about an object, we speak of it as a single unit that can't be broken down further, like a humble coffee bean. However, that's oversimplified.



Object

Example 3

Say we have a *Person* object. Our *Person* object is in fact composed of other objects, as you can see in Example 4. Our *Person* contains a *Name* object and an *Address* object. The *Name* in turn, contains a *FirstName* and a *LastName* object; the *Address* object is composed of a *Street* object and a *City* object. So when I talk about *Person* in this article, I'm actually talking about this *entire network of objects*.



Example 4

So why would we want to copy this *Person* object? An object copy, usually called a clone, is created if we want to modify or move an object, while still preserving the original object. There are many different ways to copy an object that you can learn about in another article. In this article we'll specifically be using a copy constructor to create our copies.

Shallow Copy

First let's talk about the shallow copy. A shallow copy of an object copies the 'main' object, but doesn't copy the inner objects. The 'inner objects' are shared between the original object and its copy. For example, in our *Person* object, we would create a second *Person*, but both objects would share the same *Name* and *Address* objects.

Let's look at a coding example. In Example 5, we have our class *Person*, which contains a *Name* and *Address* object. The copy constructor takes the *originalPerson* object and copies its reference variables.

```

1  public class Person {
2      private Name name;
3      private Address address;
4
5      public Person(Person originalPerson) {
6          this.name = originalPerson.name;
7          this.address = originalPerson.address;
8      }
9      [...]
10 }
```

Example 5

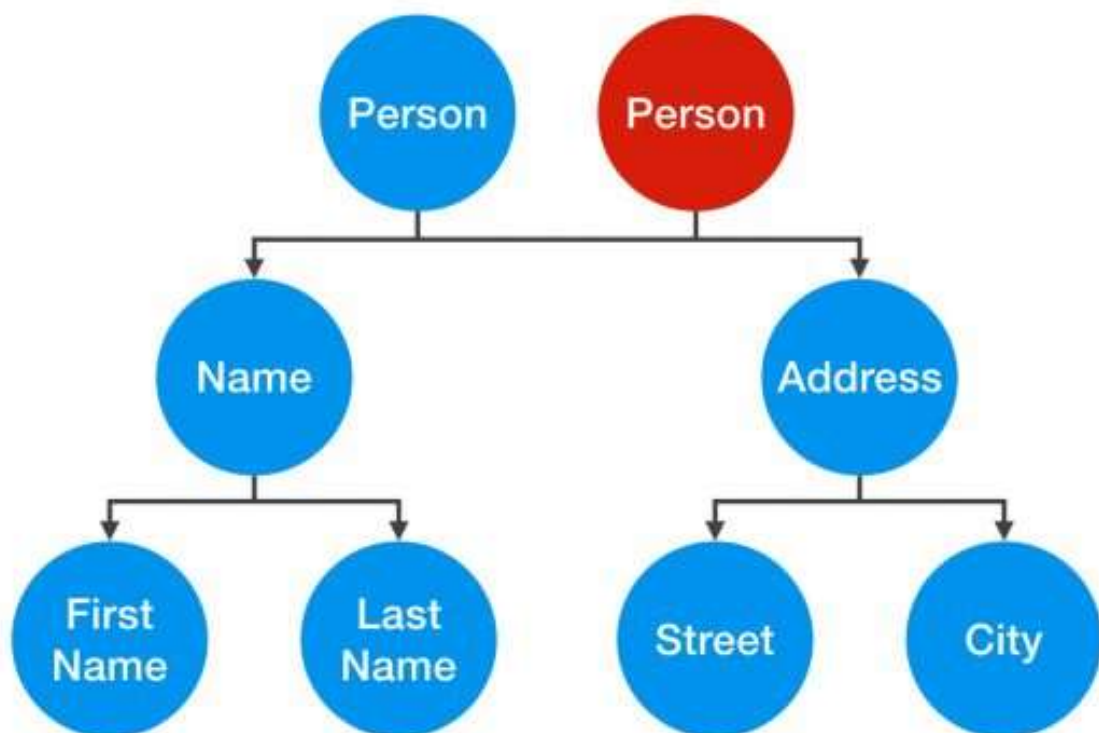
The problem with the shallow copy is that the two objects are not independent. If you modify the Name object of one *Person*, the change will be reflected in the other Person object.

Let's apply this to an example. Say we have a Person object with a reference variable mother; then, we make a copy of *mother*, creating a second *Person* object, *son*. If later on in the code, the son tries to *moveOut()* by modifying his *Address* object, the *mother* moves with him!

```
1 Person mother = new Person(new Name(...), new Address(...));  
2 [...]  
3 Person son = new Person(mother);  
4 [...]  
5 son.moveOut(new Street(...), new City(...));
```

Example 6

This occurs because our *mother* and *son* objects share the same *Address* object, as you can see illustrated in Example 7. When we change the *Address* in one object, it changes in both!

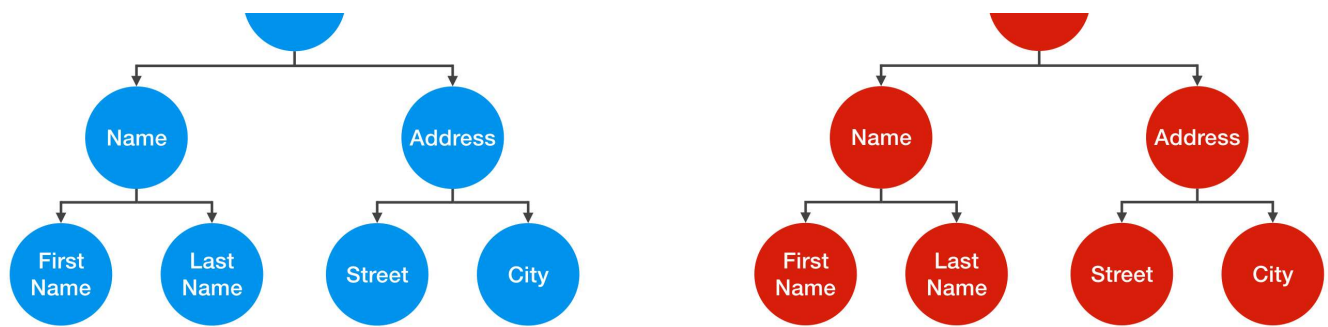


Example 7

Deep Copy

Unlike the shallow copy, a deep copy is a **fully independent copy of an object**. If we copied our Person object, we would copy the entire object structure.



*Example 8*

A change in the *Address* object of one *Person* wouldn't be reflected in the other object as you can see by the diagram in Example 8. If we take a look at the code in example 9, you can see that we're not only using a copy constructor on our *Person* object, but we are also utilizing copy constructors on the inner objects as well.

```

1  public class Person {
2      private Name name;
3      private Address address;
4
5      public Person(Person otherPerson) {
6          this.name = new Name(otherPerson.name);
7          this.address = new Address(otherPerson.address);
8      }
9      [...]
10 }

```

Example 9

Using this deep copy, we can retry the mother-son example from Example 6. Now the *son* is able to successfully move out!

However, that's not the end of the story. To create a true deep copy, we need to keep copying all of the *Person* object's nested elements, until there are only primitive types and "**Immutable**s" left. Let's look at the *Street* class to better illustrate this:

```

1  public class Street {
2      private String name;
3      private int number;
4
5      public Street(Street otherStreet){
6          this.name = otherStreet.name;
7          this.number = otherStreet.number;
8      }
9      [...]
10 }

```

Example 10

The *Street* object is composed of two instance variables – *String name* and *int number*. *int number* is

a primitive value and not an object. It's just a simple value that can't be shared, so by creating a second instance variable, we are automatically creating an independent copy. *String* is an Immutable. In short, an Immutable is an Object, that, once created, can never be changed again. Therefore, you can share it without having to create a deep copy of it.

Conclusion

To conclude, I'd like to talk about some coding techniques we used in our mother-son example. Just because a deep copy will let you change the internal details of an object, such as the *Address* object, it doesn't mean that you should. Doing so **would decrease code quality**, as it would make the *Person* class more fragile to changes – whenever the *Address* class is changed, you will have to (potentially) apply changes to the *Person* class also. For example, if the *Address* class no longer contains a *Street* object, we'd have to change the *moveOut()* method in the *Person* class on top of the changes we already made to the *Address* class.

In Example 6 of this article I only chose to use a new *Street* and *City* object to better illustrate the difference between a shallow and a deep copy. Instead, I would recommend that you assign a new *Address* object instead, effectively converting to a **hybrid** of a shallow and a deep copy, as you can see in Example 10:

```
1 Person mother = new Person(new Name(...), new Address(...));
2 [...]
3 Person son = new Person(mother);
4 [...]
5 son.moveOut(new Address(...));
```

Example 11

In object-oriented terms, this violates **encapsulation**, and therefore should be avoided. Encapsulation is **one of the most important aspects of Object Oriented programming**. In this case, I had violated encapsulation by accessing the internal details of the *Address* object in our *Person* class. This harms our code because we have now entangled the *Person* class in the *Address* class and if we make changes to the *Address* class down the line, it could harm the *Person* class as I explained above. While you obviously need to interconnect your various classes to have a coding project, whenever you connect two classes, you need to analyze the costs and benefits.

Download Building Reactive Microservices in Java: Asynchronous and Event-Based Application Design. Brought to you in partnership with Red Hat.

Like This Article? Read More From DZone



Shallow and Deep Java Cloning



Reduce Boilerplate Java Using try-with-resources




Using Java 8 CompletableFuture and Rx-Java Observable



Free DZone Refcard
Reactive Microservices With
Lagom and Java

Topics: JAVA , JAVA 8 , DEEP COPY , SHALLOW COPY

Published at DZone with permission of Marcus Biel, DZone MVB. [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

Java Partner Resources

A Better Pull Request

Atlassian



Improve Code Quality with this Tutorial

Atlassian



Reactive Microservices Architecture: Design Principles for Distributed Systems

Lightbend




A Practical Guide to Learning Git

Atlassian



Concurrency: Java Futures and Kotlin Coroutines

by Nicolas Frankel  MVB · Jun 29, 17 · Java Zone

Find your next Integration job at DZone Jobs. See jobs focused on integration, or create your profile and have the employers come to you!

This article is featured in the new DZone Guide to Java: Development and Evolution. Get your free copy for more insightful articles, industry statistics, and more!

A long time ago, one had to manually start new threads to run code concurrently in Java. Not only was this hard to write, it also was easy to introduce bugs that were hard to find. Testing, reading, and maintaining such code was no walk in the park, either. Since that time, and with a little incentive coming from multi-core machines, the Java API has evolved to make developing concurrent code easier. Meanwhile, alternative JVM languages also have their opinion about helping developers write such code. In this post, I'll compare how it's implemented in Java and Kotlin.

To keep the article focused, I deliberately left out performance to write about code readability.

About the Use Case

The use case is not very original. We need to call different web services. The naïve solution would be to call them sequentially, one after the other, and collect the result of each of them. In that

case, the overall call time would be the sum of the call time of each service. An easy improvement is to call them in parallel and wait for the last one to finish. Thus, performance improves from linear to constant — or for the more mathematically inclined, from $O(n)$ to $O(1)$.

To simulate the calling of a web service with a delay, let's use the following code (in Kotlin, because this is so much less verbose):

```
1 class DummyService(private val name: String) {
2     private val random = SecureRandom()
3     val content: ContentDuration
4     get() {
5         val duration = random.nextInt(5000)
6         Thread.sleep(duration.toLong())
7         return ContentDuration(name, duration)
8     }
9 }
10 data class ContentDuration(val content: String, val duration:
11 Int)
```

The Java Future API

Java offers a whole class hierarchy to handle concurrent calls. It's based on the following classes:

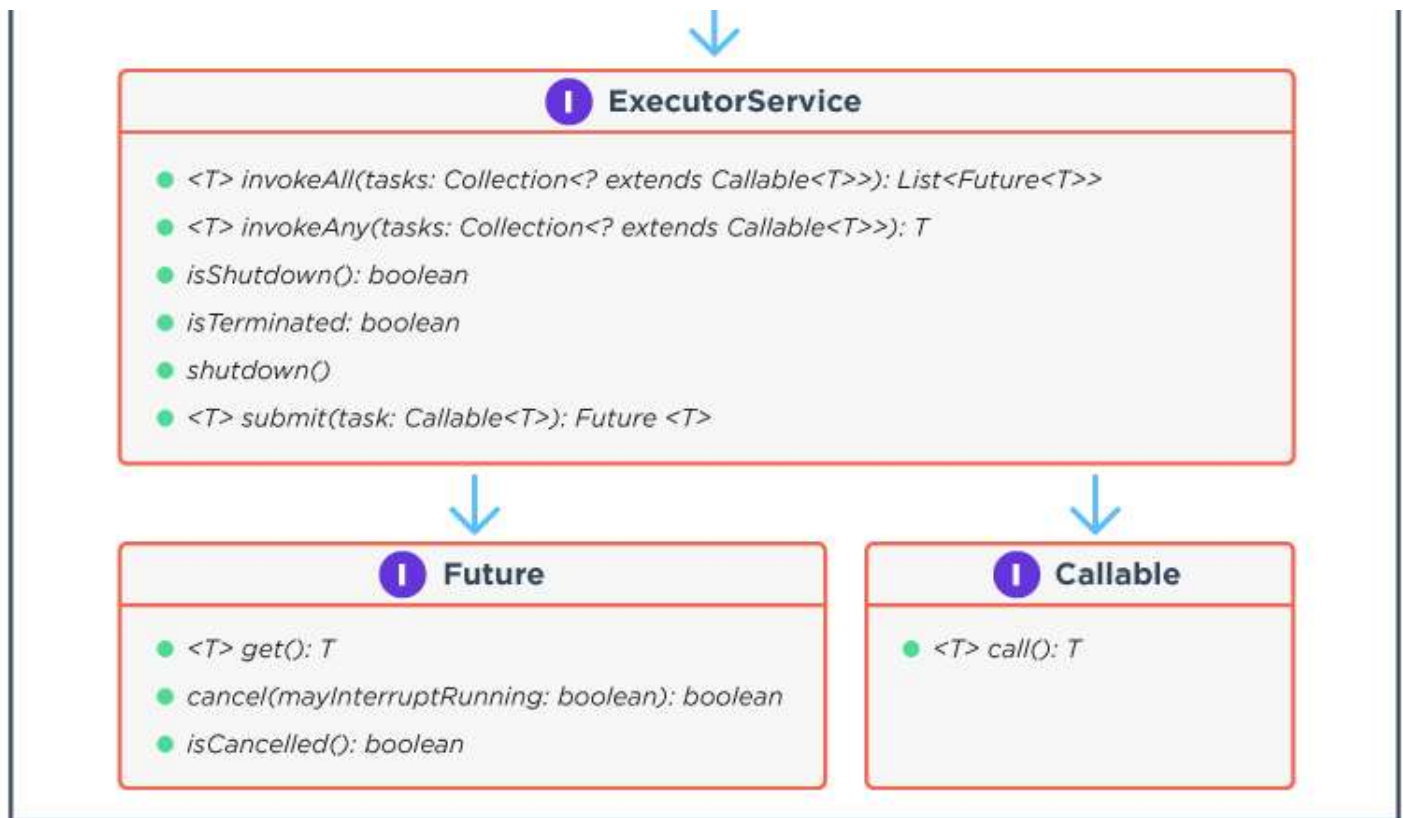
Callable: A Callable is a “task that returns a result.” From another viewpoint, it's similar to a function that takes no parameter and returns this result.

Future: A Future is “the result of an asynchronous computation.” Also, “the result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready.” In other words, it represents a wrapper around a value, where this value is the outcome of a calculation.

Executor Service: An `ExecutorService` “provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.” It is the entry point into concurrent handling code in Java. Implementations of this interface, as well as more specialized ones, can be obtained through static methods in the `Executors` class.

This is summarized in the diagram below:





Calling our services using the concurrent package is a two-step process.

Creating a Collection of Callables

First, there needs to be a collection of `Callable` to pass to the executor service. This is how it might go:

1. Form a stream of service names.
2. For each service name, create a new dummy service initialized with the string.
3. For every service, return the service's `getContent()` method reference as a `Callable`. This works because the method signature matches `Callable.call()` and `Callable` is a functional interface.

This is the preparation phase. It translates into the following code:

```

1 List<Callable<ContentDuration>> callables = Stream.
2   of("Service A", "Service B", "Service C")
3     .map(DummyService::new)
4     .map(service -> (Callable<ContentDuration>)
5       service::getContent)
6     .collect(Collectors.toList());
  
```

Processing the Callables

Once the list has been prepared, it's time for the `ExecutorService` to process it, AKA the "real work."

1. Create a new executor service — any will do.

2. Pass the list of Callable to the executor service. and stream the resulting list of Future.
3. For every future, either return the result or handle the exception.

The following snippet is a possible implementation:

```
1  ExecutorService executor = Executors.newWorkStealingPool();
2  List < ContentDuration > results = executor.
3  invokeAll(callables).stream()
4      .map(future - > {
5          try {
6              return future.get();
7          } catch (InterruptedException | ExecutionException e) {
8              throw new RuntimeException(e);
9          }
10     }).collect(Collectors.toList());
```

The Future API, But in Kotlin

Let's face it: While Java makes it possible to write concurrent code, reading and maintaining it is not that easy, mainly due to:

- Going back and forth between collections and streams.
- Handling checked exceptions in lambdas.
- Casting explicitly.

```
1  var callables: List < Callable < ContentDuration >> =
2      arrayOf("Service A", "Service B", "Service C")
3      .map {
4          DummyService(it)
5      }
6      .map {
7          Callable < ContentDuration > {
8              it.content
9          }
10     }
11  val executor = Executors.newWorkStealingPool()
12  val results = executor.invokeAll(callables).map {
13      it.get()
14  }
```

Kotlin Coroutines

With version 1.1 of Kotlin comes a new experimental feature called coroutines. From the Kotlin documentation:

“Basically, coroutines are computations that can be suspended without blocking a thread. Blocking threads is often expensive, especially under high load [...]. Coroutine suspension is almost free, on the other hand. No context switch or any other involvement of the OS is required.”

The leading design principle behind coroutines is that they must feel like sequential code but run like concurrent code. They are based on the diagram here. Nothing beats the code itself, though. Let's implement the same as above, but with coroutines in Kotlin instead of Java futures.

As a pre-step, let's just extend the service to ease further processing by adding a new computed property wrapped around content of type Deferred:

```
1 val DummyService.asyncContent: Deferred<ContentDuration>
2     get() = async(CommonPool) { content }
```

This is standard Kotlin extension property code, but notice the CommonPool parameter. This is the magic that makes the code run concurrently. It's a companion object (i.e. a singleton) that uses a multi-fallback algorithm to get an ExecutorService instance.

Now, onto the code flow proper:

1. Coroutines are handled inside a block. Declare a variable list outside the block to be assigned inside it.
2. Open the synchronization block.
3. Create the array of service names.
4. For each name, create a service and return it.
5. For each service, get its async content (declared above) and return it.
6. For each deferred, get the result and return it.

```
1 // Variable must be initialized or the compiler complains
2 // And the variable cannot be used afterwards
3 var results: List<ContentDuration>? = null
4 runBlocking {
5     results = arrayOf("Service A", "Service B", "Service C")
6         .map { DummyService(it) }
7         .map { it.asyncContent }
```

```
8         .map { it.await() }  
9     }
```

Takeaways

The Future API is not so much a problem than the Java language itself is. As soon as the code is translated into Kotlin, the readability significantly improves. Yet having to create a collection to pass to the executor service breaks the nice functional pipeline. For coroutines, the only compromise is to move from a var to a val to get the final results (or to add the results to a mutable list).

Also, remember that coroutines are still experimental. Despite all of that, the code does look sequential — and is thus more readable and behaves in parallel. The complete source code for this post can be found on GitHub in Maven format.

This article is featured in the new DZone Guide to Java: Development and Evolution. Get your free copy for more insightful articles, industry statistics, and more!

Find your next Integration job at DZone Jobs. See jobs focused on integration, or create your profile and have the employers come to you!

Like This Article? Read More From DZone



Concurrency: Java Futures and Kotlin Coroutines



Communities and Criticism



Synchronized java.util.HashMap vs. java.util.concurrent.ConcurrentHashMap



Free DZone Refcard Reactive Microservices With Lagom and Java

Topics: JAVA, JAVA FUTURE, KOTLIN COROUTINES, CONCURRENCY

Opinions expressed by DZone contributors are their own.
