Java 8: Definitive guide to CompletableFuture

Java 8 is coming so it's time to study new features. While Java 7 and Java 6 were rather minor releases, version 8 will be a big step forward. Maybe even too big? Today I will give you a thorough explanation of new abstraction in JDK 8 - `CompletableFuture<T>` [http://download.java.net/lambda/b88/docs/api/java/util/concurrent/CompletableFuture.html] . As you all know Java 8 will hopefully be released in less than a year, therefore this article is based on JDK 8 build 88 with lambda support [https://jdk8.java.net/lambda/] . `CompletableFuture<T>` [http://download.java.net/lambda/b88/docs/api/java/util/concurrent/CompletableFuture.html] extends `Future<T>` [http://download.java.net/lambda/b88/docs/api/java/util/concurrent/Future.html] by providing functional, monadic (!) operations and promoting asynchronous, event-driven programming model, as opposed to blocking in older Java. If you opened JavaDoc of `CompletableFuture<T>` [http://download.java.net/lambda/b88/docs/api/java/util/concurrent/CompletableFuture.html] you are surely overwhelmed. About **fifty methods** (!), some of them being extremely cryptic and exotic, e.g.:

```
1 public <U,V> CompletableFuture<V> thenCombineAsync(
2     CompletableFuture<? extends U> other,
3     BiFunction<? super T,? super U,? extends V> fn,
4     Executor executor)
```

Don't worry, but keep reading. `CompletableFuture` collects all the features of `ListenableFuture` in Guava [http://nurkiewicz.blogspot.no/2013/02/listenablefuture-in-guava.html] with `SettableFuture` [http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/util/concurrent/SettableFuture.html] . Moreover built-in lambda support brings it closer to Scala/Akka futures [http://nurkiewicz.blogspot.no/2013/03/futures-in-akka-with-scala.html] . Sounds too good to be true, but keep reading. `CompletableFuture` has two major areas superior to good ol' `Future<T>` - asynchronous callback/transformations support and the ability to set value of `CompletableFuture` from any thread at any point in time.

Extract/modify wrapped value

Typically futures represent piece of code running by other thread. But that's not always the case. Sometimes you want to create a `Future` representing some event that you know will occur, e.g. JMS message arrival [http://nurkiewicz.blogspot.no/2013/03/deferredresult-asynchronous-processing.html] . So you have `Future<Message>` but there is no asynchronous job underlying this future. You simply want to complete (resolve) that future when JMS message arrives, and this is driven by an event. In this case you can simply create `CompletableFuture`, return it to your client and whenever you think your results are available, simply `complete()` the future and unlock all clients waiting on that future.

For starters you can simply create new `CompletableFuture` out of thin air and give it to your client:

```
public CompletableFuture<String> ask() {
    final CompletableFuture<String> future = new Completa
    //...
    return future;
}
```

Notice that this future is not associated wtih any `Callable<String>`, no thread pool, no asynchronous job. If now the client code calls `ask().get()` it will block forever. If it registers some completion callbacks, they will never fire. So what's the point? Now you can say:

```
future.complete("42")
```

...and at this very moment all clients blocked on `Future.get()` will get the result string. Also completion callbacks will fire immediately. This comes quite handy when you want to represent a task in the future, but not necessarily computational task running on some thread of execution. `CompletableFuture.complete()` can only be called once, subsequent invocations are ignored. But there is a back-door called `CompletableFuture.obtrudeValue(...)` which overrides previous value of the `Future` with new one. Use with caution.

Sometimes you want to signal failure. As you know `Future` objects can handle either wrapped result or exception. If you want to pass some exception further, there is `CompletableFuture.completeExceptionally(ex)` (and `obtrudeException(ex)` evil brother that overrides the previous exception). `completeExceptionally()` also unlock all waiting clients, but this time throwing an exception from `get()`. Speaking of `get()`, there is also `CompletableFuture.join()` method with some subtle changes in error handling. But in general they are the same. And finally there is also `CompletableFuture.getNow(valueIfAbsent)` method that doesn't block but if the `Future` is not completed yet, returns default value. Useful when building robust systems where we don't want to wait too much.

Last `static` utility method is `completedFuture(value)` that returns already completed `Future` object. Might be useful for testing or when writing some adapter layer.

Creating and obtaining `CompletableFuture`

OK, so is creating `CompletableFuture` manually our only option? Not quite. Just as with normal `Futures` we can wrap existing task with `CompletableFuture` using the following family of factory methods:

```
static <U> CompletableFuture<U> supplyAsync(Supplier<U> s
static <U> CompletableFuture<U> supplyAsync(Supplier<U> s
static CompletableFuture<Void> runAsync(Runnable runnable
static CompletableFuture<Void> runAsync(Runnable runnable
```

Methods that do not take an `Executor` as an argument but end with `...Async` will use `ForkJoinPool.commonPool()` [http://download.java.net/lambda/b88/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool()] (global, general purpose pool introduces in JDK 8). This applies to most methods in `CompletableFuture` class. `runAsync()` is simple to understand, notice that it takes `Runnable`, therefore it returns `CompletableFuture<Void>` as `Runnable` doesn't return anything. If you need to process something asynchronously and return result, use `Supplier<U>` [http://download.java.net/lambda/b88/docs/api/java/util/function/Supplier.html] :

```
final CompletableFuture<String> future = CompletableFutur
    @Override
    public String get() {
        //...long running...
        return "42";
    }
}, executor);
```

But hey, we have lambdas in Java 8!

```
final CompletableFuture<String> future = CompletableFutur
    //...long running...
    return "42";
}, executor);
```

or even:

```
final CompletableFuture<String> future =
    CompletableFuture.supplyAsync(() -> longRunningTask(p
```

This article is not about project Lambda, but I will be using lambdas quite extensively.

Transforming and acting on one `CompletableFuture` (`thenApply`)

So I said that `CompletableFuture` is superior to `Future` but you haven't yet seen why? Simply put, it's because `CompletableFuture` is a monad and a functor. Not helping I guess? Both Scala [http://nurkiewicz.blogspot.no/2013/03/futures-in-akka-with-scala.html] and JavaScript [http://nurkiewicz.blogspot.no/2013/03/promises-and-futures-in-clojure.html] allow registering asynchronous callbacks when future is completed. We don't have to wait and block until it's ready. We can simply say: *run this function on a result, when it arrives*. Moreover, we can stack such functions, combine multiple futures together, etc. For example if we have a function from `String` to `Integer` we can turn `CompletableFuture<String>` to `CompletableFuture<Integer>` without unwrapping it. This is achieved with `thenApply()` family of methods:

```
<U> CompletableFuture<U> thenApply(Function<? super T,? e
<U> CompletableFuture<U> thenApplyAsync(Function<? super
<U> CompletableFuture<U> thenApplyAsync(Function<? super
```

As stated before `...Async` versions are provided for most operations on `CompletableFuture` thus I will skip them in subsequent sections. Just remember that first method will apply function within the same thread in which the future completed while the remaining two will apply it asynchronously in different thread pool.

Let's see how `thenApply()` works:

```
CompletableFuture<String> f1 = //...
CompletableFuture<Integer> f2 = f1.thenApply(Integer::par
CompletableFuture<Double> f3 = f2.thenApply(r -> r * r *
```

Or in one statement:

```
CompletableFuture<Double> f3 =
    f1.thenApply(Integer::parseInt).thenApply(r -> r * r
```

You see a sequence of transformations here. From `String` to `Integer` and then to `Double`. But what's most important, these transformations are neither executed immediately nor blocking. They are simply remembered and when original `f1` completes they are executed for you. If some of the transformations are time-consuming, you can supply your own `Executor` to run them asynchronously. Notice that this operation is equivalent to monadic `map` in Scala.

Running code on completion (`thenAccept/thenRun`)

```
CompletableFuture<Void> thenAccept(Consumer<? super T> bl
CompletableFuture<Void> thenRun(Runnable action);
```

These two methods are typical "final" stages in future pipeline. They allow you to consume future value when it's ready. While `thenAccept()` provides the final value, `thenRun` executes `Runnable` which doesn't even have access to computed value. Example:

```
future.thenAcceptAsync(dbl -> log.debug("Result: {}", dbl
log.debug("Continuing");
```

`...Async` variants are available as well for both methods, with implicit and explicit executor. I can't emphasize this enough: `thenAccept()`/`thenRun()` methods **do not block** (even without explicit `executor`). Treat them like an event listener/handler that you attach to a future and that will execute some time in the future. `"Continuing"` message will appear immediately, even if `future` is not even close

to completion.

Error handling of single `CompletableFuture`

So far we only talked about result of computation. But what about exceptions? Can we handle them asynchronously as well? Sure!

```
eFuture<String> safe =
.exceptionally(ex -> "We have a problem: " + ex.getMessage());
```

`exceptionally()` takes a function that will be invoked when original future throws an exception. We then have an opportunity to recover by transforming this exception into some value compatible with `Future`'s type. Further transformations of `safe` will no longer yield an exception but instead a `String` returned from supplied function.

A more flexible approach is `handle()` that takes a function receiving either correct result or exception:

```
CompletableFuture<Integer> safe = future.handle((ok, ex)
    if (ok != null) {
        return Integer.parseInt(ok);
    } else {
        log.warn("Problem", ex);
        return -1;
    }
});
```

`handle()` is called always, with either result or exception argument being not-`null`. This is a one-stop catch-all strategy.

Combining two `CompletableFuture` together

Asynchronous processing of one `CompletableFuture` is nice but it really shows its power when multiple such futures are combined together in various ways.

Combining (chaining) two futures (`thenCompose()`)

Sometimes you want to run some function on future's value (when it's ready). But this function returns future as well. `CompletableFuture` should be smart enough to understand that the result of our function should now be used as top-level future, as opposed to `CompletableFuture<CompletableFuture<T>>`. Method `thenCompose()` is thus equivalent to `flatMap` in Scala:

```
<U> CompletableFuture<U> thenCompose(Function<? super T,C
```

`...Async` variations are available as well. Example below, look carefully at the types and the difference between `thenApply()` (`map`) and `thenCompose()` (`flatMap`) when applying a `calculateRelevance()` function returning `CompletableFuture<Double>`:

```
1  CompletableFuture<Document> docFuture = //...
2
3  CompletableFuture<CompletableFuture<Double>> f =
4      docFuture.thenApply(this::calculateRelevance);
5
6  CompletableFuture<Double> relevanceFuture =
7      docFuture.thenCompose(this::calculateRelevance);
8
9  //...
10
11 private CompletableFuture<Double> calculateRelevance(Doc
```

`thenCompose()` is an essential method that allows building robust, asynchronous pipelines, without blocking or waiting for intermediate steps.

Transforming values of two futures (`thenCombine()`)

While `thenCompose()` is used to chain one future dependent on the other, `thenCombine` combines two independent futures when they are both done:

```
1  <U,V> CompletableFuture<V> thenCombine(CompletableFuture<
```

`...Async` variations are available as well. Imagine you have two `CompletableFutures`, one that loads `Customer` and other that loads nearest `Shop`. They are completely independent from each other, but when both of them are completed, you want to use their values to calculate `Route`. Here is a stripped example:

```
1  CompletableFuture<Customer> customerFuture = loadCustomer
2  CompletableFuture<Shop> shopFuture = closestShop();
3  CompletableFuture<Route> routeFuture =
4      customerFuture.thenCombine(shopFuture, (cust, shop) -
5
6  //...
7
8  private Route findRoute(Customer customer, Shop shop) //.
```

Notice that in Java 8 you can replace `(cust, shop) -> findRoute(cust, shop)` with simple `this::findRoute` method reference:

```
1  customerFuture.thenCombine(shopFuture, this::findRoute);
```

So you get the idea. We have `customerFuture` and `shopFuture`. Then `routeFuture` wraps them and "waits" for both to complete. When both of them are

ready, it runs our supplied function that combines results (`findRoute()`). Thus `routeFuture` will complete when two underlying futures are resolved *and* `findRoute()` is done.

Waiting for *both* `CompletableFutures` to complete

If instead of producing new `CompletableFuture` combining both results we simply want to be notified when they finish, we can use `thenAcceptBoth()`/`runAfterBoth()` family of methods (`...Async` variations are available as well). They work similarly to `thenAccept()` and `thenRun()` but wait for two futures instead of one:

```
ure<Void> thenAcceptBoth(CompletableFuture<? extends U> other,
Void> runAfterBoth(CompletableFuture<?> other, Runnable action)
```

Imagine that in the example above, instead of producing new `CompletableFuture<Route>` you simply want send some event or refresh GUI immediately. This can be easily achieved with `thenAcceptBoth()`:

```
1 customerFuture.thenAcceptBoth(shopFuture, (cust, shop) ->
2     final Route route = findRoute(cust, shop);
3     //refresh GUI with route
4 });
```

I hope I'm wrong but maybe some of you are asking themselves a question: *why can't I simply block on these two futures?* Like here:

```
1 Future<Customer> customerFuture = loadCustomerDetails(123
2 Future<Shop> shopFuture = closestShop();
3 findRoute(customerFuture.get(), shopFuture.get());
```

Well, of course you can. But the whole point of `CompletableFuture` is to allow asynchronous, event driven programming model instead of blocking and eagerly waiting for result. So functionally two code snippets above are equivalent, but the latter unnecessarily occupies one thread of execution.

Waiting for first `CompletableFuture` to complete

Another interesting part of the `CompletableFuture` API is the ability to wait for *first* (as opposed to *all*) completed future. This can come handy when you have two tasks yielding result of the same type and you only care about response time, not which task resulted first. API methods (`...Async` variations are available as well):

```
1 CompletableFuture<Void> acceptEither(CompletableFuture<?
2 CompletableFuture<Void> runAfterEither(CompletableFuture<
```

As an example say you have two systems you integrate with. One has smaller average response times but high standard deviation. Other one is slower in general, but more predictable. In order to take best of both worlds (performance and predictability) you call both systems at the same time and wait for the first one to complete. Normally it will be the first one, but in case it became slow, second one finishes in an acceptable time:

```
CompletableFuture<String> fast = fetchFast();
CompletableFuture<String> predictable = fetchPredictably(
fast.acceptEither(predictable, s -> {
    System.out.println("Result: " + s);
});
```

`s` represents `String` reply either from `fetchFast()` or from `fetchPredictably()`. We neither know nor care.

Transforming first completed

`applyToEither()` is an older brother of `acceptEither()`. While the latter simply calls some piece of code when faster of two futures complete, `applyToEither()` will return a new future. This future will complete when *first* of the two underlying futures complete. API is a bit similar (`...Async` variations are available as well):

```
<U> CompletableFuture<U> applyToEither(CompletableFuture<
```

The extra `fn` function is invoked on the result of first future that completed. I am not really sure what's the purpose of such a specialized method, after all one could simply use: `fast.applyToEither(predictable).thenApply(fn)`. Since we are stuck with this API but we don't really need extra function application, I will simply use `Function.identity()` [http://download.java.net/lambda/b88/docs/api/java/util/function/Function.html#identity()] placeholder:

```
CompletableFuture<String> fast = fetchFast();
CompletableFuture<String> predictable = fetchPredictably(
CompletableFuture<String> firstDone =
    fast.applyToEither(predictable, Function.<String>ider
```

`firstDone` future can then be passed around. Notice that from the client perspective the fact that two futures are actually behind `firstDone` is hidden. Client simply waits for future to complete and `applyToEither()` takes care of notifying the client when any of the two finish first.

Combining multiple `CompletableFuture` together

So we now know how to wait for two futures to complete (using `thenCombine()`) and for the first one to complete (`applyToEither()`). But can it scale to arbitrary

number of futures? Sure, using `static` helper methods:

```
tic CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
tic CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs
```

`allOf()` takes an array of futures and returns a future that completes when all of the underlying futures are completed (barrier waiting for all). `anyOf()` on the other hand will wait only for the fastest of the underlying futures. Please look at the generic type of returned futures. Not quite what you would expect? We will take care of this issue in the next article.

Summary

We explored pretty much whole `CompletableFuture` API [http://download.java.net/lambda/b88/docs/api/java/util/concurrent/CompletableFuture.html] . I'm sure this was quite overwhelming so in the next article shortly we will develop yet another implementation of simple web crawling program, taking advantage of `CompletableFuture` functionalities and Java 8 lambdas. We will also look at disadvantages and shortcomings of `CompletableFuture`.

Posted 9th May 2013 by Tomasz Nurkiewicz

Labels: CompletableFuture, functional programming, java 8, multithreading

View comments