DZone

20 Examples of Using Java's CompletableFuture

by Mahmoud Anouti MVB · Feb. 27, 18 · Java Zone

This post revisits Java 8's `CompletionStage` API and specifically its implementation in the standard Java library, `CompletableFuture`. The API is explained by examples that illustrate the various behaviors, where each example focuses on a specific one or two behaviors.

Since the `CompletableFuture` class implements the `CompletionStage` interface, we first need to understand the contract of that interface. It represents *a stage of a certain computation which can be done either synchronously or asynchronously*. You can think of it as just a single unit of a pipeline of computations that ultimately generate a final result of interest. This means that several `CompletionStage` commands can be chained together so that one stage's completion triggers the execution of another stage, which in turns triggers another, and so on.

In addition to implementing the `CompletionStage` interface, `CompletableFuture` also implements `Future`, which represents a pending asynchronous event, with the ability to *explicitly complete* this `Future`, hence the name CompletableFuture.

1. Creating a Completed CompletableFuture

The simplest example creates an already completed `CompletableFuture` with a predefined result. Usually this may act as the starting stage in your computation.

```
static void completedFutureExample() {
    CompletableFuture<String> cf = CompletableFuture.completedFuture("message");
    assertTrue(cf.isDone());
    assertEquals("message", cf.getNow(null));
}
```

The `getNow(null)` returns the result if completed (which is obviously the case), otherwise returns `null` (the argument).

2. Running a Simple Asynchronous Stage

The next example is how to create a stage that executes a `Runnable` asynchronously:

```
static void runAsyncExample() {
    CompletableFuture<Void> cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

The takeaway of this example is two things:

1. A CompletableFuture is executed asynchronously when the method typically ends with the keyword *`Async`*.
2. By default (when no `Executor` is specified), asynchronous execution uses the common `ForkJoinPool` implementation, which uses daemon threads to execute the `Runnable` task. Note that this is specific to `CompletableFuture`. Other `CompletionStage` implementations can override the default behavior.

3. Applying a Function on Previous Stage

The below example takes the completed `CompletableFuture` from example #1, which bears the result string `"message"`, and applies a function that converts it to uppercase:

```
static void thenApplyExample() {
    CompletableFuture<String> cf = CompletableFuture.completedFuture("message").thenA
        assertFalse(Thread.currentThread().isDaemon());
        return s.toUpperCase();
    });
    assertEquals("MESSAGE", cf.getNow(null));
}
```

Note the behavioral keywords in `thenApply`:

1. *then*, which means that the action of this stage happens when the current stage completes normally (without an exception). In this case, the current stage is already completed with the value "message".
2. *Apply*, which means the returned stage will apply a `Function` on the result of the previous stage.

The execution of the `Function` will be blocking, which means that getNow() will only be reached when the uppercase operation is done.

4. Asynchronously Applying a `Function` on Previous Stage

By appending the `Async` suffix to the method in the previous example, the chained `CompletableFuture` would execute asynchronously (using `ForkJoinPool.commonPool()`).

```
static void thenApplyAsyncExample() {
    CompletableFuture<String> cf = CompletableFuture.completedFuture("message").thenA
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
        return s.toUpperCase();
    });
    assertNull(cf.getNow(null));
    assertEquals("MESSAGE", cf.join());
}
```

5. Asynchronously Applying a Function on Previous Stage Using a Custom Executor

A very useful feature of asynchronous methods is the ability to provide an `Executor` to use it to execute the desired `CompletableFuture`. This example shows how to use a fixed thread pool to apply the uppercase conversion function:

```
static ExecutorService executor = Executors.newFixedThreadPool(3, new ThreadFactory()
    int count = 1;

    @Override
    public Thread newThread(Runnable runnable) {
        return new Thread(runnable, "custom-executor-" + count++);
    }
});

static void thenApplyAsyncWithExecutorExample() {
    CompletableFuture<String> cf = CompletableFuture.completedFuture("message").thenA
        assertTrue(Thread.currentThread().getName().startsWith("custom-executor-"));
        assertFalse(Thread.currentThread().isDaemon());
        randomSleep();
        return s.toUpperCase();
    }, executor);

    assertNull(cf.getNow(null));
    assertEquals("MESSAGE", cf.join());
}
```

6. Consuming Result of Previous Stage

If the next stage accepts the result of the current stage but does not need to return a value in the computation (i.e. its return type is void), then instead of applying a `Function`, it can accept a `Consumer`, hence the method `thenAccept`:

```
static void thenAcceptExample() {
    StringBuilder result = new StringBuilder();
    CompletableFuture.completedFuture("thenAccept message")
            .thenAccept(s -> result.append(s));
    assertTrue("Result was empty", result.length() > 0);
}
```

The `Consumer` will be executed synchronously, so we don't need to join on the returned `CompletableFuture`.

7. Asynchronously Consuming Result of Previous Stage

Again, using the async version of `thenAccept`, the chained `CompletableFuture` would execute asynchronously:

```
static void thenAcceptAsyncExample() {
    StringBuilder result = new StringBuilder();
    CompletableFuture<Void> cf = CompletableFuture.completedFuture("thenAcceptAsync m
            .thenAcceptAsync(s -> result.append(s));
    cf.join();
    assertTrue("Result was empty", result.length() > 0);
}
```

8. Completing a Computation Exceptionally

Now let us see how an asynchronous operation can be explicitly completed *exceptionally*, indicating a failure in the computation. For simplicity, the operation takes a string and converts it to upper case, and we simulate a delay in the operation of 1 second. To do that, we will use the `thenApplyAsync(Function, Executor)` method, where the first argument is the uppercase function, and the executor is a *delayed executor* that waits for 1 second before actually submitting the operation to the common `ForkJoinPool`.

```
static void completeExceptionallyExample() {
    CompletableFuture<String> cf = CompletableFuture.completedFuture("message").thenA
            CompletableFuture.delayedExecutor(1, TimeUnit.SECONDS));
    CompletableFuture<String> exceptionHandler = cf.handle((s, th) -> { return (th !=
    cf.completeExceptionally(new RuntimeException("completed exceptionally"));
assertTrue("Was not completed exceptionally", cf.isCompletedExceptionally());
    try {
        cf.join();
        fail("Should have thrown an exception");
    } catch(CompletionException ex) { // just for testing
        assertEquals("completed exceptionally", ex.getCause().getMessage());
    }

    assertEquals("message upon cancel", exceptionHandler.join());
}
```

Let's examine this example in detail:

- First, we create a `CompletableFuture` that is already completed with the value `"message"`. Next we call `thenApplyAsync` which returns a new `CompletableFuture`. This method applies an uppercase conversion in an asynchronous fashion upon completion of the first stage (which is already complete, thus the `Function` will be immediately executed). This example also illustrates a way to delay the asynchronous task using the `delayedExecutor(timeout, timeUnit)` method.
- We then create a separate "handler" stage, `exceptionHandler`, that handles any exception by returning another message `"message upon cancel"`.
- Next we explicitly complete the second stage with an exception. This makes the `join()` method on the stage, which is doing the uppercase operation, throw a `CompletionException` (normally `join()` would have waited for 1 second to get the uppercase string). It will also trigger the handler stage.

9. Canceling a Computation

Very close to exceptional completion, we can cancel a computation via the `cancel(boolean mayInterruptIfRunning)` method from the `Future` interface. For `CompletableFuture`, the boolean parameter is not used because the implementation does not employ interrupts to do the cancelation. Instead, `cancel()` is equivalent to `completeExceptionally(new CancellationException())`.

```
static void cancelExample() {
    CompletableFuture<String> cf = CompletableFuture.completedFuture("message").thenA
            CompletableFuture.delayedExecutor(1, TimeUnit.SECONDS));
    CompletableFuture<String> cf2 = cf.exceptionally(throwable -> "canceled message")
    assertTrue("Was not canceled", cf.cancel(true));
    assertTrue("Was not completed exceptionally", cf.isCompletedExceptionally());
    assertEquals("canceled message", cf2.join());
}
```

10. Applying a Function to Result of Either of Two Completed Stages

The below example creates a `CompletableFuture` that applies a `Function` to the result of either of two previous stages (no guarantees on which one will be passed to the `Function`). The two stages in question are: one that applies an uppercase conversion to the original string, and another that applies a lowercase conversion:

```
static void applyToEitherExample() {
    String original = "Message";
    CompletableFuture<String> cf1 = CompletableFuture.completedFuture(original)
            .thenApplyAsync(s -> delayedUpperCase(s));
    CompletableFuture<String> cf2 = cf1.applyToEither(
            CompletableFuture.completedFuture(original).thenApplyAsync(s -> delayedLo
            s -> s + " from applyToEither");
    assertTrue(cf2.join().endsWith(" from applyToEither"));
}
```

11. Consuming Result of Either of Two Completed Stages

Similar to the previous example, but using a `Consumer` instead of a `Function` (the dependent `CompletableFuture` has a type void):

```
static void acceptEitherExample() {
    String original = "Message";
    StringBuilder result = new StringBuilder();
    CompletableFuture<Void> cf = CompletableFuture.completedFuture(original)
            .thenApplyAsync(s -> delayedUpperCase(s))
            .acceptEither(CompletableFuture.completedFuture(original).thenApplyAsync(
                    s -> result.append(s).append("acceptEither"));
    cf.join();
    assertTrue("Result was empty", result.toString().endsWith("acceptEither"));
}
```

12. Running a Runnable upon Completion of Both Stages

This example shows how the dependent `CompletableFuture` that executes a `Runnable` triggers upon completion of both of two stages. Note all below stages run synchronously, where a stage first converts a message string to uppercase, then a second converts the same message string to lowercase.

```
static void runAfterBothExample() {
    String original = "Message";
    StringBuilder result = new StringBuilder();
    CompletableFuture.completedFuture(original).thenApply(String::toUpperCase).runAft
            CompletableFuture.completedFuture(original).thenApply(String::toLowerCase
            () -> result.append("done"));
    assertTrue("Result was empty", result.length() > 0);
}
```

13. Accepting Results of Both Stages in a Biconsumer

Instead of executing a `Runnable` upon completion of both stages, using `BiConsumer` allows processing of their results if needed:

```
static void thenAcceptBothExample() {
    String original = "Message";
    StringBuilder result = new StringBuilder();
    CompletableFuture.completedFuture(original).thenApply(String::toUpperCase).thenAc
            CompletableFuture.completedFuture(original).thenApply(String::toLowerCase
            (s1, s2) -> result.append(s1 + s2));
    assertEquals("MESSAGEmessage", result.toString());
}
```

14. Applying a Bifunction on Results of Both Stages (i.e. Combining Their Results)

If the dependent `CompletableFuture` is intended to combine the results of two previous `CompletableFuture` s by applying a function on them and returning a result, we can use the method `thenCombine()` . The entire pipeline is synchronous, so `getNow()` at the end would retrieve the final result, which is the concatenation of the uppercase and the lowercase outcomes.

```
static void thenCombineExample() {
    String original = "Message";
    CompletableFuture<String> cf = CompletableFuture.completedFuture(original).thenAp
            .thenCombine(CompletableFuture.completedFuture(original).thenApply(s -> d
                    (s1, s2) -> s1 + s2);
    assertEquals("MESSAGEmessage", cf.getNow(null));
}
```

15. Asynchronously Applying a BiFunction on (i.e. Combining) Results of Both Stages

Similar to the previous example, but with a different behavior: since the two stages upon which `CompletableFuture` depends both run asynchronously, *the `thenCombine()` method executes asynchronously, even though it lacks the `Async` suffix*. This is documented in the class Javadocs: *"Actions supplied for dependent completions of non-async methods may be performed by the thread that completes the current CompletableFuture, or by any other caller of a completion method."* Therefore, we need to `join()` on the combining `CompletableFuture` to wait for the result.

```
static void thenCombineAsyncExample() {
    String original = "Message";
    CompletableFuture<String> cf = CompletableFuture.completedFuture(original)
            .thenApplyAsync(s -> delayedUpperCase(s))
            .thenCombine(CompletableFuture.completedFuture(original).thenApplyAsync(s
                    (s1, s2) -> s1 + s2);
    assertEquals("MESSAGEmessage", cf.join());
}
```

16. Composing CompletableFutures

We can use composition using `thenCompose()` to accomplish the same computation done in the previous two examples. This method waits for the first stage (which applies an uppercase conversion) to complete. Its result is passed to the specified `Function` which returns a `CompletableFuture` , whose result will be the result of the returned `CompletableFuture` . In this case, the Function takes the uppercase string (`upper`), and returns a `CompletableFuture` that converts the `original` string to lowercase and then appends it to `upper`.

```
static void thenComposeExample() {
    String original = "Message";
    CompletableFuture<String> cf = CompletableFuture.completedFuture(original).thenAp
            .thenCompose(upper -> CompletableFuture.completedFuture(original).thenApp
                    .thenApply(s -> upper + s));
    assertEquals("MESSAGEmessage", cf.join());
}
```

17. Creating a Stage That Completes When Any of Several Stages Completes

The below example illustrates how to create a `CompletableFuture` that completes when any of several `CompletableFuture`s completes, with the same result. Several stages are first created, each converting a string from a list to uppercase. Because all of these `CompletableFuture`s are executing synchronously (using `thenApply()`), the `CompletableFuture` returned from `anyOf()` would execute immediately, since by the time it is invoked, all stages are completed. We then use the `whenComplete(BiConsumer<? super Object, ? super Throwable> action)`, which processes the result (asserting that the result is uppercase).

```
static void anyOfExample() {
    StringBuilder result = new StringBuilder();
    List messages = Arrays.asList("a", "b", "c");
    List<CompletableFuture<String>> futures = messages.stream()
            .map(msg -> CompletableFuture.completedFuture(msg).thenApply(s -> delayed
            .collect(Collectors.toList());
    CompletableFuture.anyOf(futures.toArray(new CompletableFuture[futures.size()])).w
        if(th == null) {
            assertTrue(isUpperCase((String) res));
            result.append(res);
        }
    });
    assertTrue("Result was empty", result.length() > 0);
}
```

18. Creating a Stage That Completes When All Stages Complete

The next two examples illustrate how to create a `CompletableFuture` that completes when all of several `CompletableFuture`s completes, in a synchronous and then asynchronous fashion, respectively. The scenario is the same as the previous example: a list of strings is provided where each element is converted to uppercase.

```
static void allOfExample() {
    StringBuilder result = new StringBuilder();
    List messages = Arrays.asList("a", "b", "c");
    List<CompletableFuture<String>> futures = messages.stream()
```

```
            .map(msg -> CompletableFuture.completedFuture(msg).thenApply(s -> delayed
            .collect(Collectors.toList());
    CompletableFuture.allOf(futures.toArray(new CompletableFuture[futures.size()])).w
        futures.forEach(cf -> assertTrue(isUpperCase(cf.getNow(null))));
        result.append("done");
    });
    assertTrue("Result was empty", result.length() > 0);
}
```

19. Creating a Stage That Completes Asynchronously When All Stages Complete

By switching to `thenApplyAsync()` in the individual `CompletableFuture` s, the stage returned by `allOf()` gets executed by one of the common pool threads that completed the stages. So we need to call `join()` on it to wait for its completion.

```
static void allOfAsyncExample() {
    StringBuilder result = new StringBuilder();
    List messages = Arrays.asList("a", "b", "c");
    List<CompletableFuture<String>> futures = messages.stream()
            .map(msg -> CompletableFuture.completedFuture(msg).thenApplyAsync(s -> de
            .collect(Collectors.toList());
    CompletableFuture<Void> allOf = CompletableFuture.allOf(futures.toArray(new Compl
            .whenComplete((v, th) -> {
                futures.forEach(cf -> assertTrue(isUpperCase(cf.getNow(null))));
                result.append("done");
            });
    allOf.join();
    assertTrue("Result was empty", result.length() > 0);
}
```

20. Real Life Example

Now that the functionality of `CompletionStage` and specifically `CompletableFuture` is explored, the below example applies them in a practical scenario:

1. First fetch a list of `Car` objects asynchronously by calling the `cars()` method, which returns a `CompletionStage<List>` . The `cars()` method could be consuming a remote REST endpoint behind the scenes.
2. We then compose another `CompletionStage<List>` that takes care of filling the rating of each car, by calling the `rating(manufacturerId)` method which returns a `CompletionStage` that asynchronously fetches the car rating (again could be consuming a REST endpoint).
3. When all `Car` objects are filled with their rating, we end up with a `List<CompletionStage>` , so we call `allOf()` to get a final stage (stored in variable `done`) that completes upon completion of all

these stages.

4. Using `whenComplete()` on the final stage, we print the `Car` objects with their rating.

```
cars().thenCompose(cars -> {
    List<CompletionStage<Car>> updatedCars = cars.stream()
            .map(car -> rating(car.manufacturerId).thenApply(r -> {
                car.setRating(r);
                return car;
            })).collect(Collectors.toList());

    CompletableFuture<Void> done = CompletableFuture
            .allOf(updatedCars.toArray(new CompletableFuture[updatedCars.size()]));
    return done.thenApply(v -> updatedCars.stream().map(CompletionStage::toCompletabl
            .map(CompletableFuture::join).collect(Collectors.toList()));
}).whenComplete((cars, th) -> {
    if (th == null) {
        cars.forEach(System.out::println);
    } else {
        throw new RuntimeException(th);
    }
}).toCompletableFuture().join();
```

Since the `Car` instances are all independent, getting each rating asynchronously improves performance. Furthermore, waiting for all car ratings to be filled is done using a more natural `allOf()` method, as opposed to manual thread waiting (e.g. using `Thread#join()` or a `CountDownLatch`).

Working through these examples helps better understand this API. You can find the full code of these examples on GitHub.