Advanced Java CompletableFuture Features: Single Stage Completion Methods (Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

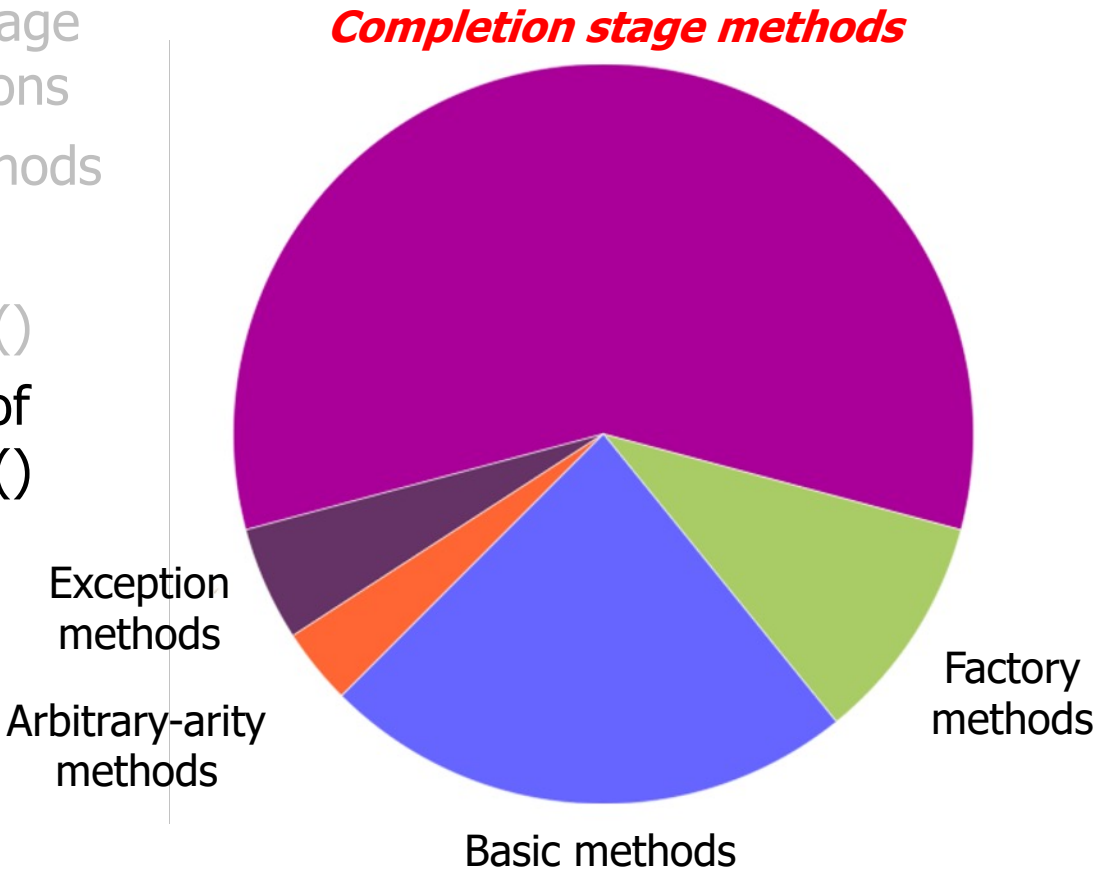
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how completion stage methods chain dependent actions
- Know how to group these methods
- Single stage methods, e.g.
  - thenApply() & thenCompose()
  - thenAccept() & comparison of thenApply() & thenCompose()



Methods Triggered by Completion of a Single Stage

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenAccept()

```
CompletableFuture<Void>
  thenAccept
    (Consumer<? super T> action)
{ ... }
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenAccept

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenAccept()
    - Applies a consumer action to handle previous stage's result

```
CompletableFuture<Void>
  thenAccept
    (Consumer<? super T> action)
{ ... }
```

See docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenAccept()
    - Applies a consumer action to handle previous stage's result

```
CompletableFuture<Void>
   thenAccept
      (Consumer<? super T> action)
{  ...  }
```

This action behaves as a "callback" with a side-effect

See en.wikipedia.org/wiki/Callback_(computer_programming)

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void

```
CompletableFuture<Void>
  thenAccept
    (Consumer<? super T> action)
{ ... }
```

See www.baeldung.com/java-void-type

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger("..."),
            new BigInteger("..."),
            false); // Don't reduce!

Supplier<BigFraction> reduce = ()
   -> BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

thenApply() returns a string future that thenAccept() prints when it completes

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger("..."),
            new BigInteger("..."),
            false); // Don't reduce!

Supplier<BigFraction> reduce = ()
   -> BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

println() is a callback that has a side-effect (i.e., printing the mixed string)

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void
    - Often used at the end of a chain of completion stages
    - May lead to "callback hell" if used excessively!

See dzone.com/articles/callback-hell

Comparing thenApply() & thenCompose()

Comparing thenApply() & thenCompose()

- thenApply() & thenCompose() have similar method signatures

```
CompletableFuture<U> thenApply
  (Function<? super T,
            ? extends U> fn)
{ ... }

CompletableFuture<U> thenCompose
  (Function<? super T,
            ? extends
            CompletionStage<U>> fn)
{ ... }
```

Comparing thenApply() & thenCompose()

- Unlike thenApply(), however, thenCompose() avoids unwieldy nesting of futures

```
Function<BF, CompletableFuture<
              CompletableFuture<BF>>>
  reduceAndMultiplyFractions =
    unreduced -> CompletableFuture
    .supplyAsync
      (() -> BF.reduce(unreduced))


    .thenApply
      (reduced -> CompletableFuture
       .supplyAsync(() ->
        reduced.multiply(...)));
  ...
```

Nesting is unwieldy!

Comparing thenApply() & thenCompose()

- Unlike thenApply(), however, thenCompose() avoids unwieldy nesting of futures

Eliminates the nesting of futures via "flattening"!

```
Function<BF,
        CompletableFuture<BF>>
  reduceAndMultiplyFractions =
    unreduced -> CompletableFuture
    .supplyAsync
      (() -> BF.reduce(unreduced))


    .thenCompose
      (reduced -> CompletableFuture
       .supplyAsync(() ->
        reduced.multiply(...)));
  ...
```

Comparing thenApply() & thenCompose()

- Unlike thenApply(), however, thenCompose() avoids unwieldy nesting of futures
  - thenApplyAsync() can often be used to replace nesting of thenCompose(supplyAsync())

```
Function<BF,
         CompletableFuture<BF>>
  reduceAndMultiplyFractions =
    unreduced -> CompletableFuture
    .supplyAsync
      (() -> BF.reduce(unreduced))



    .thenApplyAsync(reduced
       -> reduced.multiply(...)));
...
...
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenApplyAsync

Comparing thenApply() & thenCompose()

- Unlike thenApply(), however, thenCompose() avoids unwieldy nesting of futures
  - thenApplyAsync() can often be used to replace nesting of thenCompose(supplyAsync())
  - thenCompse() can also avoid calling join() when flattening nested completable futures

```
CompletableFuture<Integer> countF =
   .CompletableFuture
      .supplyAsync
         (() ->
          longRunnerReturnsCF())

      .thenCompose
        (Function.identity())
...
```

supplyAsync() will return a CompletableFuture to a CompletableFuture here!!

Comparing thenApply() & thenCompose()

- Unlike thenApply(), however, thenCompose() avoids unwieldy nesting of futures
  - thenApplyAsync() can often be used to replace nesting of thenCompose(supplyAsync())
  - thenCompse() can also avoid calling join() when flattening nested completable futures

```
CompletableFuture<Integer> countF =
   .CompletableFuture
      .supplyAsync
         (() ->
          longRunnerReturnsCF())

      .thenCompose
        (Function.identity())
...
```

This idiom flattens the return value to "just" one CompletableFuture!

Comparing thenApply() & thenCompose()

- Unlike thenApply(), however, thenCompose() avoids unwieldy nesting of futures
  - thenApplyAsync() can often be used to replace nesting of thenCompose(supplyAsync())
  - thenCompse() can also avoid calling join() when flattening nested completable futures
  - thenComposeAsync() can avoid calling supplyAsync() again in a chain

```
CompletableFuture<Integer> countF =
   .CompletableFuture
      .supplyAsync
         (() ->
          longRunnerReturnsCF())

      .thenComposeAsync
        (this::longerBlockerReturnsCF)
...
```

Runs longBlockerReturnsCF() in a common fork-join pool thread

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenComposeAsync

End of Advanced Java CompletableFuture Features: Single Stage Completion Methods (Part 2)