Advanced Java CompletableFuture Features: Introducing Completion Stage Methods

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 advanced features of completable futures, e.g.
  - Factory methods initiate async computations
  - Completion stage methods chain together actions to perform async result processing & composition
Completion Stage Methods
Chain Actions Together
A completable future can serve as a "completion stage" for async result processing.

Completion Stage Methods Chain Actions Together

interface CompletionStage<T>

All Known Implementing Classes:
CompletableFuture

public interface CompletionStage<T>

A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes. A stage completes upon termination of its computation, but this may in turn trigger other dependent stages. The functionality defined in this interface takes only a few basic forms, which expand out to a larger set of methods to capture a range of usage styles:

- The computation performed by a stage may be expressed as a Function, Consumer, or Runnable (using methods with names including apply, accept, or run, respectively) depending on whether it requires arguments and/or produces results. For example, stage.thenApply(x -> square(x)).thenAccept(x -> System.out.println(x)).thenRun(() -> System.out.println()). An additional form (compose) applies functions of stages themselves, rather than their results.
- One stage's execution may be triggered by completion of a single stage, or both of two stages, or either of two stages. Dependencies on a single stage are arranged using methods with prefix then. Those triggered by completion of both of two stages may combine their results or effects, using correspondingly named methods. Those triggered by either of two stages make no guarantees about which of the results or effects are used for the dependent stage's computation.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html
BigFraction unreduced = BigFraction
  .valueOf(new BigInteger
      ("846122553600669882"),
      new BigInteger
      ("188027234133482196")(false)); // Don’t reduce!

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

CompletableFuture
  .supplyAsync(reduce)
  .thenApply(BigFraction
              ::toMixedString)
  ...

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8
A completable future can serve as a "completion stage" for async result processing.

A dependent action runs on a completed async call result.

Create an unreduced big fraction variable: 

```java
BigFraction unreduced = BigFraction
    .valueOf(new BigInteger
        ("846122553600669882"),
        new BigInteger
        ("188027234133482196"),
        false); // Don’t reduce!

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

CompletableFuture
    .supplyAsync(reduce)
    .thenApply(BigFraction
        ::toMixedString)
    ...
```
A completable future can serve as a "completion stage" for async result processing. A dependent action runs on a completed async call result.

Create a supplier lambda variable that will reduce the big fraction:

```java
BigFraction unreduced = BigFraction.
    .valueOf(new BigInteger
        ("846122553600669882"),
        new BigInteger
        ("188027234133482196"),
        false); // Don't reduce!

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

CompletableFuture
    .supplyAsync(reduce)
    .thenApply(BigFraction
        ::toMixedString)
    ...
```
A completable future can serve as a "completion stage" for async result processing.

A dependent action runs on a completed async call result.

```java
BigFraction unreduced = BigFraction
    .valueOf(new BigInteger("846122553600669882"),
             new BigInteger("188027234133482196"),
             false); // Don't reduce!

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

CompletableFuture
    .supplyAsync(reduce)
    .thenApply(BigFraction::toMixedString)
    ...
```

This factory method will asynchronously reduce the big fraction supplier lambda.
Completion Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for async result processing
- A dependent action runs on a completed async call result

```
BigFraction unreduced = BigFraction
    .valueOf(new BigInteger
        ("846122553600669882"),
        new BigInteger
        ("188027234133482196"),
        false); // Don’t reduce!
Supplier<BigFraction> reduce = () ->
    BigFraction.reduce(unreduced);
CompletableFuture
    .supplyAsync(reduce)
    .thenApply(BigFraction
        ::toMixedString)
    ...
```

*thenApply()’s action is triggered when future from supplyAsync() completes*
A completable future can serve as a ”completion stage” for async result processing

A dependent action runs on a completed async call result

Methods can be chained together “fluently”

```java
BigFraction unreduced = BigFraction
    .valueOf(new BigInteger
        ("846122553600669882"),
        new BigInteger
        ("188027234133482196"),
        false); // Don’t reduce!

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

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

thenAccept()’s action is triggered when future from thenApply() completes

See [en.wikipedia.org/wiki/Fluent_interface](en.wikipedia.org/wiki/Fluent_interface)
A completable future can serve as a "completion stage" for async result processing.

A dependent action runs on a completed async call result.

Methods can be chained together "fluently".

Each method registers a lambda action to apply.
A completable future can serve as a “completion stage” for async result processing.

A dependent action runs on a completed async call result.

Methods can be chained together “fluently”

Each method registers a lambda action to apply.

A lambda action is called only after previous stage completes successfully.

This is what is meant by “chaining”:
A completable future can serve as a "completion stage" for async result processing

- A dependent action runs on a completed async call result
- Methods can be chained together "fluently"
- Each method registers a lambda action to apply
- A lambda action is called only after previous stage completes successfully

```java
BigFraction unreduced = BigFraction
    .valueOf(new BigInteger("846122553600669882"),
             new BigInteger("188027234133482196"),
             false); // Don't reduce!

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

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

Action is "deferred" until previous stage completes & fork-join thread is available
Use completion stages to avoid blocking a thread until the result *must* be obtained.
Completion Stage Methods Chain Actions Together

- Use completion stages to avoid blocking a thread until the result *must* be obtained, e.g.
- Try not to call `join()` or `get()` unless absolutely necessary

Servers may avoid blocking completely, whereas clients may need `join()` sparingly.
Completion Stage Methods Chain Actions Together

- Use completion stages to avoid blocking a thread until the result *must* be obtained, e.g.
  - Try not to call join() or get() unless absolutely necessary
  - The goal is to improve responsiveness
A completable future can serve as a "completion stage" for async result processing.

Juggling is a good analogy for completion stages!
End of Advanced Java
CompletableFuture Features:
Introducing Completion Stage Methods