Advanced Java CompletableFuture Features: Introducing Completion Stage Methods (Part 1)

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 dependent actions
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 dependent actions
  - Perform async result processing & composition

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html
Overview of Completion Stages
Overview of Completion Stages

- A completable future can serve as a "completion stage" for async result processing

### Interface CompletionStage<T>

All Known Implementing Classes:
ComplettableFuture

```java
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(x)).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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html)
Overview of Completion Stages

- A completable future can serve as a "completion stage" for async result processing.
- Performs an action or computes a value when another CompletionStage completes.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html)
Overview of Completion Stages

- A completable future can serve as a "completion stage" for async result processing
  - Performs an action or computes a value when another CompletionStage completes
- Juggling is a good analogy for completion stages!

See en.wikipedia.org/wiki/Juggling
Overview of Completion Stages

• A completable future can serve as a “completion stage” for async result processing
  • Performs an action or computes a value when another CompletionStage completes
• Juggling is a good analogy for completion stages!
• Resources are only consumed when an action runs, which reduces system overhead

See en.wikipedia.org/wiki/Start-stop_system
Chaining Actions Together via Completion Stage Methods
Chaining Actions Together via Completion Stage Methods

- CompletableObject futures can be chained together via completion stage methods.

See [github.com/douglascraigschmidt/LiveLessons](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8)

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html)
Chaining Actions Together via Completion Stage Methods

- Complettable futures can be chained together via completion stage methods
- A dependent action handles the result after a previous async call completes

```
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/douglasraigschmidt/LiveLessons/tree/master/Java8/ex8](github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex8)
Chaining Actions Together via Completion Stage Methods

- Complettable futures can be chained together via completion stage methods.
- A dependent action handles the result after a previous async call completes.

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)
    ...
```

See [math.answers.com/questions/What_is_an_unreduced_fraction](http://math.answers.com/questions/What_is_an_unreduced_fraction)
Completable futures can be chained together via completion stage methods.

A dependent action handles the result after a previous async call completes.

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)
  ...

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

See docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html
Completable futures can be chained together via completion stage methods

A dependent action handles the result after a previous async call completes

```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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#supplyAsync
Completable futures can be chained together via completion stage methods.

A dependent action handles the result after a previous async call completes.

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

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

CompletableFuture<BigFraction> result = CompletableFuture.supplyAsync(reduce)
    .thenApply(BigFraction::toMixedString);
```

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

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#supplyAsync](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#supplyAsync)
Chaining Actions Together via Completion Stage Methods

- Completable futures can be chained together via completion stage methods
- A dependent action handles the result after a previous async call completes
- 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);
```

See [en.wikipedia.org/wiki/Fluent_interface](en.wikipedia.org/wiki/Fluent_interface)
Chaining Actions Together via Completion Stage Methods

- Completable futures can be chained together via completion stage methods
- A dependent action handles the result after a previous async call completes
- 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 docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenAccept
Chaining Actions Together via Completion Stage Methods

- Completetable futures can be chained together via completion stage methods
- A dependent action handles the result after a previous async call completes
- Methods can be chained together “fluently”
- Each method registers an action to apply

```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);
```
Chaining Actions Together via Completion Stage Methods

- Completable futures can be chained together via completion stage methods
- A dependent action handles the result after a previous async call completes
- Methods can be chained together “fluently”
- Each method registers an 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);
```

This is what is meant by “chaining” via the Fluent Interface pattern
• Completable futures can be chained together via completion stage methods
  - A dependent action handles the result after a previous async call completes
• Methods can be chained together “fluently”
  - Each method registers an action to apply
• A lambda action is called only after previous stage completes successfully

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 & a fork-join thread is available
• Completable futures can be chained together via completion stage methods

• A dependent action handles the result after a previous async call completes

• Methods can be chained together “fluently”

• Fluent chaining enables async programming to look like sync programming

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);
End of Advanced Java CompletableFuture Features: Introducing Completion Stage Methods (Part 1)