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.
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)
    ...
```

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

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)
    ...
```
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

```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)
...
```

*Create a supplier lambda variable that will reduce the big fraction*
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.<BigFraction>supplyAsync(reduce)
    .thenApply(BigFraction::toMixedString)
    ...
```

This factory method will asynchronously reduce the big fraction supplier lambda.
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).
  ...
```

*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

```
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);
```
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);
```

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.
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.”

Fluent chaining enables async programming to look like sync programming.

```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);
```
Completion Stage Methods Chain Actions Together

- Use completion stages to avoid blocking the caller thread until the result *must* be obtained
Completion Stage Methods Chain Actions Together

- Use completion stages to avoid blocking the caller thread, e.g.
- Avoid calling `join()` or `get()` unless absolutely necessary
Completion Stage Methods Chain Actions Together

• Use completion stages to avoid blocking the caller thread, e.g.
  • Avoid calling `join()` or `get()` unless absolutely necessary
    • Improves responsiveness by not blocking
• Use completion stages to avoid blocking the caller thread, e.g.
  • Avoid calling join() or get() unless absolutely necessary
    • Improves responsiveness by not blocking
      • Clients & servers that apply completion stage methods may avoid blocking completely
A completable future can serve as a “completion stage” for async result processing.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html
A completable future can serve as a “completion stage” for async result processing.

Completion Stage Methods Chain Actions Together

Juggling is a good analogy for completion stages!
A completable future can serve as a “completion stage” for async result processing. It only consumes resources when an action runs, which reduces system overhead.

See [en.wikipedia.org/wiki/Start-stop_system](en.wikipedia.org/wiki/Start-stop_system)
End of Advanced Java
CompletableFuture Features:
Introducing Completion Stage Methods