Understand 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
Completion Stage Methods Chain Actions Together

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

**Interface CompletionStage<T>**

**All Known Implementing Classes:**
CompletableFuture

```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.print(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](https://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.

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/douglasraigschmidt/LiveLessons/tree/master/Java8/ex8](https://github.com/douglasraigschmidt/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
• 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(()'
```

`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.

```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

- 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"
### Completion Stage Methods Chain Actions Together

<table>
<thead>
<tr>
<th>Method</th>
<th>Code</th>
</tr>
</thead>
</table>
| A completable future can serve as a “completion stage” for async result processing | ```java
BigFraction unreduced = BigFraction.valueOf(new BigInteger("846122553600669882"),
                                             new BigInteger("188027234133482196"),
                                             false); // Don’t reduce!
```
| A dependent action runs on a completed async call result               | ```java
Supplier<BigFraction> reduce = () ->
BigFraction.reduce(unreduced);
```
| Methods can be chained together “fluently”                            | ```java
CompletableFuture.supplyAsync(reduce)
.thenApply(BigFraction::toMixedString)
.thenAccept(System.out::println);
```
| Each method registers a lambda action to apply                         | Action is “deferred” until previous stage completes & fork-join thread is available |
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
- 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
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 the *Asynchronous Completion Token* (ACT) pattern 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
End of Understand Advanced Java CompletableFuture Features: Introducing Completion Stage Methods