Overview of Advanced Java 8
CompletableFuture Features (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 advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
- Completion stage methods used to chain together actions that 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

**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
• An action is performed 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.

An action is performed 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.

An action is performed 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);
```
• A completable future can serve as a "completion stage" for async result processing
• An action is performed on a completed async call result
• Methods can be chained together "fluently"
• Each method registers a lambda action to apply

Invocation of a lambda action is "deferred" until previous future 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)
    .thenAccept(System.out::println);
```
A completable future can serve as a “completion stage” for async result processing.

An action is performed 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 the previous stage 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)
    .thenAccept(System.out::println);
```
A completable future can serve as a "completion stage" for async result processing

- An action is performed on a completed async call result
- Methods can be chained together "fluently"

Completion stages avoid blocking a thread until the result \textit{must} be obtained

```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);
```
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!
Grouping CompletableFuture
Completion Stage Methods
Completion stage methods are grouped based on how a stage is triggered by a previous stage.

Completion stage methods are grouped based on how a stage is triggered by a previous stage.

- Completion of a single previous stage

### Completion Stage Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Params</th>
<th>Returns</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>thenApply</td>
<td>Function</td>
<td>Completetable Future with Function result</td>
<td>Apply function to result of the previous stage</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thenCompose</td>
<td>Function</td>
<td>Completetable Future with Function result</td>
<td>Apply function to result of the previous stage</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td>directly, not a nested future</td>
<td></td>
</tr>
<tr>
<td>thenAccept</td>
<td>Consumer</td>
<td>Completetable Future&lt;Void&gt;</td>
<td>Consumer handles result of previous stage</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thenRun</td>
<td>Runnable</td>
<td>Completetable Future&lt;Void&gt;</td>
<td>Run action w/out returning value</td>
</tr>
</tbody>
</table>

### Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by a previous stage.
- Completion of a single previous stage.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Params</th>
<th>Returns</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>thenApply</td>
<td>Function</td>
<td>Completetable Future with Function result</td>
<td>Apply function to result of the previous stage</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thenCompose</td>
<td>Function</td>
<td>Completetable Future with Function result</td>
<td>Apply function to result of the previous stage. Not a nested future.</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thenAccept</td>
<td>Consumer</td>
<td>Completetable Future&lt;Void&gt;</td>
<td>Consumer handles result of previous stage.</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thenRun</td>
<td>Runnable</td>
<td>Completetable Future&lt;Void&gt;</td>
<td>Run action w/out returning value.</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Async() variants run in common fork-join thread (by default, run in same thread).
Completion stage methods are grouped based on how a stage is triggered by a previous stage:

- Completion of a single previous stage
- Completion of both of 2 previous stages
- i.e., an “and”

<table>
<thead>
<tr>
<th>Methods</th>
<th>Params</th>
<th>Returns</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>then Combine (Async)</td>
<td>Bi Function</td>
<td>Completable Future with Bi Function result</td>
<td>Apply bifunction to results of both previous stages</td>
</tr>
<tr>
<td>then Accept Both (Async)</td>
<td>Bi Consumer</td>
<td>Completable Future&lt;Void&gt;</td>
<td>BiConsumer handles results of both previous stages</td>
</tr>
<tr>
<td>runAfter Both (Async)</td>
<td>Runnable</td>
<td>Completable Future&lt;Void&gt;</td>
<td>Run action when both previous stages complete</td>
</tr>
</tbody>
</table>
Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by a previous stage
  - Completion of a single previous stage
  - Completion of both of 2 previous stages
  - Completion of either of 2 previous stages
    - i.e., an “or”

<table>
<thead>
<tr>
<th>Methods</th>
<th>Params</th>
<th>Returns</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>applyTo</td>
<td>Function</td>
<td>CompletableObject Future with Function result</td>
<td>Apply function to results of either previous stage</td>
</tr>
<tr>
<td>Either</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>accept</td>
<td>Consumer</td>
<td>CompletableObject Future&lt;Void&gt;</td>
<td>Consumer handles results of either previous stage</td>
</tr>
<tr>
<td>Either</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>runAfter</td>
<td>Runnable</td>
<td>CompletableObject Future&lt;Void&gt;</td>
<td>Run action when either previous stage completes</td>
</tr>
<tr>
<td>Either</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key CompletableFuture
Completion Stage Methods
Key CompletableFuture Completion Stage Methods

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

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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenApply
Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - `thenApply()`
    - Applies a function action to the previous stage’s result

```java
CompletableFuture<U> thenApply
    (Function<? super T,
    ? extends U> fn)

{ ... }
```
• Methods triggered by completion of a single previous stage
  • thenApply()
    • Applies a function action to the previous stage’s result
  • Returns a future containing the result of the action

CompletableFuture\<U\> thenApply
  (Function\<\? super T, 
   ? extends U\> fn)

{ ... }
### Key CompletableFuture Completion Stage Methods

- **Methods triggered by completion of a single previous stage**
  - `thenApply()`
    - Applies a function action to the previous stage’s result
    - Returns a future containing the result of the action
    - Used for a synchronous action that returns a value, not a future

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

Supplier<BigFraction> reduce = () -> BigFraction.reduce(unreduced);
CompletableFuture.supplyAsync(reduce)    
  .thenApply(BigFraction::toMixedString)
  ...
```

- e.g., `toMixedString()` returns a string value
Key CompletableFuture Completion Stage Methods

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

CompletableFuture<U> thenCompose
(Function<? super T, ? extends CompletionStage<U>> fn)

{ ... }

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenCompose
<table>
<thead>
<tr>
<th>Key CompletableFuture Completion Stage Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Methods triggered by completion of a single previous stage</td>
</tr>
<tr>
<td>• thenApply()</td>
</tr>
<tr>
<td>• thenCompose()</td>
</tr>
<tr>
<td>• Applies a function action to the previous stage’s result</td>
</tr>
</tbody>
</table>

```java
CompletableFuture<U> thenCompose(
    Function<? super T, ? extends CompletionStage<U>> fn)
{
    ... 
```
# Key CompletableFuture Completion Stage Methods

- **Methods triggered by completion of a single previous stage**
  - `thenApply()`
  - `thenCompose()`

- **`thenCompose()`**
  - Applies a function action to the previous stage’s result
  - Returns a future containing result of the action directly
  - *i.e., not a nested future*

```java
CompletableFuture<U> thenCompose
    (Function<? super T, ? extends CompletionStage<U>> fn)
{ ... }
```
Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - `thenApply()`
  - `thenCompose()`
    - Applies a function action to the previous stage’s result
    - Returns a future containing result of the action directly
      - *i.e., not* a nested future

CompletableFuture<U> `thenCompose`
(Fункцион< ? super T,
? extends
CompletionStage<U>> fn)

```
{ ... }
```

```
8 + 1 = 26
```

```
8 + 1 ≠ 10
```

`thenCompose()` is similar to `flatMap()` on a Stream or Optional
Key CompletableFuture Completion Stage Methods

- **Methods triggered by completion of a single previous stage**
  - `thenApply()`
  - `thenCompose()`

- **thenCompose()**
  - Applies a function action to the previous stage’s result
  - Returns a future containing result of the action directly
  - Used for an `async` action that returns a completable future

```java
Function<BF, CompletableFuture<BF>>
reduceAndMultiplyFractions = unreduced -> CompletableFuture
.supplyAsync(() -> BF.reduce(unreduced))
 .thenCompose(reduced -> CompletableFuture
                 .supplyAsync(() -> reduced.multiply(...)));
```

**e.g., supplyAsync() returns a completable future**
Function<BF, CompletableFuture<CompletableFuture<BF>>>
reduceAndMultiplyFractions = unreduced -> CompletableFuture.supplyAsync(() -> BF.reduce(unreduced)).thenApply(reduced -> CompletableFuture.supplyAsync(() -> reduced.multiply(...)))

**Key CompletableFuture Completion Stage Methods**

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage’s result
    - Returns a future containing result of the action directly
    - Used for an async action that returns a completable future
    - Avoids unwieldy nesting of futures à la thenApply()

- Unwieldy!

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage’s result
    - Returns a future containing result of the action directly
    - Used for an async action that returns a completable future
    - Avoids unwieldy nesting of futures à la thenApply()
**Key CompletableFuture Completion Stage Methods**

- Methods triggered by completion of a single previous stage
  - `thenApply()`
  - `thenCompose()`
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for an `async` action that returns a completable future
    - Avoids unwieldy nesting of futures à la `thenApply()`

```java
CompletableFuture<Integer> countF = CompletableFuture.supplyAsync(() -> returnCompletableFuture())
  .thenCompose(Function.identity())
  ...
```

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

Can be used to avoid calling `join()` when flattening nested completable futures...
This idiom flattens the return value to “just” a CompletableFuture!
Key CompletableFuture Completion Stage Methods

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

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

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenAccept](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenAccept)
Key CompletableFuture Completion Stage Methods

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

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

Note: This action behaves as a "callback" with a side-effect.

See [en.wikipedia.org/wiki/Callback_(computer_programming)](en.wikipedia.org/wiki/Callback_(computer_programming))
Key CompletableFuture Completion Stage Methods

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

```java
CompletableFuture<Void>
    .thenAccept
    (Consumer<? super T> action)
    { ... }
```
### Key CompletableFuture Completion Stage Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thenApply()</td>
<td>Applies a consumer action to handle previous stage's result</td>
</tr>
<tr>
<td>thenCompose()</td>
<td></td>
</tr>
<tr>
<td>thenAccept()</td>
<td>Applies a consumer action to handle previous stage's result</td>
</tr>
</tbody>
</table>

```java
def 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*
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);

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

println() is a callback that has a side-effect (i.e., printing the mixed string)
Methods triggered by completion of both of two previous stages

thenCombine()

```java
CompletableFuture<U> thenCombine
(CompletionStage<? Extends U> other,
{ ... }
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenCombine](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenCombine)
• Methods triggered by completion of both of two previous stages
• \textit{thenCombine()} (CompletionStage<? Extends U> other, BiFunction<? super T, ? super U, ? extends V> fn)
Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of both of two previous stages
  - `thenCombine()`
    - Applies a bifunction action to two previous stages’ results
    - Returns a future containing the result of the action

```java
CompletableFuture<U> thenCombine
    (CompletionStage<? Extends U> other,
     BiFunction<? super T,
               ? super U,
               ? extends V> fn)
```

{ ... }
**Key CompletableFuture Completion Stage Methods**

- Methods triggered by completion of both of two previous stages
  - `thenCombine()`
    - Applies a bifunction action to two previous stages’ results
    - Returns a future containing the result of the action

```java
CompletableFuture<? extends U> thenCombine
    (CompletionStage<? extends U> other,
     BiFunction<? super T,
                ? super U,
                ? extends V> fn)
```

`thenCombine()` essentially performs a “reduction”
Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages’ results
    - Returns a future containing the result of the action
    - Used to “join” two paths of execution

thenCombine()’s action is triggered when its two associated futures complete

```java
CompletableFuture<BF> compF1 = CompletableFuture.supplyAsync(() ->
    /* multiply two BFs. */);
CompletableFuture<BF> compF2 = CompletableFuture.supplyAsync(() ->
    /* divide two BFs. */);
compF1.thenCombine(compF2,
    BigFraction::add)
    .thenAccept(System.out::println);
```
Methods triggered by completion of either of two previous stages
• acceptEither()

```java
CompletableFuture<Void> acceptEither(
    CompletionStage<? Extends T> other,
    Consumer<? super T> action)
{
    ... 
}
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#acceptEither
Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of either of two previous stages
  - `acceptEither()`
    - Applies a consumer action that handles either of the previous stage’s results

```java
CompletableFuture<Void> acceptEither(CompletionStage<? Extends T> other,
                                      Consumer<? super T> action)
{ ... }
```
Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stage’s results
    - Returns a future to Void

CompletableFuture<Void> acceptEither(CompletionStage<? Extends T> other, Consumer<? super T> action)
{ ... }
### Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of either of two previous stages
- `acceptEither()`
  - Applies a consumer action that handles either of the previous stage’s results
  - Returns a future to Void
  - Often used at the end of a chain of completion stages

```java
CompletableFuture<List<BigFraction>> quickSort = CompletableFuture
    .supplyAsync(() ->
        quickSort(list));

CompletableFuture<List<BigFraction>> mergeSort = CompletableFuture
    .supplyAsync(() ->
        mergeSort(list));

quickSort.acceptEither
    (mergeSort, results -> results
     .forEach(fraction ->
         System.out.println
         (fraction
           .toMixedString())));
```

Print out sorted results from which ever sorting routine finished first.
End of Overview of Advanced Java 8 Completable Future Features (Part 2)