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

```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.

An action is performed on a completed async call result.
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](http://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
• 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);
```
A completable future can serve as a "completion stage" for async result processing.

<table>
<thead>
<tr>
<th>Completion Stage Methods Chain Actions Together</th>
</tr>
</thead>
</table>

- 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 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.
Completion Stage Methods Chain Actions Together

- 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
  - This approach helps improve responsiveness
A completable future can serve as a "completion stage" for async result processing.

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 one or more previous stage(s)

### Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by one or more previous stage(s).
- 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>CompletableFuture</td>
<td>Apply function to result of the previous stage</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td>Future with Function result</td>
<td></td>
</tr>
<tr>
<td>thenCompose</td>
<td>Function</td>
<td>CompletableFuture</td>
<td>Apply function to result of the previous stage</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td>Future with Function result</td>
<td></td>
</tr>
<tr>
<td>direct, <em>not</em></td>
<td></td>
<td>nested future</td>
<td></td>
</tr>
<tr>
<td>thenAccept</td>
<td>Consumer</td>
<td>CompletableFuture 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>CompletableFuture</td>
<td>Run action w/out returning value</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td>Future&lt;Void&gt;</td>
<td></td>
</tr>
</tbody>
</table>

*These methods run in the invoking thread or the same thread as previous stage*

The thread that executes these methods depends on various runtime factors.
Completion stage methods are grouped based on how a stage is triggered by one or more previous stage(s).

Completion of a single previous stage

*Async() variants run in common fork-join pool

See [blog.krecan.net/2013/12/25/completablefutures-why-to-use-async-methods](blog.krecan.net/2013/12/25/completablefutures-why-to-use-async-methods)
Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by one or more previous stage(s)
  - 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</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>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>then Accept Both</td>
<td>Bi Consumer</td>
<td>Completable Future&lt; Void&gt;</td>
<td>BiConsumer handles results of both previous stages</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>runAfter Both</td>
<td>Runnable</td>
<td>Completable Future&lt; Void&gt;</td>
<td>Run action when both previous stages complete</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by one or more previous stage(s)
  - 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 Either (Async)</td>
<td>Function</td>
<td>Completable Future with Function result</td>
<td>Apply function to results of either previous stage</td>
</tr>
<tr>
<td>accept Either (Async)</td>
<td>Consumer</td>
<td>Completable Future&lt;Void&gt;</td>
<td>Consumer handles results of either previous stage</td>
</tr>
<tr>
<td>runAfter Either (Async)</td>
<td>Runnable</td>
<td>Completable Future&lt;Void&gt;</td>
<td>Run action when either previous stage completes</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)
    { ... }
```
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

```java
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 `sync` 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
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

```java
CompletableFuture<? extends CompletionStage<?>> thenCompose(
    Function<? super T, ? extends CompletionStage<?>> 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\(<\text{U}\>\) `thenCompose`

\((\text{Function}\langle ? \; \text{super} \; \text{T}, \; ? \; \text{extends} \; \text{CompletionStage}\langle \text{U}\rangle \rangle \; \text{fn})\)

\{
  ... \}

27
### 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**

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

See [dzone.com/articles/understanding-flatmap](https://dzone.com/articles/understanding-flatmap)
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

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

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

  - 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()`

  `thenApplyAsync()` can often replace `thenCompose(supplyAsync())` nestings
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()

CompletableFuture<Integer> countF = CompletableFuture.supplyAsync(() -> longRunnerReturnsCF()).thenCompose(Function.identity());

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

Can be used to avoid calling join() when flattening nested completable futures
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(() -> longRunnerReturnsCF()).thenCompose(Function.identity());
```

This idiom flattens the return value to "just" one CompletableFuture!

Can be used to avoid calling `join()` when flattening nested completable futures
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(() ->
    longRunnerReturnsCF());

.thenComposeAsync(this::longBlockerReturnsCF)
...<br>

Runs `longBlockerReturnsCF()` in a thread in the fork-join pool
```

`thenComposeAsync()` can be used to avoid calling `supplyAsync()` again in a chain
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

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

*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)
    { ... }
```
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);

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);

println() is a callback that has a side-effect (i.e., printing the mixed string)
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
  - Often used at the end of a chain of completion stages
  - May lead to “callback hell!”

See dzone.com/articles/callback-hell
Methods triggered by completion of both of two previous stages

- thenCombine()

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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenCombine
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

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

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 asynchronous execution

```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);
```

`thenCombine()`’s action is triggered when its two associated futures complete
Key CompletableFuture Completion Stage Methods

- 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](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#acceptEither)
• Methods triggered by completion of either of two previous stages
• `acceptEither()`
  • Applies a consumer action that handles either of the previous stages' results
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 stages' results
    - Returns a future to Void

```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 stages' results
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

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

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

Create two completable futures that will contain the results of sorting the list using two different algorithms in two different threads
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 stages' results
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

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

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

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

Printout sorted results from which ever sorting routine finished first

`acceptEither()` does not cancel the second future after the first one completes
End of Overview of Advanced Java 8 Completable Future Features (Part 2)