

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

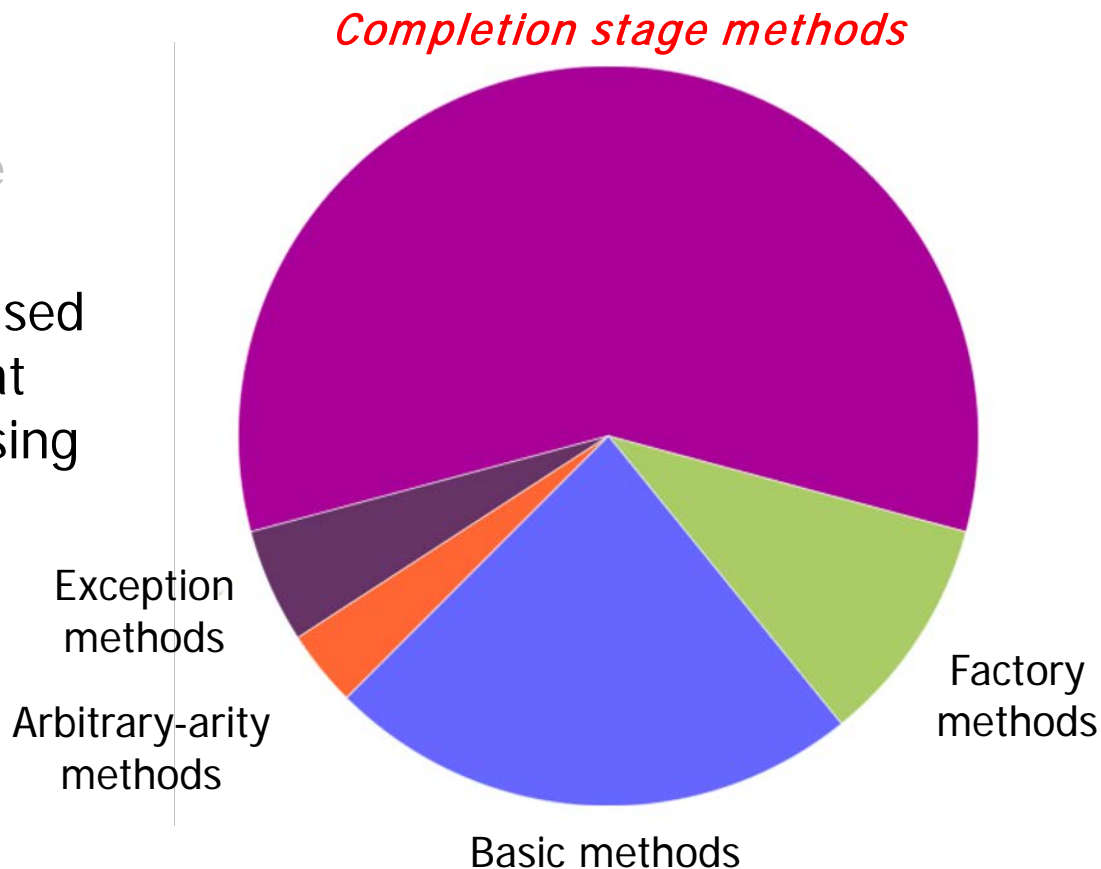
Institute for Software
Integrated Systems

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

Completion Stage Methods Chain Actions Together


- A completable future can serve as a "completion stage" for async result processing
- An action is performed 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()'s action is
triggered when future from
supplyAsync() completes*



Completion Stage Methods Chain Actions Together


- 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"

```
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

Completion Stage Methods Chain Actions Together

- 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

```
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
- An action is performed 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);
```



Invocation of a lambda action is "deferred" until previous future completes

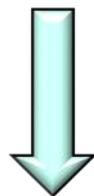
Completion Stage Methods Chain Actions Together

- 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

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



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

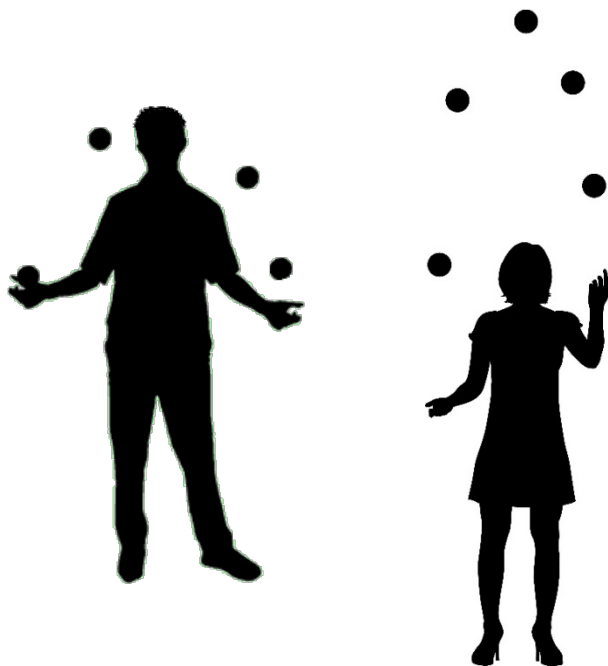
```
Supplier<BigFraction> reduce = () ->
    BigFraction.reduce(unreduced);
```

```
CompletableFuture
    .supplyAsync(reduce)
    .thenApply(BigFraction
        ::toMixedString)
    .thenAccept(System.out::println);
```

Completion stages avoid blocking a thread until the result *must* be obtained

Completion Stage Methods Chain Actions Together

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



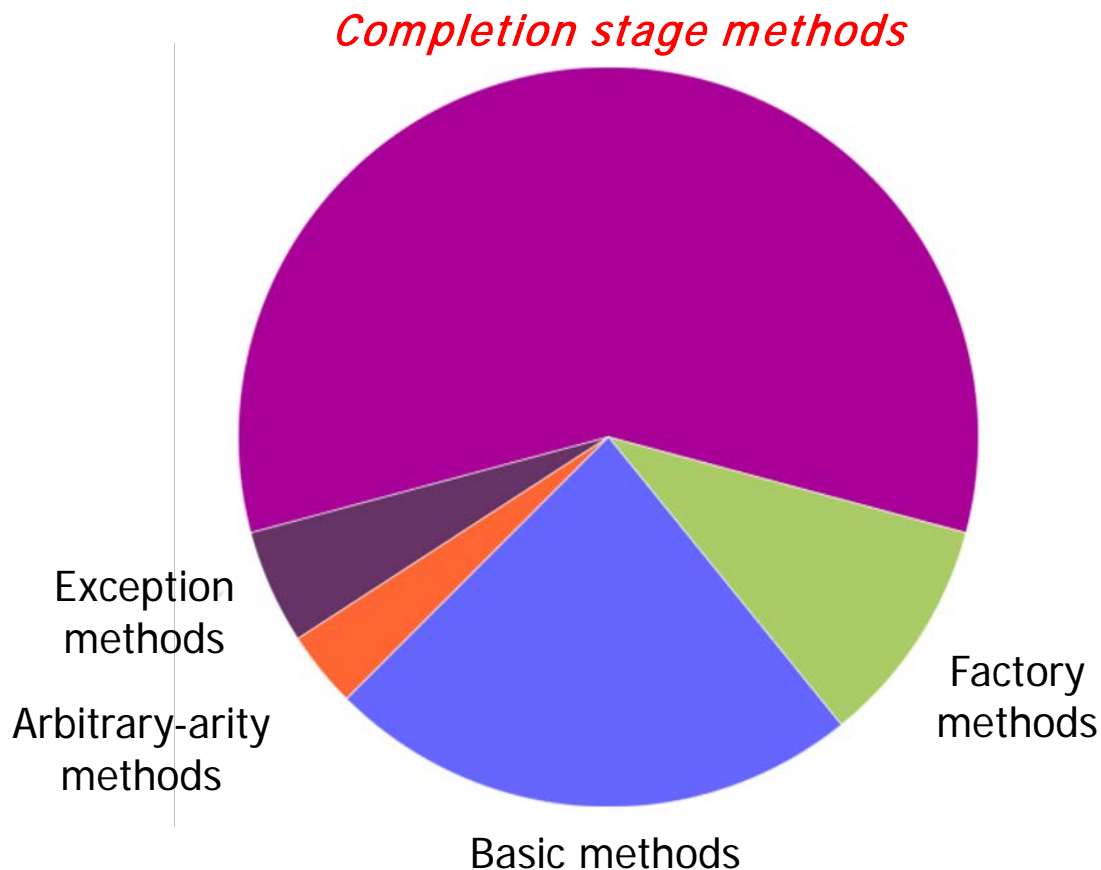
<<Java Class>>	
G CompletableFuture<T>	
•	CompletableFuture()
•	cancel(boolean):boolean
•	isCancelled():boolean
•	isDone():boolean
•	get()
•	get(long,TimeUnit)
•	join()
•	complete(T):boolean
•	supplyAsync(Supplier<U>):CompletableFuture<U>
•	supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
•	runAsync(Runnable):CompletableFuture<Void>
•	runAsync(Runnable,Executor):CompletableFuture<Void>
•	completedFuture(U):CompletableFuture<U>
•	thenApply(Function<?>):CompletableFuture<U>
•	thenAccept(Consumer<? super T>):CompletableFuture<Void>
•	thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
•	thenCompose(Function<?>):CompletableFuture<U>
•	whenComplete(BiConsumer<?>):CompletableFuture<T>
•	allOf(CompletableFuture[]<?>):CompletableFuture<Void>
•	anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Juggling is a good analogy for completion stages!

Grouping CompletableFuture Completion Stage Methods

Grouping CompletableFuture Completion Stage Methods

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



Grouping Completable Future 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

Methods	Params	Returns	Behavior
<code>thenApply</code> (Async)	Function	Completable Future with Function result	Apply function to result of the previous stage
<code>thenCompose</code> (Async)	Function	Completable Future with Function result directly, <i>not</i> a nested future	Apply function to result of the previous stage
<code>thenAccept</code> (Async)	Consumer	Completable Future<Void>	Consumer handles result of previous stage
<code>thenRun</code> (Async)	Runnable	Completable Future<Void>	Run action w/out returning value

See www.jesperdj.com/2015/09/26/the-future-is-completable-in-java-8

Grouping Completable Future 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

Methods	Params	Returns	Behavior
<code>thenApply</code> (<code>Async</code>)	Function	Completable Future with Function result	Apply function to result of the previous stage
<code>thenCompose</code> (<code>Async</code>)	Function	Completable Future with Function result directly, <i>not</i> a nested future	Apply function to result of the previous stage
<code>thenAccept</code> (<code>Async</code>)	Consumer	Completable Future<Void>	Consumer handles result of previous stage
<code>thenRun</code> (<code>Async</code>)	Runnable	Completable Future<Void>	Run action w/out returning value

`Async()` variants run in common fork-join thread (by default, run in same thread)

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
 - i.e., an “and”

Methods	Params	Returns	Behavior
<code>thenCombine</code> (Async)	Bi Function	Completable Future with Bi Function result	Apply bifunction to results of both previous stages
<code>thenAcceptBoth</code> (Async)	Bi Consumer	Completable Future<Void>	BiConsumer handles results of both previous stages
<code>runAfterBoth</code> (Async)	Runnable	Completable Future<Void>	Run action when both previous stages complete

Grouping Completable Future 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”

Methods	Params	Returns	Behavior
<code>applyToEither(Async)</code>	<code>Function</code>	<code>CompletableFuture</code> with <code>Function</code> result	Apply function to results of either previous stage
<code>acceptEither(Async)</code>	<code>Consumer</code>	<code>CompletableFuture<Void></code>	Consumer handles results of either previous stage
<code>runAfterEither(Async)</code>	<code>Runnable</code>	<code>CompletableFuture<Void></code>	Run action when either previous stage completes

Key CompletableFuture Completion Stage Methods

Key CompletableFuture Completion Stage Methods

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

```
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

```
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

```
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

```
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)
    { ... }
```

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

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

Key CompletableFuture Completion Stage Methods

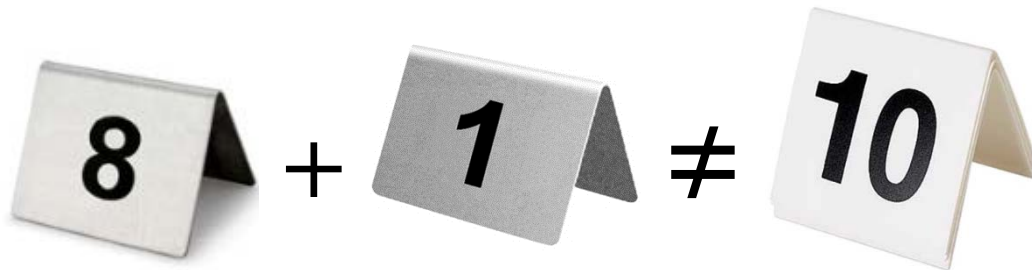
- Methods triggered by completion of a single previous stage

- `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  
  (Function<? super T,  
    ? extends  
    CompletionStage<U>> fn)  
  { ... }
```



`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()`

- 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

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

e.g., supplyAsync() returns a completable future

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!

```
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()`

```
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

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  
            ( () ->  
                returnCompletableFuture() )  
  
    .thenCompose  
        ( Function.identity() )  
  
    ...
```

This idiom flattens the return value to "just" a 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()`
 - `thenAccept()`

```
CompletableFuture<Void>
```

```
    thenAccept
```

```
        (Consumer<? super T> action)
```

```
    { ... }
```

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

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

*This action behaves as a
"callback" with a side-effect*

See [en.wikipedia.org/wiki/Callback_\(computer_programming\)](https://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`

```
CompletableFuture<Void>
```

```
    thenAccept
```

```
        (Consumer<? super T> action)
```

```
    { ... }
```

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

```
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

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

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

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

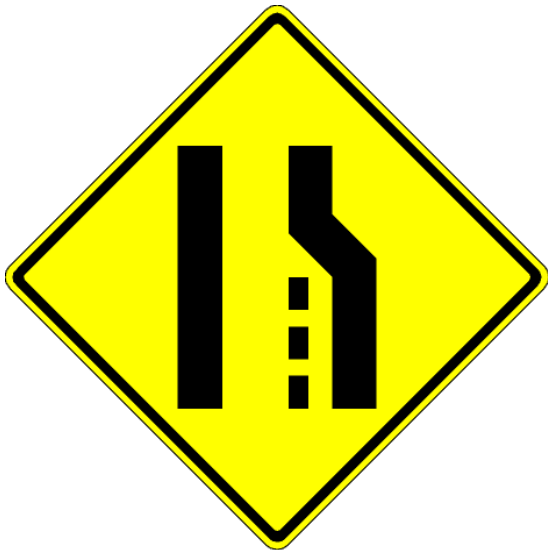
```
CompletableFuture  
    .supplyAsync(reduce)  
    .thenApply(BigFraction  
                ::toMixedString)  
    .thenAccept(System.out::println);
```

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 both of two previous stages
- `thenCombine()`

```
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

```
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

```
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

```
CompletableFuture<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

```
CompletableFuture<BF> compF1 =  
    CompletableFuture  
        .supplyAsync(() ->  
            /* multiply two BF's. */);
```

```
CompletableFuture<BF> compF2 =  
    CompletableFuture  
        .supplyAsync(() ->  
            /* divide two BF's. */);
```

```
compF1  
    .thenCombine(compF2,  
                BigFraction::add)  
    .thenAccept(System.out::println);
```


Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of either of two previous stages
- `acceptEither()`

```
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

```
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

```
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())));
```

*Printout sorted results from which
ever sorting routine finished first*

End of Overview of Advanced Java 8 Completable Future Features (Part 2)