

Advanced Java CompletableFuture

Features: Introducing Factory Methods

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

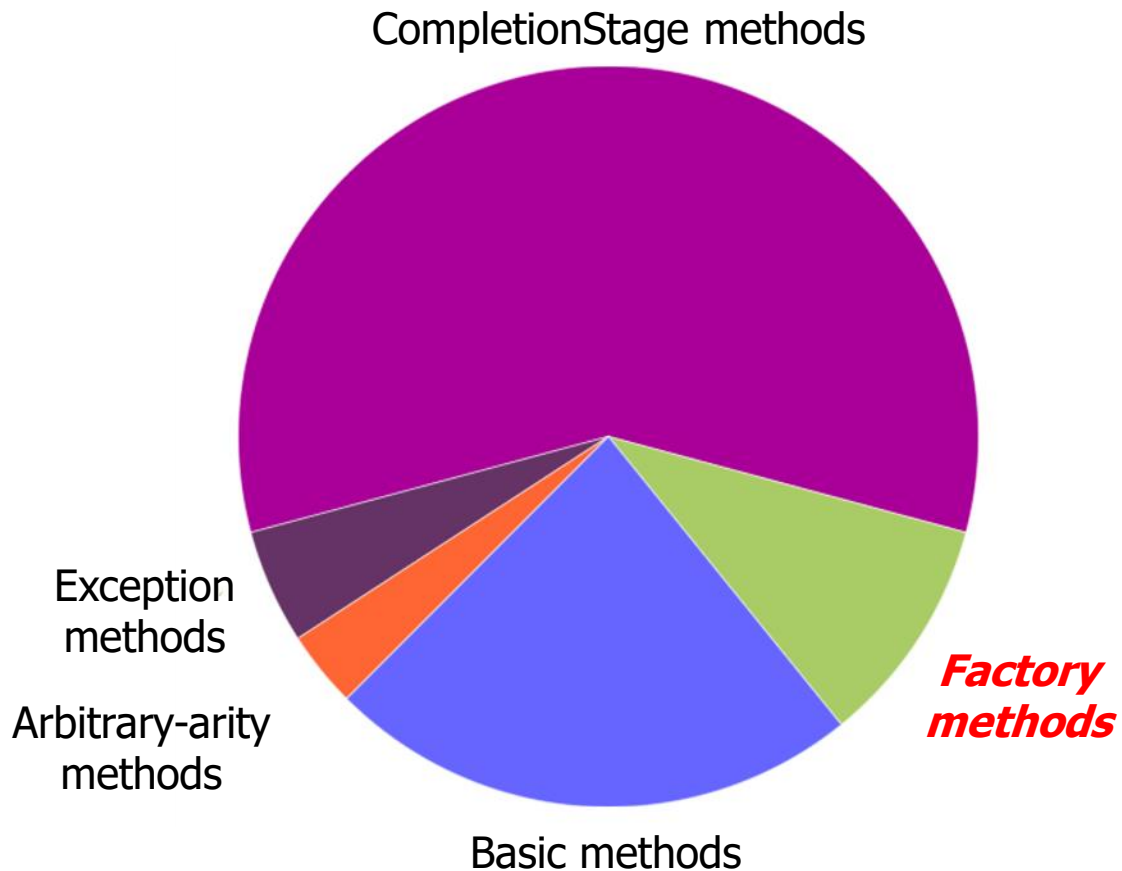
**Institute for Software
Integrated Systems**

**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



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

Factory Methods Initiate Async Computations

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations



<<Java Class>>	
G CompletableFuture<T>	
•	CompletableFuture()
•	cancel(boolean):boolean
•	isCancelled():boolean
•	isDone():boolean
•	get()
•	get(long,TimeUnit)
•	join()
•	complete(T):boolean
S	supplyAsync(Supplier<U>):CompletableFuture<U>
S	supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
S	runAsync(Runnable):CompletableFuture<Void>
S	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>
S	allOf(CompletableFuture[]<?>):CompletableFuture<Void>
S	anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

See en.wikipedia.org/wiki/Factory_method_pattern

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
- These computations may or may not return a value



<<Java Class>>	
G CompletableFuture<T>	
•	CompletableFuture()
•	cancel(boolean):boolean
•	isCancelled():boolean
•	isDone():boolean
•	get()
•	get(long,TimeUnit)
•	join()
•	complete(T):boolean
S	supplyAsync(Supplier<U>):CompletableFuture<U>
S	supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
S	runAsync(Runnable):CompletableFuture<Void>
S	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>
S	allOf(CompletableFuture[]<?>):CompletableFuture<Void>
•	anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
 - `supplyAsync()` allows two-way calls via a supplier



Methods	Params	Returns	Behavior
<code>supplyAsync</code>	<code>Supplier</code>	<code>Completable Future</code> with result of <code>Supplier</code>	Asynchronously run supplier in common fork/join pool
<code>supplyAsync</code>	<code>Supplier, Executor</code>	<code>Completable Future</code> with result of <code>Supplier</code>	Asynchronously run supplier in given executor pool

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
- These computations may or may not return a value
 - `supplyAsync()` allows two-way calls via a supplier
 - Can be passed params

```
String f1 = "62675744/15668936";  
String f2 = "609136/913704";
```

```
CompletableFuture<BigFraction> future  
    = CompletableFuture  
        .supplyAsync(() -> {  
            BigFraction bf1 =  
                new BigFraction(f1);  
            BigFraction bf2 =  
                new BigFraction(f2);  
  
            return bf1.multiply(bf2);  
        }) ;
```

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
 - supplyAsync() allows two-way calls via a supplier
 - Can be passed params

```
String f1 = "62675744/15668936";
```

```
String f2 = "609136/913704";
```

```
CompletableFuture<BigFraction> future  
    = CompletableFuture  
        .supplyAsync(() -> {  
            BigFraction bf1 =  
                new BigFraction(f1);  
            BigFraction bf2 =  
                new BigFraction(f2);  
  
            return bf1.multiply(bf2);  
        });
```

Params are passed as "effectively final" objects to the supplier lambda

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
- These computations may or may not return a value
 - `supplyAsync()` allows two-way calls via a supplier
 - Can be passed params
 - Returns a value

```
String f1 = "62675744/15668936";  
String f2 = "609136/913704";
```

```
CompletableFuture<BigFraction> future  
    = CompletableFuture  
        .supplyAsync(() -> {  
            BigFraction bf1 =  
                new BigFraction(f1);  
            BigFraction bf2 =  
                new BigFraction(f2);  
  
            return bf1.multiply(bf2);  
        }) ;
```

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
 - `supplyAsync()` allows two-way calls via a supplier
 - `runAsync()` enables one-way calls via a runnable



Methods	Params	Returns	Behavior
<code>runAsync</code>	<code>Runnable</code>	<code>CompletableFuture</code> with result of <code>Void</code>	Asynchronously run runnable in common fork/join pool
<code>runAsync</code>	<code>Runnable, Executor</code>	<code>CompletableFuture</code> with result of <code>Void</code>	Asynchronously run runnable in given executor pool

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations

- These computations may or may not return a value

- `supplyAsync()` allows two-way calls via a supplier
- `runAsync()` enables one-way calls via a runnable
 - Can be passed params

```
String f1 = "62675744/15668936";  
String f2 = "609136/913704";
```

```
CompletableFuture<Void> future  
    = CompletableFuture  
        .runAsync(() -> {  
            BigFraction bf1 =  
                new BigFraction(f1);  
            BigFraction bf2 =  
                new BigFraction(f2);  
  
            System.out.println  
                (bf1.multiply(bf2)  
                    .toMixedString());  
        }) ;
```

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
- These computations may or may not return a value
 - `supplyAsync()` allows two-way calls via a supplier
 - `runAsync()` enables one-way calls via a runnable
 - Can be passed params
 - Returns no value

```
String f1 = "62675744/15668936";  
String f2 = "609136/913704";
```

```
CompletableFuture<Void> future  
    = CompletableFuture  
        .runAsync(() -> {  
            BigFraction bf1 =  
                new BigFraction(f1);  
            BigFraction bf2 =  
                new BigFraction(f2);  
  
            System.out.println  
                (bf1.multiply(bf2)  
                    .toMixedString());  
        }) ;
```

*"Void" is not
a value!*

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
- These computations may or may not return a value
 - `supplyAsync()` allows two-way calls via a supplier
 - `runAsync()` enables one-way calls via a runnable
 - Can be passed params
 - Returns no value

```
String f1 = "62675744/15668936";  
String f2 = "609136/913704";
```

```
CompletableFuture<Void> future  
    = CompletableFuture  
        .runAsync(() -> {  
            BigFraction bf1 =  
                new BigFraction(f1);  
            BigFraction bf2 =  
                new BigFraction(f2);
```

```
            System.out.println  
                (bf1.multiply(bf2)  
                    .toMixedString())  
        }  
    );
```



Any output must therefore come from "side-effects"

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
 - `supplyAsync()` allows two-way calls via a supplier
 - `runAsync()` enables one-way calls via a runnable



`supplyAsync()` is more commonly used than `runAsync()` in practice

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
- Async functionality runs in a thread pool



<<Java Class>>	
G CompletableFuture<T>	
•	CompletableFuture()
•	cancel(boolean):boolean
•	isCancelled():boolean
•	isDone():boolean
•	get()
•	get(long,TimeUnit)
•	join()
•	complete(T):boolean
S	supplyAsync(Supplier<U>):CompletableFuture<U>
S	supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
S	runAsync(Runnable):CompletableFuture<Void>
S	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>
S	allOf(CompletableFuture[]<?>):CompletableFuture<Void>
S	anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Help make programs more *elastic* by leveraging a pool of worker threads

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
- Async functionality runs in a thread pool



<<Java Class>>
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>

By default, the common fork-join pool is used

See dzone.com/articles/common-fork-join-pool-and-streams

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
- Async functionality runs in a thread pool



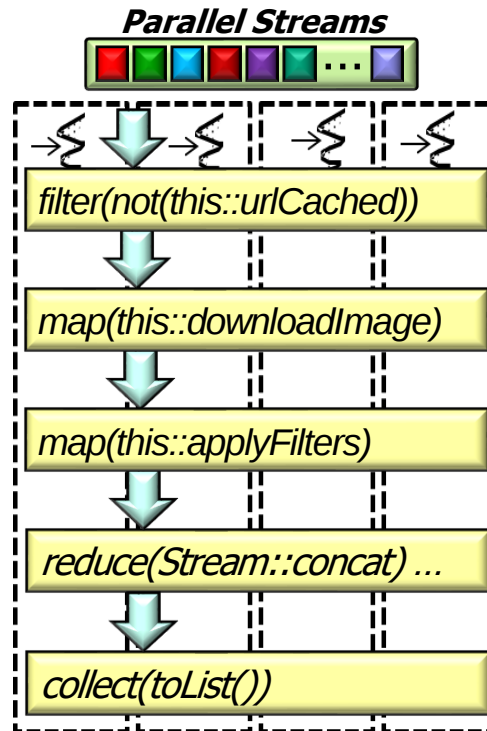
<<Java Class>>
CompletableFuture<T>

```
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>
```

However, a pre- or user-defined thread pool can also be given

Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
 - These computations may or may not return a value
- Async functionality runs in a thread pool
 - In contrast, Java parallel streams are designed for use with the common fork-join pool



See lesson on "Java Parallel Stream Internals: Parallel Processing via the Common Fork-Join Pool"

End of Advanced Java

CompletableFuture Features: Introducing Factory Methods