Overview of Java 8 CompletableFutures (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 the basic completable futures features
- Understand several advanced completable futures features

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage<T>, Future<T>

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

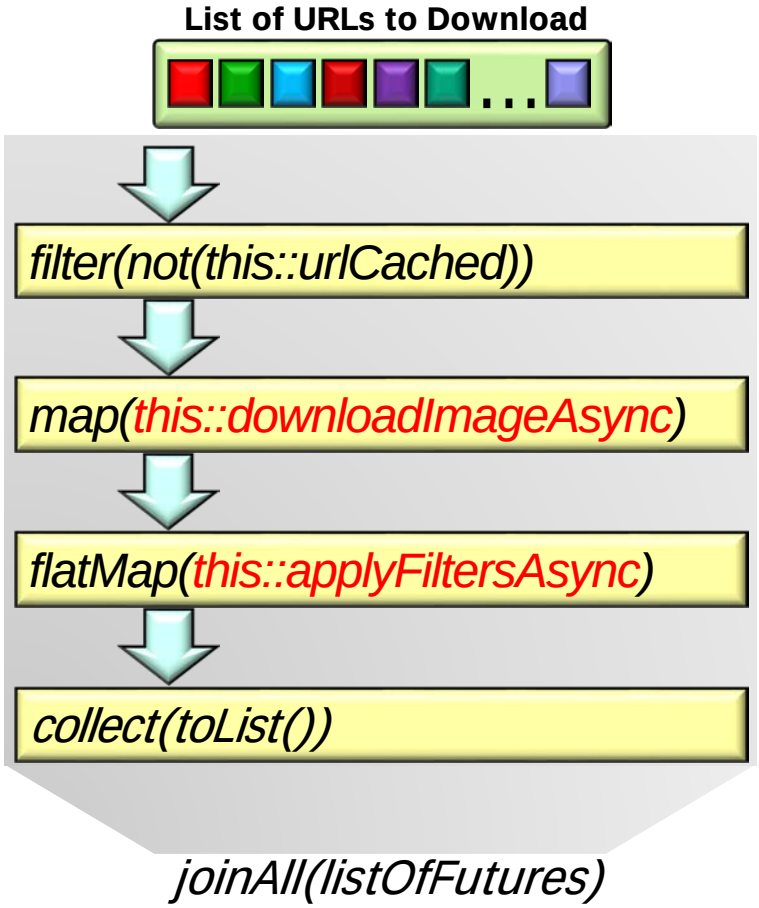
A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.

When two or more threads attempt to complete, completeExceptionally, or cancel a CompletableFuture, only one of them succeeds.

In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

Learning Objectives in this Part of the Lesson

- Understand the basic completable futures features
- Understand several advanced completable futures features
  - Methods from a completable futures implementation of ImageStreamGang are used as examples



Summary of Advanced Completable Futures Features

Summary of Advanced Completable Futures Features

- Completable futures have several advanced features

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage<T>, Future<T>

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.

When two or more threads attempt to complete, completeExceptionally, or cancel a CompletableFuture, only one of them succeeds.

In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Advanced Completable Futures Features

- Completable futures have several advanced features
  - Initiate asynchronous two-way or one-way functions

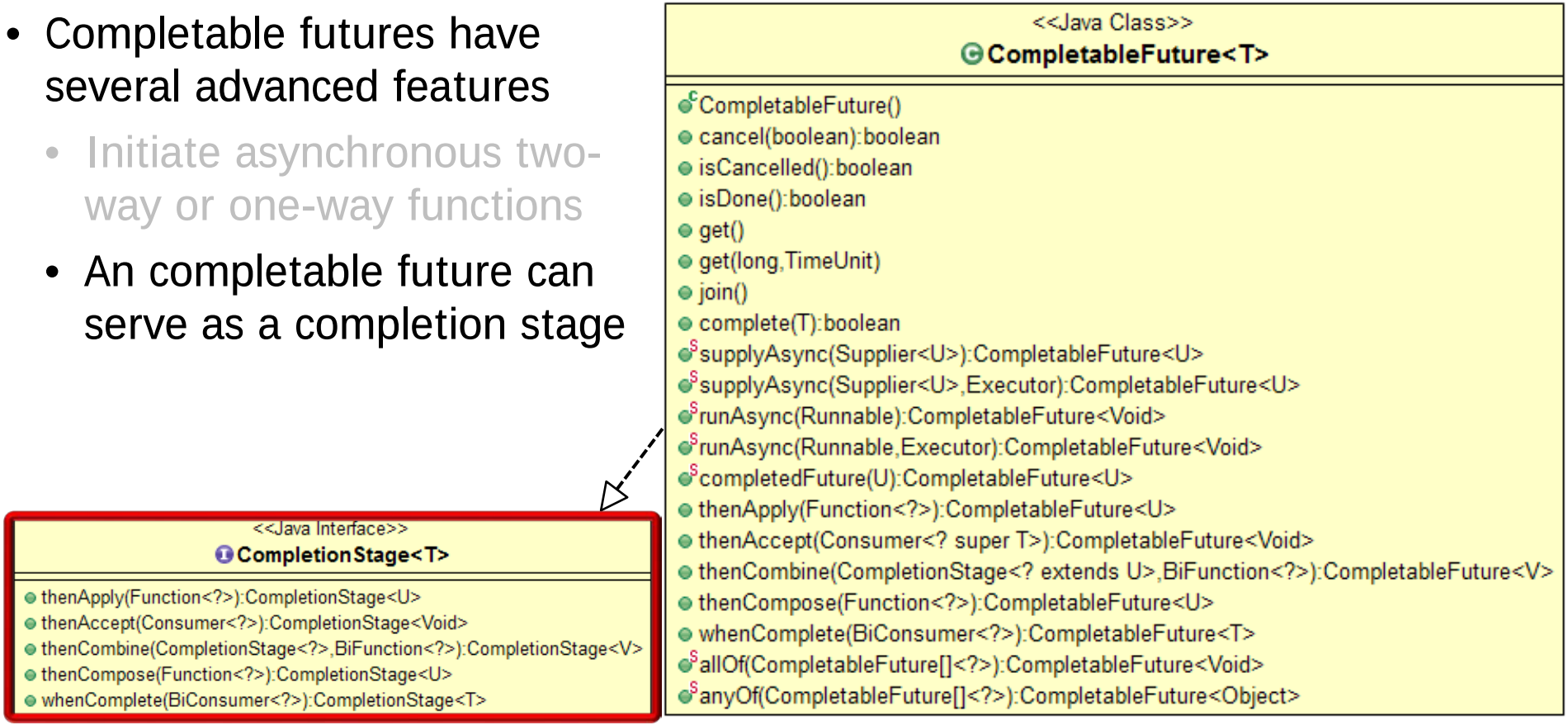
<<Java Class>>

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>

Advanced Completable Futures Features

- Completable futures have several advanced features
  - Initiate asynchronous two-way or one-way functions
  - An completable future can serve as a completion stage



Advanced Completable Futures Features

- Completable futures have several advanced features
  - Initiate asynchronous two-way or one-way functions
  - An completable future can serve as a completion stage
  - Provide “arbitrary-arity” methods

```
<<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>
```

See en.wikipedia.org/wiki/Arity

Initiating Asynchronous One-way or Two-way Functions

Initiating Asynchronous One-way or Two-way Functions

- An completable future can initiate an asynchronous two-way or one-way function

```
<<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>
```

Initiating Asynchronous One-way or Two-way Functions

- An completable future can initiate an asynchronous two-way or one-way function
  - This code runs in a thread pool

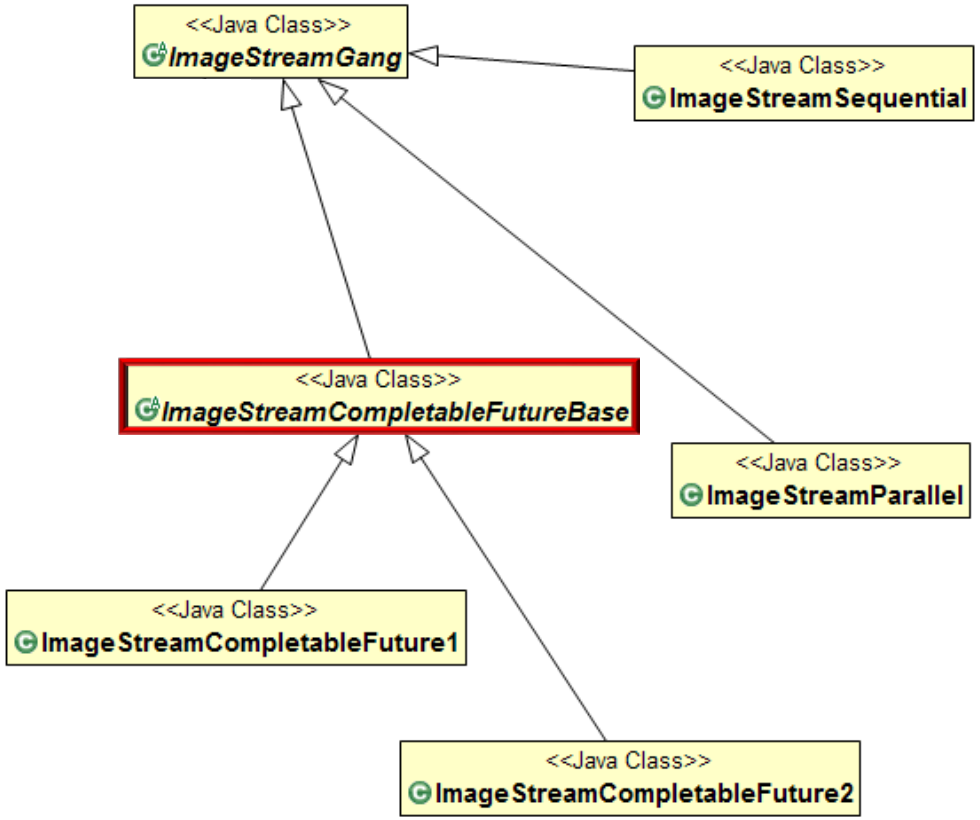

```
<<Java Class>>
CompletableFuture<T>

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>
```

This thread pool defaults to common fork-join pool, but can be given explicitly

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places



See app/src/main/java/livelessons/imagestreamgang/streams/ImageStreamCompletableFutureBase.java

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()

```
CompletableFuture<Image>
  downloadImageAsync(URL url) {
  return CompletableFuture
         .supplyAsync(() ->
                 downloadImage(url),
                     getExecutor());
}
```

Asynchronously download image at the given URL

See app/src/main/java/livelessons/imagestreamgang/streams/ImageStreamCompletableFutureBase.java

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()

```
CompletableFuture<Image>
  downloadImageAsync(URL url) {
  return CompletableFuture
         .supplyAsync(() ->
                 downloadImage(url),
                     getExecutor());

}
```

Run the downloadImage() method asynchronously

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()

```
CompletableFuture<Image>
  downloadImageAsync(URL url) {
  return CompletableFuture
         .supplyAsync(() ->
                      downloadImage(url),
                      getExecutor());
}
```

Specify a fixed-size thread pool executor

You could also simply use the default thread pool (common fork-join pool)

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()

```
CompletableFuture<Image>
  downloadImageAsync(URL url) {
  return CompletableFuture
         .supplyAsync(() ->
                 downloadImage(url),
                     getExecutor());
}
```

Returns a completable future to an image that triggers when image downloading is finished

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()
  - filterImageAsync()

```
CompletableFuture<Image>
  filterImageAsync
    (FilterDecoratorWithImage
     filterDecoratorWithImage) {
  return CompletableFuture
         .supplyAsync
            (filterDecoratorWithImage
             ::run,
            getExecutor());
```

Asynchronous filter an image & store it into a file

See app/src/main/java/livelessons/imagestreamgang/streams/ImageStreamCompletableFutureBase.java

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()
  - filterImageAsync()

```
CompletableFuture<Image>
  filterImageAsync
    (FilterDecoratorWithImage
     filterDecoratorWithImage) {
  return CompletableFuture
         .supplyAsync
            (filterDecoratorWithImage
             ::run,
            getExecutor());
}
```

asynchronously run the filterDectoratorWithImage.run() method

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()
  - filterImageAsync()

```
CompletableFuture<Image>
  filterImageAsync
    (FilterDecoratorWithImage
     filterDecoratorWithImage) {
  return CompletableFuture
         .supplyAsync
            (filterDecoratorWithImage
             ::run,
            getExecutor());
}
```

Specify a fixed-size thread pool executor

Initiating Asynchronous One-way or Two-way Functions

- supplyAsync() is used by the ImageStreamGang app in several places, e.g.
  - downloadImageAsync()
  - filterImageAsync()

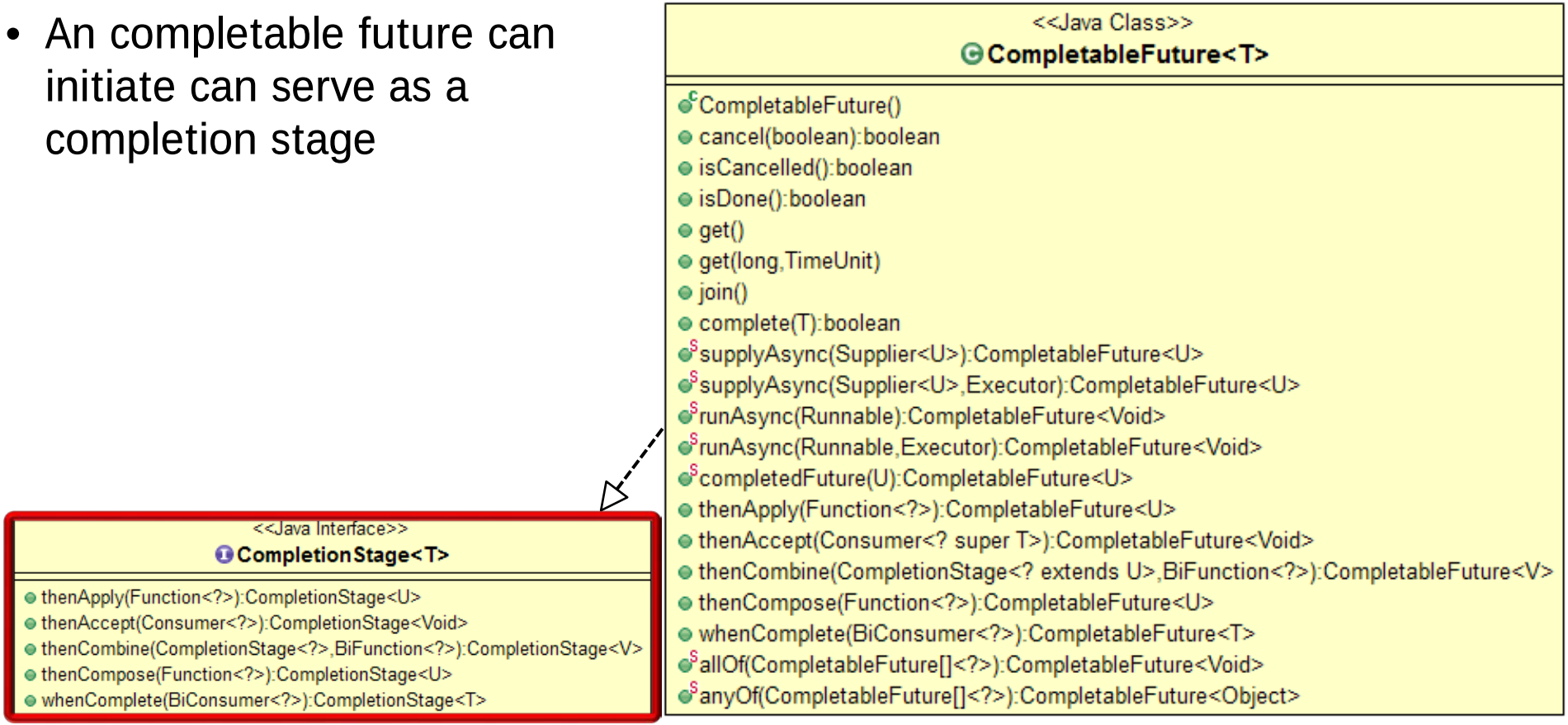
```
CompletableFuture<Image>
  filterImageAsync
    (FilterDecoratorWithImage
     filterDecoratorWithImage) {
  return CompletableFuture
         .supplyAsync
            (filterDecoratorWithImage
             ::run,
            getExecutor());
}
```

Returns a completable future to an image that triggers when image filtering/store is finished

Serving as a Completion Stage

Serving as a Completion Stage

- An completable future can initiate can serve as a completion stage



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

Serving as a Completion Stage

- An completable future can initiate can serve as a completion stage
  - Perform an action after an earlier completion stage completes

<<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>

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

Serving as a Completion Stage

- An completable future can initiate can serve as a completion stage
  - Perform an action after an earlier completion stage completes
  - May trigger dependent completion stages after async functions complete

<<Java Class>>

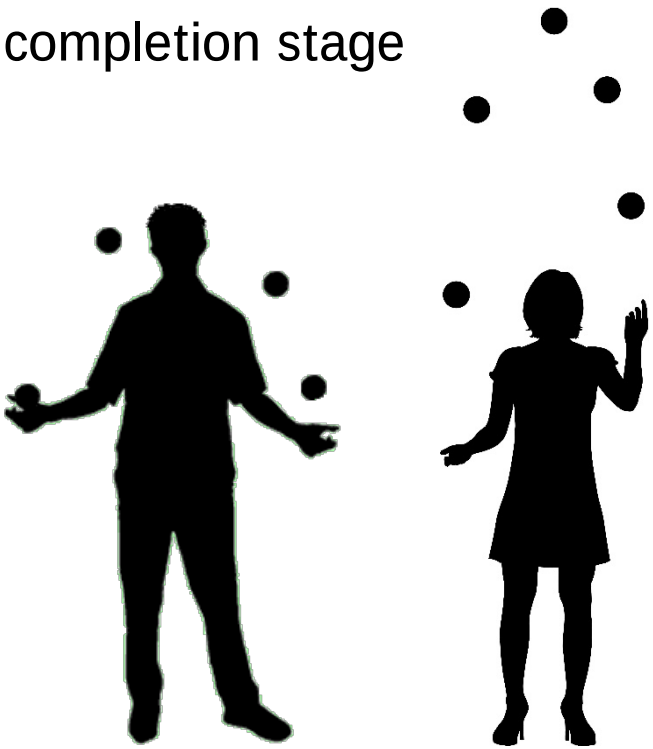
CompletableFuture<T>

- 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>

A goal of completion stages is to avoid blocking until the result must be obtained

Serving as a Completion Stage

- An completable future can initiate can serve as a completion stage

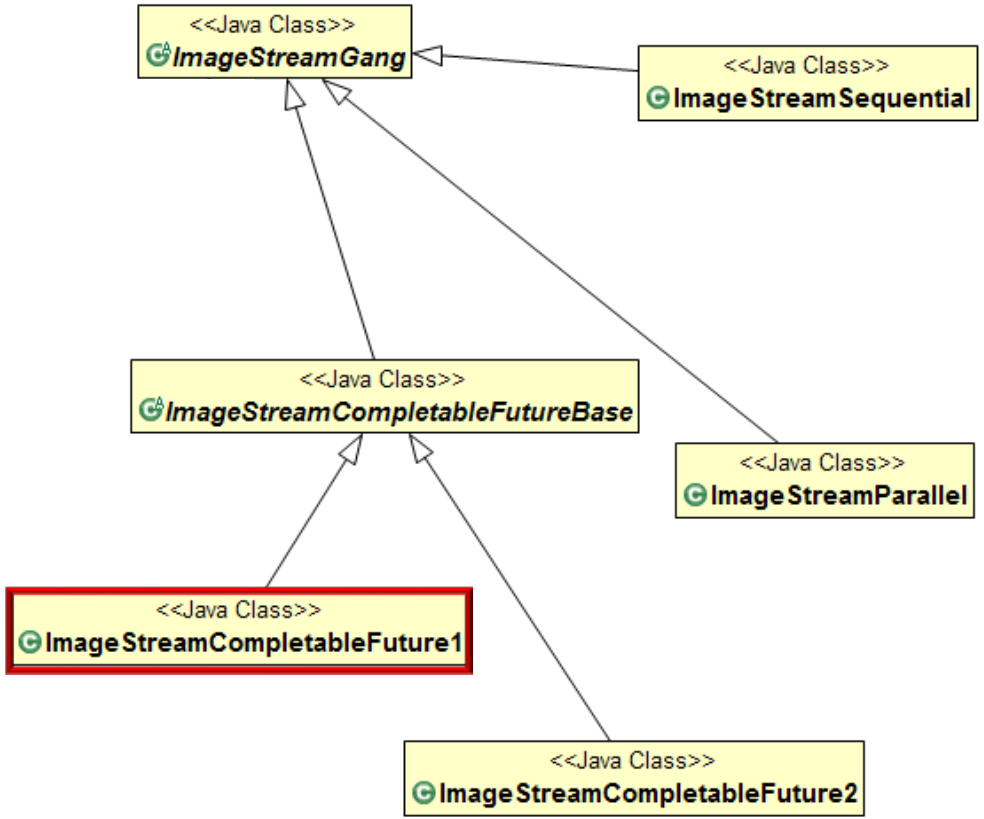


```
<<Java Class>>
CompletableFuture<T>
CompletableFuture()
cancel(boolean):boolean
isCancelled():boolean
isDone():boolean
get()
get(long,TimeUnit)
join()
complete(T):boolean
supplyAsync(Supplier<U>):CompletableFuture<U>
supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
runAsync(Runnable):CompletableFuture<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>
```

Juggling is a good analogy for completion stages!

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places



See app/src/main/java/livelessons/imagestreamgang/streams/ImageStreamCompletableFuture1.java

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
         (filter -> CompletableFuture
          .supplyAsync(filter::run,
                       getExecutor())));
}
```

Asynchronous filter images & store them into files

See app/src/main/java/livelessons/imagestreamgang/streams/ImageStreamCompletableFuture1.java

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
        makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
        filterFuture.thenCompose
          (filter -> CompletableFuture
           .supplyAsync(filter::run,
                        getExecutor())));
}
```

Two completion stage methods

See www.youtube.com/watch?v=-MBPQ7NIL_Y

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

This is the completable future returned from downloadImageAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
         (filter -> CompletableFuture
          .supplyAsync(filter::run,
                       getExecutor())));
}
```

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

Convert this list of filters into a stream

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
     .map(filter -> imFuture.thenApply
       (image ->
        makeFilterDecoratorWithImage
           (filter, image)))

     .map(filterFuture ->
        filterFuture.thenCompose
          (filter -> CompletableFuture
           .supplyAsync(filter::run,
                        getExecutor())));
}
```

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
        makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
        filterFuture.thenCompose
          (filter -> CompletableFuture
           .supplyAsync(filter::run,
                        getExecutor())));
}
```

Create a completable future to a FilterDecoratorWithImage object for each filter/image

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
         (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
         (filter -> CompletableFuture
          .supplyAsync(filter::run,
                       getExecutor())));
}
```

thenApply() defines a computation that's not executed immediately, but is remembered & executed when imFuture completes

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

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
         (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
         (filter -> CompletableFuture
          .supplyAsync(filter::run,
                       getExecutor())));
}
```

Returns a new completion stage that (when completed normally) is executed with this stage's result as the argument to the supplied lambda expression

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

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
         (filter -> CompletableFuture
          .supplyAsync(filter::run,
                       getExecutor())));
}
```

Asynchronously filter the image & store it in an output file

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
          (filter -> CompletableFuture
           .supplyAsync(filter::run,
                        getExecutor())));
}
```

thenCompose() can be used to combine two completable future together without blocking or waiting for intermediate results

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

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
        (filter -> CompletableFuture
         .supplyAsync(filter::run,
                      getExecutor())));
}
```

It also returns a new completion stage that (when completed normally) is executed with this stage's result as the argument to the supplied lambda expression

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

Serving as a Completion Stage

- Completion states are used by ImageStreamGang in several places, e.g.
  - applyFiltersAsync()

Returns a stream of completable futures to filtered/store images

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
  (CompletableFuture<Image> imFuture){
  return mFilters.stream()
    .map(filter -> imFuture.thenApply
      (image ->
       makeFilterDecoratorWithImage
          (filter, image)))

    .map(filterFuture ->
       filterFuture.thenCompose
         (filter -> CompletableFuture
          .supplyAsync(filter::run,
                       getExecutor())));
}
```

The flatMap() operation processes this stream return value, as we'll show shortly

End of Overview of Java 8 Completable Futures (Part 2)