Java 8 CompletableFutures

ImageStreamGang Example (Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

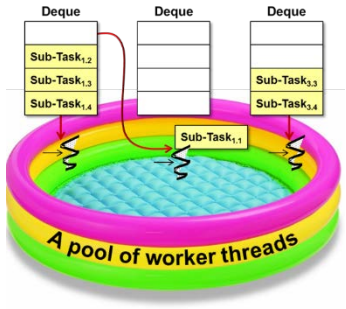
Professor of Computer Science

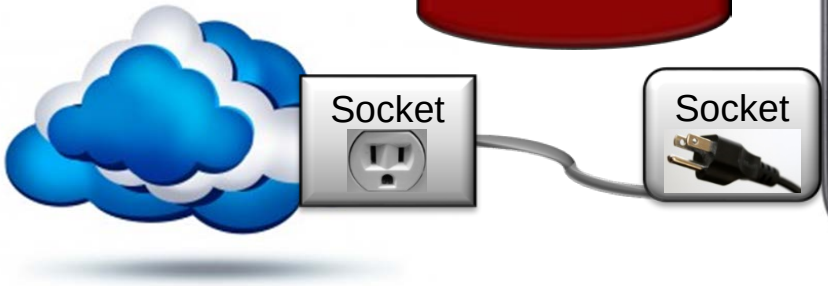
Institute for Software
Integrated Systems

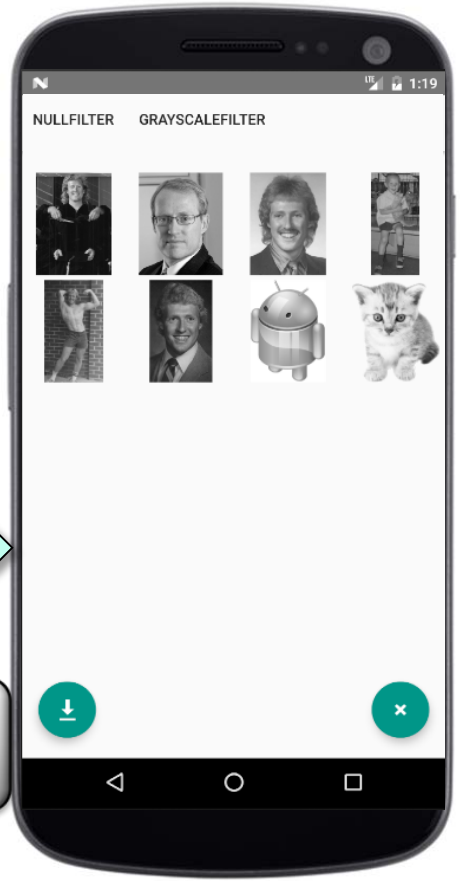
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand the design of the Java 8 completable future version of ImageStreamGang
- Know how completable futures are applied to ImageStreamGang

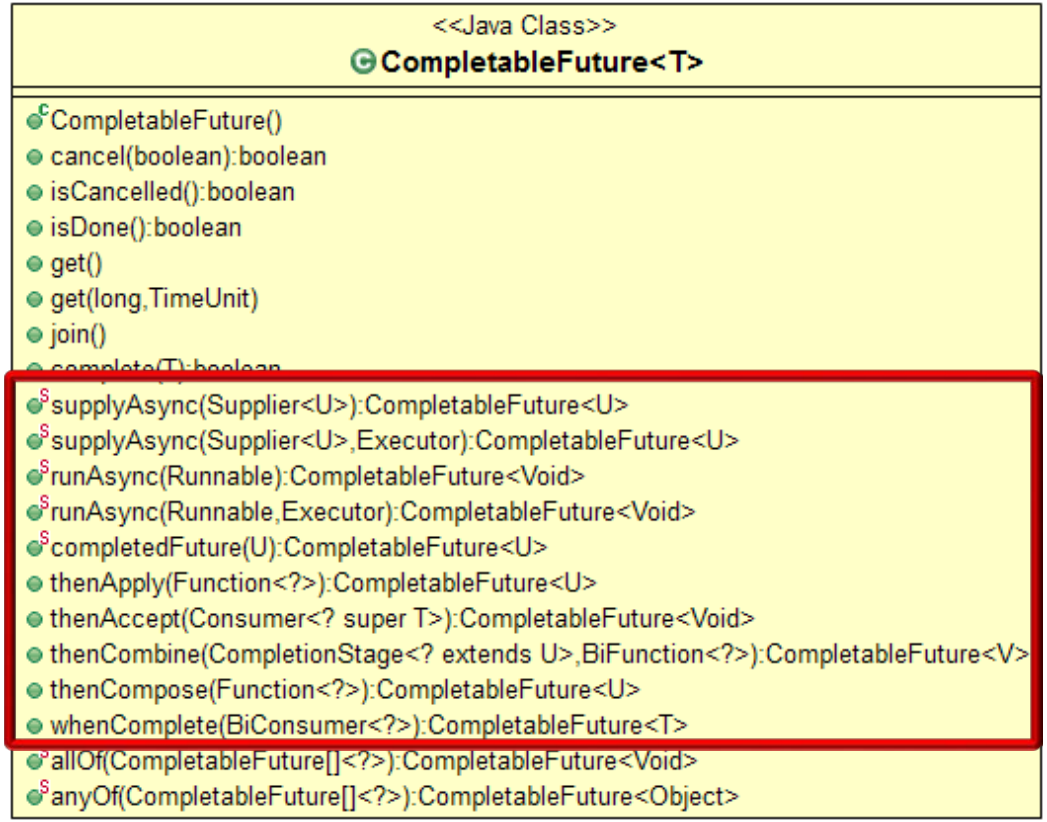






Learning Objectives in this Part of the Lesson

- Understand the design of the Java 8 completable future version of ImageStreamGang
- Know how completable futures are applied to ImageStreamGang
  - Factory methods & completion stage methods



Applying Completable Futures to ImageStreamGang

Applying Completable Futures to ImageStreamGang

- Focus on processStream()

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
  ...
```

See imagestreamgang/streams/ImageStreamCompletableFuture1.java

Applying Completable Futures to ImageStreamGang

- Focus on processStream()

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
...
```

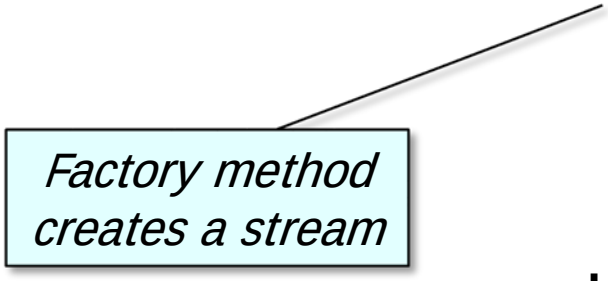
Combines a Java 8 sequential stream with completable futures

Applying Completable Futures to ImageStreamGang

- Focus on processStream()

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
...
```



Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This intermediate operation is identical to parallel streams

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
...
```

Ignore images that are already in the cache

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This intermediate operation is identical to parallel streams

```
boolean urlCached(URL url) {
  return mFilters
    .stream()
    .filter(filter ->
       urlCached(url,
          filter.getName()))
    .count() > 0;
}
```

```
void processStream() {
  List<URL> urls = getInput();

List<CompletableFuture<Image>>
listOfFutures = urls
   .stream()
   .filter(not(this::urlCached))
   .map(this::downloadImageAsync)
   .flatMap(this::applyFiltersAsync)
   .collect(toList());
...
```

See imagestreamgang/streams/ImageStreamGang.java

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This intermediate operation is identical to parallel streams

```
boolean urlCached(URL url,
          String filterName) {
  File file =
    new File(getPath(),
             filterName);

  File imageFile =
    new File(file,
          getNameForUrl(url));

  return imageFile.exists();
}
```

```
void processStream() {
  List<URL> urls = getInput();

List<CompletableFuture<Image>>
listOfFutures = urls
  .stream()
  .filter(not(this::urlCached))
  .map(this::downloadImageAsync)
  .flatMap(this::applyFiltersAsync)
  .collect(toList());
...
```

There are clearly more sophisticated ways of implementing an image cache!

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This intermediate operation is identical to parallel streams
  - Other intermediate operations differ since they input/output stream of completable futures

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
  ...
```

Applying Factory Methods in ImageStreamGang

Applying Factory Methods in ImageStreamGang

- map() calls the behavior downloadImageAsync() to asynchronously download each URL in the input stream

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
  ...
```

Applying Factory Methods in ImageStreamGang

- map() calls the behavior downloadImageAsync() to asynchronously download each URL in the input stream

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
...
```

Asynchronously downloads an image & stores it in memory

Applying Factory Methods in ImageStreamGang

- map() calls the behavior downloadImageAsync() to asynchronously download each URL in the input stream

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
  ...
```

Returns a completable future to each downloaded image in the output stream

Applying Factory Methods in ImageStreamGang

- downloadImageAsync() uses the supplyAsync() factory method internally

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

Asynchronously download image at the given URL

Applying Factory Methods in ImageStreamGang

- downloadImageAsync() uses the supplyAsync() factory method internally

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

Run the downloadImage() method asynchronously

Applying Factory Methods in ImageStreamGang

- downloadImageAsync() uses the supplyAsync() factory method internally

```
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)

Applying Factory Methods in ImageStreamGang

- downloadImageAsync() uses the supplyAsync() factory method internally

```
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

Applying Completion Stage Methods in ImageStreamGang

Applying Completion Stage Methods in ImageStreamGang

- flatMap() calls the behavior applyFiltersAsync() to filter & store each downloaded image in the input stream

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
  ...
```

Applying Completion Stage Methods in ImageStreamGang

- flatMap() calls the behavior applyFiltersAsync() to filter & store each downloaded image in the input stream

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
...
```

Asynchronous filter images & store them into files

Applying Completion Stage Methods in ImageStreamGang

- flatMap() calls the behavior applyFiltersAsync() to filter & store each downloaded image in the input stream

```
void processStream() {
  List<URL> urls = getInput();

  List<CompletableFuture<Image>>
  listOfFutures = urls
    .stream()
    .filter(not(this::urlCached))
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toList());
...
```

Flatten all filtered/stored images into a single output stream

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
         (FilterDecoratorWithImage
          ::run,
          getExecutor()));
}
```

Asynchronous filter images & store them into files

See imagestreamgang/streams/ImageStreamCompletableFuture1.java

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

This completable future is what's returned from downloadImageAsync()

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
         (FilterDecoratorWithImage
          ::run,
          getExecutor()));
}
```

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
      filterFuture.thenApplyAsync
        (FilterDecoratorWithImage
         ::run,
         getExecutor()));}
```

Two completion stage methods

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

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

     .map(filterFuture ->
        filterFuture.thenApplyAsync
          (FilterDecoratorWithImage
           ::run,
           getExecutor()));
}
```

Convert this list of filters into a stream

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
      filterFuture.thenApplyAsync
        (FilterDecoratorWithImage
         ::run,
         getExecutor()));
}
```

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

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
         (FilterDecoratorWithImage
          ::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

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
         (FilterDecoratorWithImage
          ::run,
          getExecutor()));
}
```

It returns a new completable future that will trigger after the decorator has been created

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

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
         (FilterDecoratorWithImage
          ::run,
          getExecutor()));
}
```

Asynchronously filter the image & store it in an output file

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
         (FilterDecoratorWithImage
          ::run,
          getExecutor()));
}
```

thenApplyAsync() runs the actions in a separate thread from the designated thread pool

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

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
        (FilterDecoratorWithImage
         ::run,
         getExecutor()));
}
```

It also returns a new completable future that will trigger when the image has been filtered/stored

Applying Completion Stage Methods in ImageStreamGang

- applyFiltersAsync() internally uses the thenApplyAsync() & thenApply() completion stage methods

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

    .map(filterFuture ->
       filterFuture.thenApplyAsync
         (FilterDecoratorWithImage
          ::run,
          getExecutor()));
}
```

Returns a stream of completable futures to filtered/store images

The flatMap() operation then processes this stream return value

End of Java 8 Completable Futures ImageStreamGang Example (Part 2)