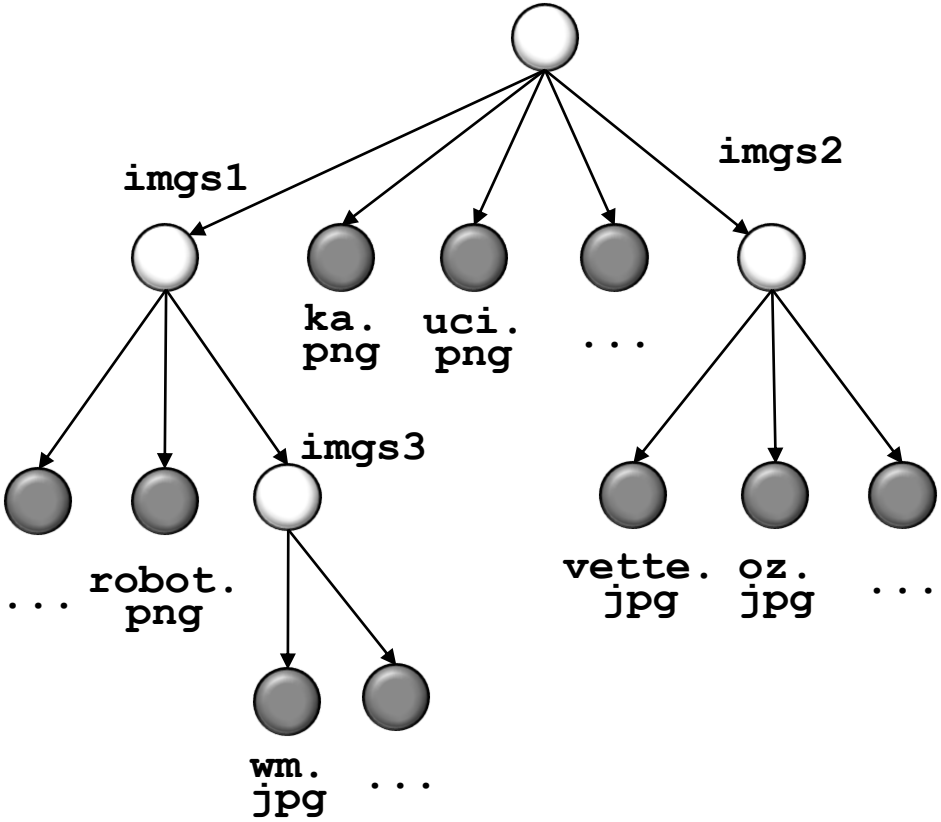
Java Completable Futures Image Crawler Example

(Part 1)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

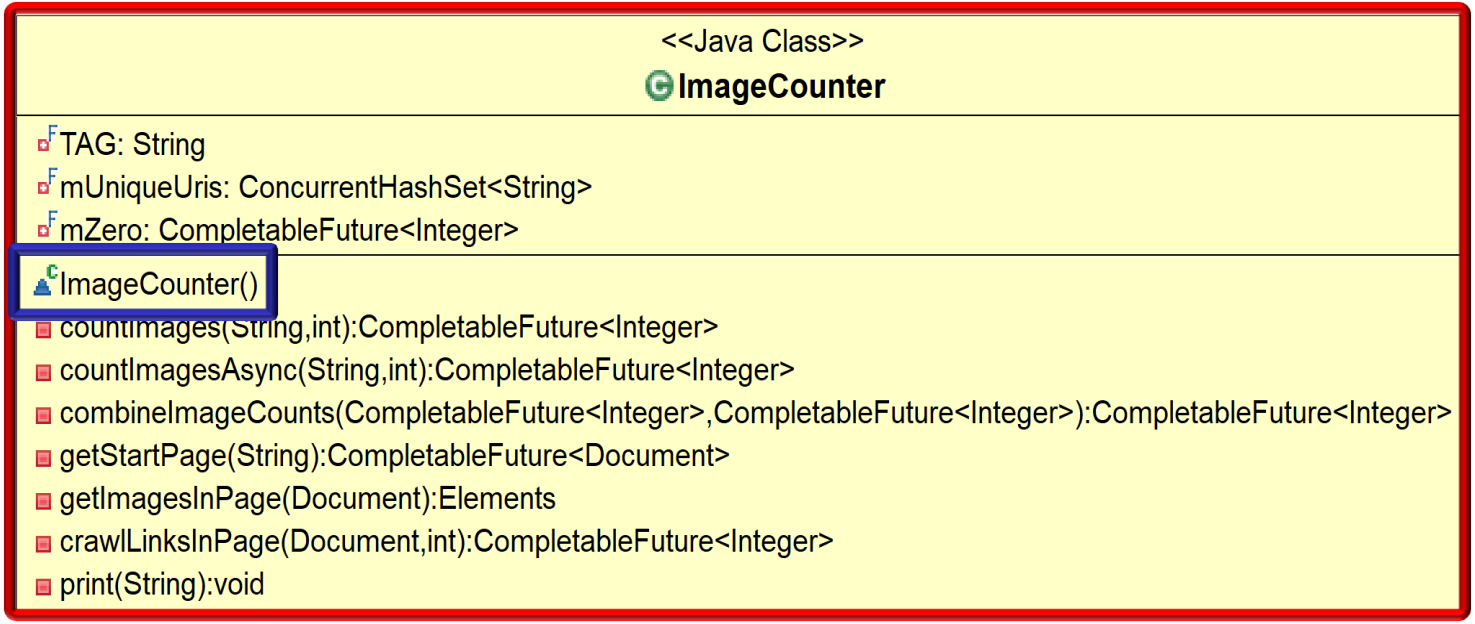
- Know how the Java completable future framework is applied in the image crawler program



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19

Learning Objectives in This Part of the Lesson

- Know how the Java completable future framework is applied in the image crawler program
  - The ImageCounter class and constructor

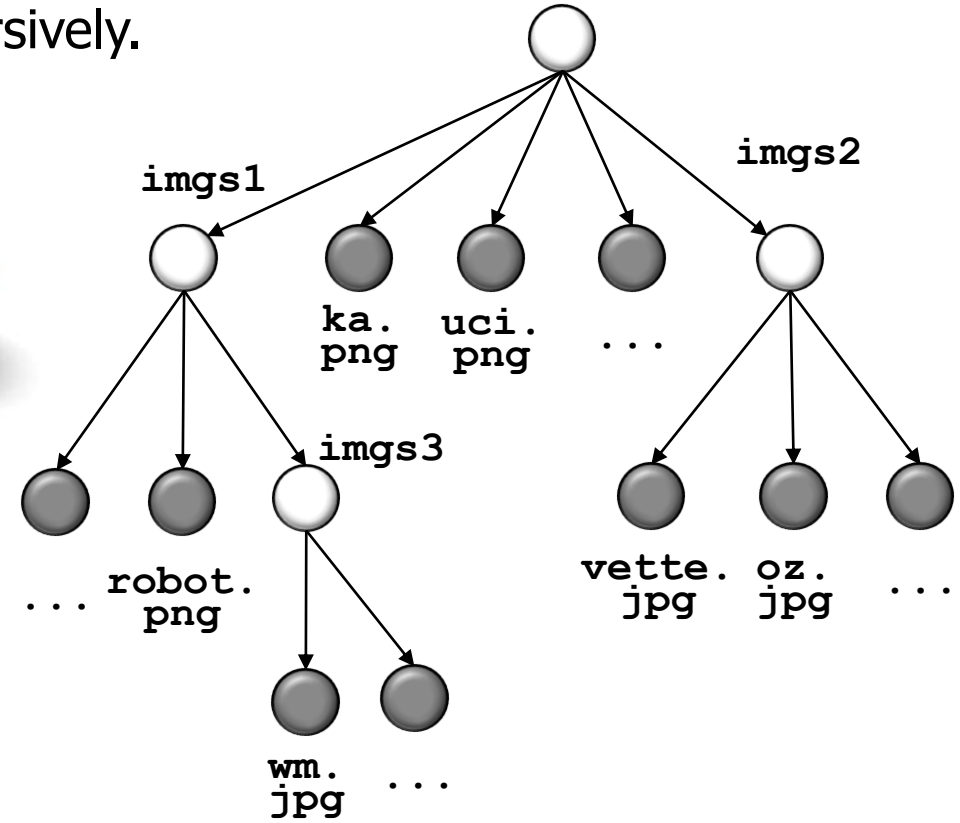


See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19

Applying Completion Stage Methods to Image Crawler

Applying Completion Stage Methods to Image Crawler

- This example shows how many CompletableFuture methods are applied in a program that crawls web pages recursively.

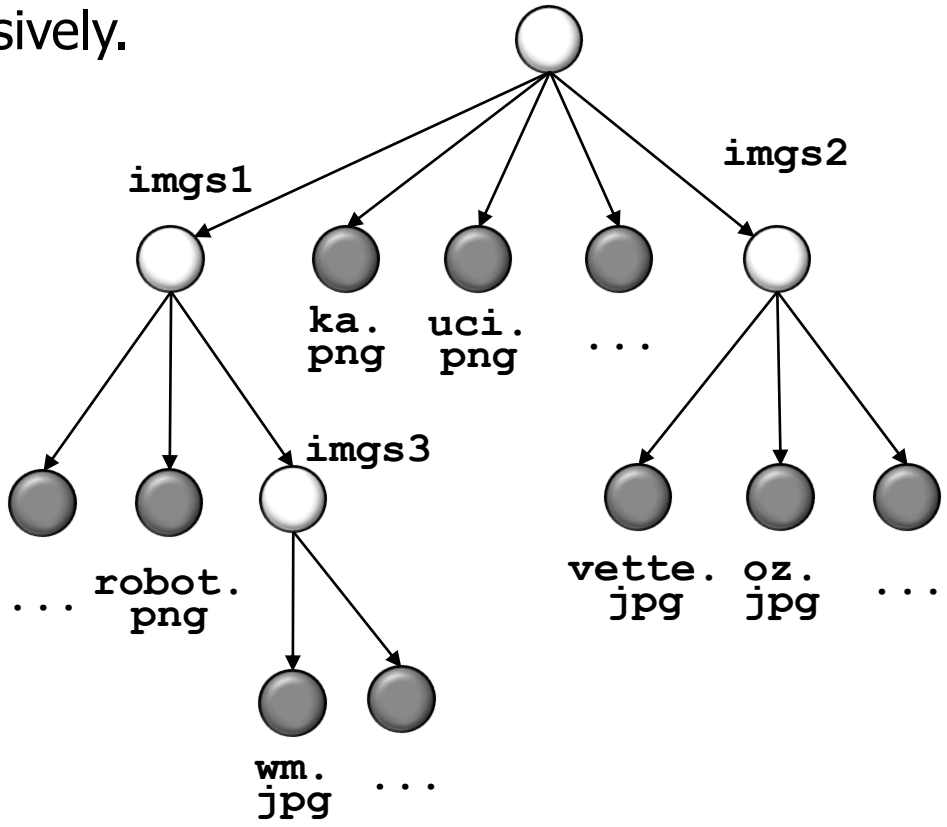


See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19

Applying Completion Stage Methods to Image Crawler

- This example shows how many CompletableFuture methods are applied in a program that crawls web pages recursively.
  - This program counts the number of images on each page

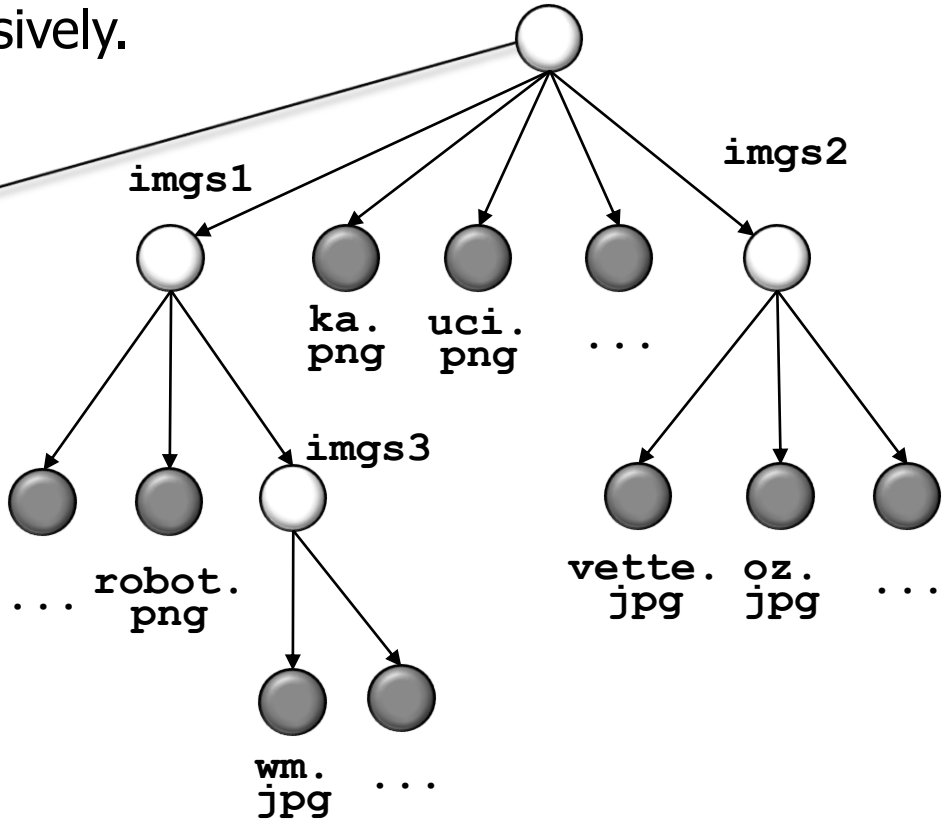
```
ImageCounter[Depth2]: Already
  processed imgs1/index.html
ImageCounter[Depth3]: Exceeded max
  depth of 2
ImageCounter[Depth2]: found 7 images
  for imgs1/index.html in thread 12
ImageCounter[Depth2]: found 7 images
  for imgs2/index.html in thread 13
ImageCounter[Depth1]: found 21 images
  for index.html in thread 13
ImageCounter: 21 total image(s) are
  reachable from index.html
```



Applying Completion Stage Methods to Image Crawler

- This example shows how many CompletableFuture methods are applied in a program that crawls web pages recursively.
  - This program counts the number of images on each page

The root folder can either reside locally (filesystem-based) or be accessed remotely (web-based).



The ImageCounter Class and Constructor

The ImageCounter Class and Constructor

- The ImageCounter class is the heart of this program.

<<Java Class>>

ImageCounter

TAG: String
mUniqueUris: ConcurrentHashSet<String>
mZero: CompletableFuture<Integer>

ImageCounter()
countImages(String,int):CompletableFuture<Integer>
countImagesAsync(String,int):CompletableFuture<Integer>
combineImageCounts(CompletableFuture<Integer>,CompletableFuture<Integer>):CompletableFuture<Integer>
getStartPage(String):CompletableFuture<Document>
getImagesInPage(Document):Elements
crawlLinksInPage(Document,int):CompletableFuture<Integer>
print(String):void

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19

The ImageCounter Class and Constructor

- ImageCounter class and fields

```
class ImageCounter {
```

Counts number of images in a recursively-defined folder structure using many features defined in the completable future framework

```
  private final ConcurrentHashSet<String> mUniqueUris =
    new ConcurrentHashSet<>();


  private final CompletableFuture<Integer> mZero =
    CompletableFuture.completedFuture(0);

  ...
```

The ImageCounter Class and Constructor

- ImageCounter class and fields

```
class ImageCounter {
```

A cache of unique URIs that have already been processed

```
  private final ConcurrentHashSet<String> mUniqueUris =
    new ConcurrentHashSet<>();


  private final CompletableFuture<Integer> mZero =
    CompletableFuture.completedFuture(0);

  ...
```

See Java8/ex19/src/main/java/utils/ConcurrentHashSet.java

The ImageCounter Class and Constructor

- ImageCounter class and fields

```
class ImageCounter {



  private final ConcurrentHashSet<String> mUniqueUris =
    new ConcurrentHashSet<>();


  private final CompletableFuture<Integer> mZero =
    CompletableFuture.completedFuture(0);

  ...
```

Stores a completed future with value of 0

The ImageCounter Class and Constructor

- ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .exceptionally(ex -> 0)

 .thenAccept(total ->
             print(TAG + ": "
                       + total
                       + " total image(s) are reachable from "
                       + rootUri))
  .join();
}
```

The ImageCounter Class and Constructor

- ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .exceptionally(ex -> 0)

 .thenAccept(total ->
             print(TAG + ": "
                       + total
                       + " total image(s) are reachable from "
                       + rootUri))
  .join();
}
```

Start at root Uri of the web page

The ImageCounter Class and Constructor

- ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .exceptionally(ex -> 0)

 .thenAccept(total ->
             print(TAG + ": "
                       + total
                       + " total image(s) are reachable from "
                       + rootUri))
  .join();
}
```

Start count of images at the root URI (with depth of 1) and return future to Integer

See upcoming lesson on "*Java CompletableFuture Image Crawler Example (Part 2)*"

The ImageCounter Class and Constructor

- ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .exceptionally(ex -> 0)

 .thenAccept(total ->
             print(TAG + ": "
                       + total
                       + " total image(s) are reachable from "
                       + rootUri))
  .join();
}
```

Return 0 if an exception occurs

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

The ImageCounter Class and Constructor

- ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .exceptionally(ex -> 0)

 .thenAccept(total ->
             print(TAG + ": "
                       + total
                       + " total image(s) are reachable from "
                       + rootUri))
  .join();
}
```

When future completes print total number of images

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

The ImageCounter Class and Constructor

- ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .exceptionally(ex -> 0)

 .thenAccept(total ->
             print(TAG + ": "
                       + total
                       + " total image(s) are reachable from "
                       + rootUri))
  .join();
}
```

Block until all futures complete

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

The ImageCounter Class and Constructor

- An alternative ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .handle((total, ex) -> {
    if (total == null) total = 0;
    print(TAG + ": " + total
              + " image(s) are reachable from " + rootUri);
    return total;
  })

  .join();
}
```

An alternative means of handling exceptions

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

The ImageCounter Class and Constructor

- An alternative ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .handle((total, ex) -> {
    if (ex != null) total = 0;
    print(TAG + ": " + total
              + " image(s) are reachable from " + rootUri);
    return total;
  })

  .join();
}
```

The exception path

The ImageCounter Class and Constructor

- An alternative ImageCounter constructor implementation

```
public ImageCounter() {
  String rootUri = Options.instance().getRootUri();

  countImages(rootUri, 1)

 .handle((total, ex) -> {
    if (ex != null) total = 0;
    print(TAG + ": " + total
              + " image(s) are reachable from " + rootUri);
    return total;
  })

  .join();
}
```

The non-exception path

The End

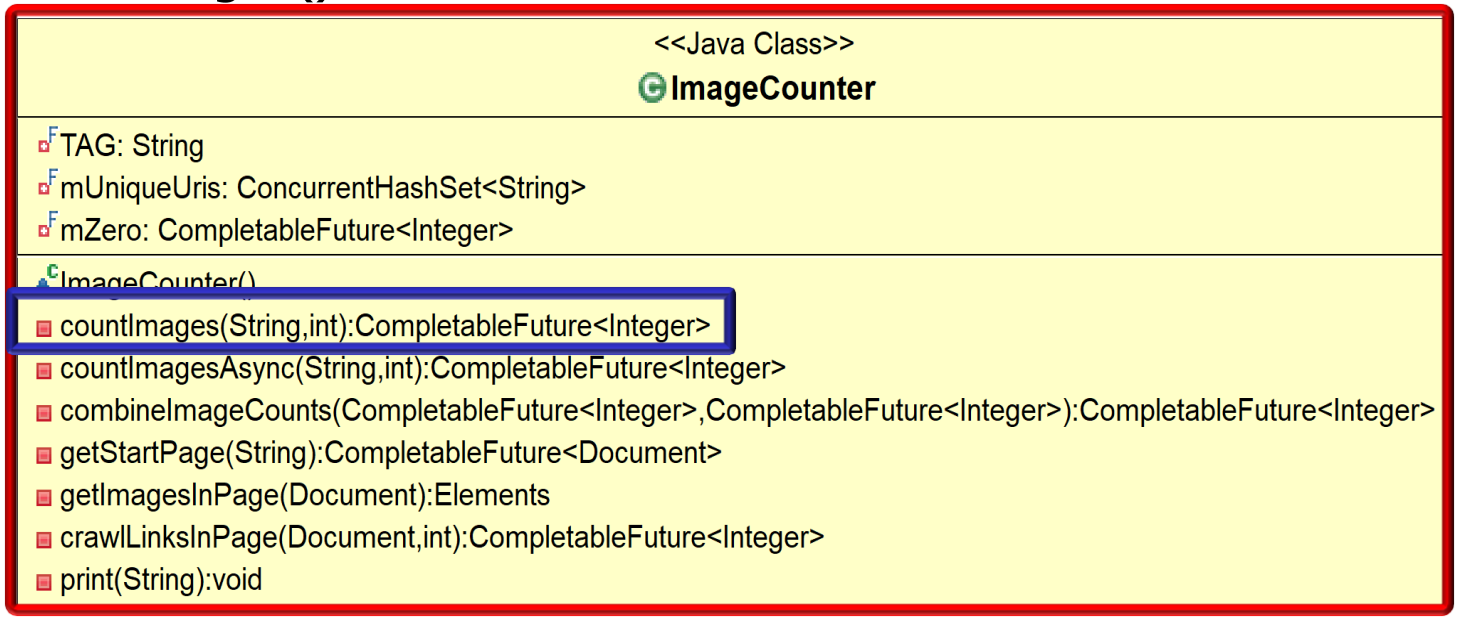
Java Completable Futures Image Crawler Example

(Part 2)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Know how the Java completable future framework is applied in the image crawler program
  - The ImageCounter class and constructor
  - The countImages() method



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19

The countImages() Method

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {
```

Controls the overall flow

```
    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {
```

Return 0 if recursion depth exceeded

```
  if (depth > Options.instance().maxDepth()) return mZero;

  else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

  else return countImagesAsync(pageUri, depth)
    .whenComplete((total, ex) -> {
      if (total != null)
        print(TAG + "[Depth" + depth + "]: found " + total
                  + " images for " + pageUri);
      else print(TAG + ": exception " + ex.getMessage());
    }); ...
```

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

Only process each page once

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

An atomic "check-then-act" method

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

Return 0 if already processed the page

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

Return result of helper method

See upcoming lesson on "*Java CompletableFuture Image Crawler Example (Part 3)*"

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

Handle outcome of stage

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

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

Non-exception path

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

Exception path

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

These values are mutually exclusive.

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {

    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

This method is always called, can't change return value, and doesn't swallow exception.

The countImages() Method

- The main entry point method for counting images asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,
                                       int depth) {
```

Result is a future to number of images at this depth

```
    if (depth > Options.instance().maxDepth()) return mZero;

    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;

    else return countImagesAsync(pageUri, depth)
      .whenComplete((total, ex) -> {
        if (total != null)
          print(TAG + "[Depth" + depth + "]: found " + total
                    + " images for " + pageUri);
        else print(TAG + ": exception " + ex.getMessage());
      }); ...
```

The End

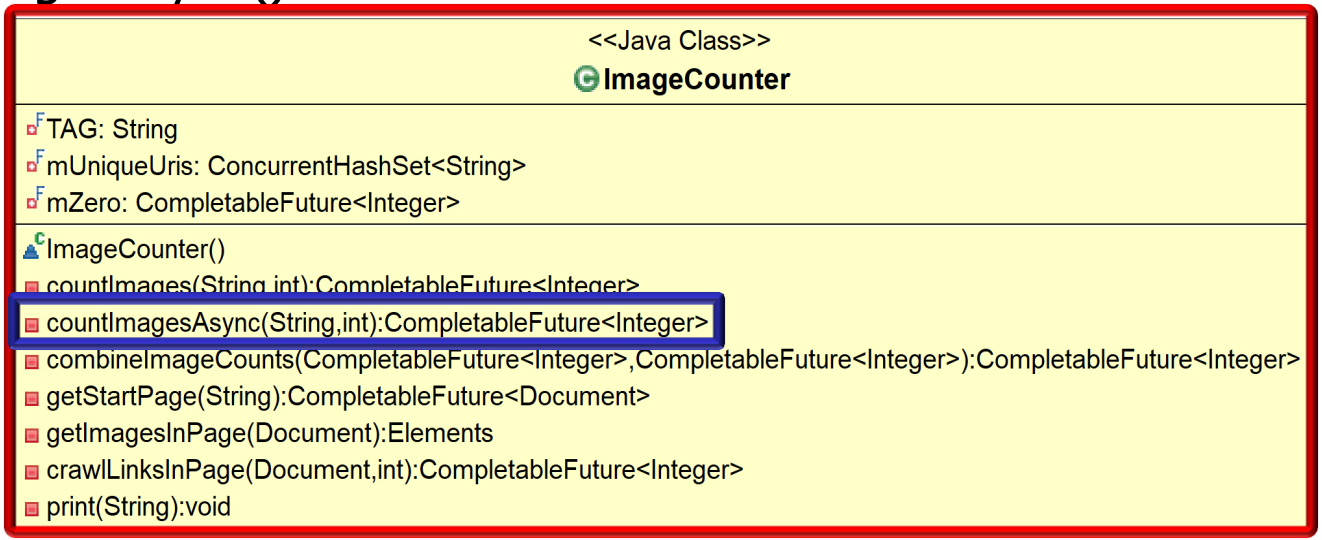
Java Completable Futures Image Crawler Example

(Part 3)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Know how the Java completable futures framework is applied in the image crawler program, e.g.
  - ImageCounter class and constructor
  - countImages() method
  - countImagesAsync() method



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19

The countImagesAsync() Method

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApplyList::size);


  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))


  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
```

The control flow can be modeled via a data flow diagram.

Task1: Get start page asynchronously

Task 2: Count images on the page asynchronously

Task 3: Count images on all hyperlinked pages asynchronously

Task 4: Combine results to create the total asynchronously

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApplyList::size);


  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))


  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

Task 1: Asynchronously get the page at the root URI

The countImagesAsync() Method

- Returns a future to the page at the root URI

```
CompletableFuture<Document> getStartPage(String pageUri) {
        return CompletableFuture
          .supplyAsync(() -> Options
                        .instance()
                        .getJSuper()
                        .getPage(pageUri));
}
```

The countImagesAsync() Method

- Returns a future to the page at the root URI

```
CompletableFuture<Document> getStartPage(String pageUri) {
        return CompletableFuture
          .supplyAsync(() -> Options
                          .instance()
                          .getJSuper()
                          .getPage(pageUri));
}
```

Register an action to run asynchronously

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

The countImagesAsync() Method

- Returns a future to the page at the root URI

```
CompletableFuture<Document> getStartPage(String pageUri) {
        return CompletableFuture
          .supplyAsync(() -> Options
                        .instance()
                        .getJSuper()
                        .getPage(pageUri));
}
```

This code provides a thin wrapper around the jsoup Java HTML parser that works with both web content and local (file) content

See jsoup.org/apidocs/org/jsoup

The countImagesAsync() Method

- Returns a future to the page at the root URI

```
CompletableFuture<Document> getStartPage(String pageUri) {
        return CompletableFuture
          .supplyAsync(() -> Options
                        .instance()
                        .getJSuper()
                        .getPage(pageUri));
}
```

Return a future that completes when the page has finished downloadingrequested

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);
```

pageF completes after the given page finishes downloading

```
  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))


  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);

  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))

  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

Tasks 2 and 3 run concurrently after pageF completes downloading.

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);

  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))


  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

Task 2: Async count number of images on page and return a future to that count

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);


  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))


  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

Sync returns a collection of IMG SRC URLs in this page.

The countImagesAsync() Method

- Synchronously returns a collection of IMG SRC URLs in this page.

```
Elements getImagesInPage(Document page) {
  return page.select("img");
}
```

The countImagesAsync() Method

- Synchronously returns a collection of IMG SRC URLs in this page.

```
Elements getImagesInPage(Document page) {
  return page.select("img");
}
```

This implementation uses the jsoup HTML parsing library, which operates synchronously—hence the need to call this method via thenApplyAsync().

See jsoup.org/apidocs/org/jsoup/select/Selector.html

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);

  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))


  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

Sync returns the number of IMG SRC URLs in this page (very fast).

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);


  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

Task 3: Async count number of images in links on page and return a future to count.

```
  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);


  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

Return future to the number of IMG SRC URLs accessible via links in this page

```
  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

See upcoming lesson on "*Java CompletableFuture Image Crawler Example (Part 4)*"

The countImagesAsync() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);


  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

thenComposeAsync() flattens the future returned by crawlLinksInPage()!

```
  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

The End

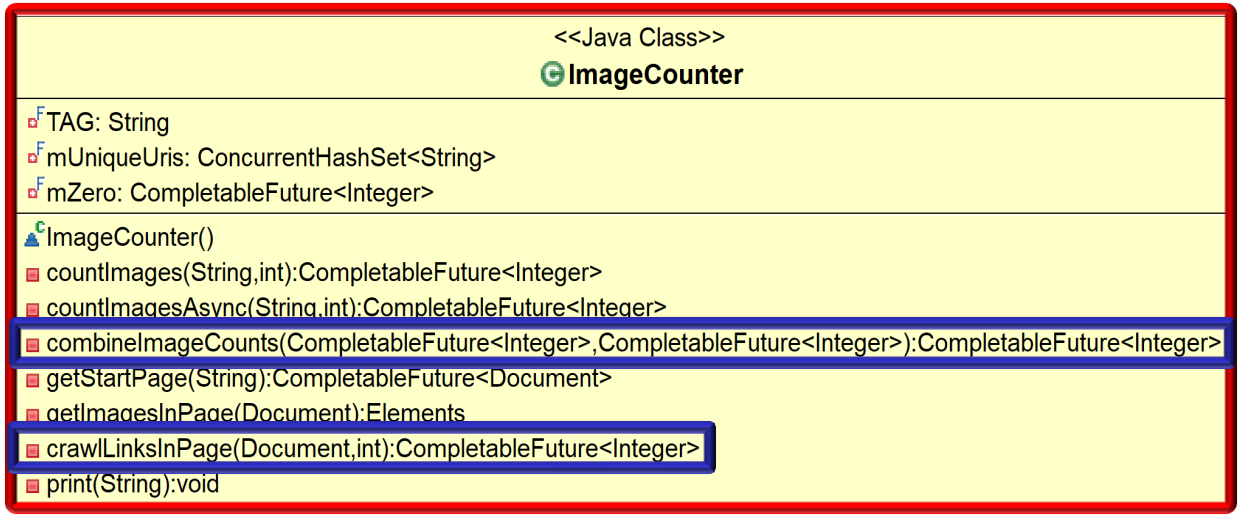
Java Completable Futures Image Crawler Example

(Part 4)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Know how the Java completable futures framework is applied in the image crawler program, e.g.
  - ImageCounter class and constructor
  - countImages() method
  - countImagesAsync() method
  - crawlLinksInPage() and combineImageCounts() methods



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19

The crawlLinksInPage() Method

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                                .getHyperLink(hyperLink), depth+1))



    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

Return future to the number of IMG SRC URLs accessible via links in this page

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                                .getHyperLink(hyperLink), depth+1))



    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

This jsoup call returns a list of all the hyperlinks in this page

See jsoup.org/apidocs/org/jsoup/select/Selector.html

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                                .getHyperLink(hyperLink), depth+1))



    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

This jsoup call operates synchronously, so the thenComposeAsync() method is used

See previous lesson on "*Java CompletableFuture Image Crawler Example (Part 3)*"

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                                .getHyperLink(hyperLink), depth+1))



    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

Convert list of hyperlinks into a sequential stream

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                                .getHyperLink(hyperLink), depth+1))
```

Recursively visit all hyperlinks on this page and increment depth by one

```
    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                     .getHyperLink(hyperLink), depth+1))
```

This method is async and returns a future to number of images at this link/depth.

```
    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

See previous lesson on "*Java Completable Futures Image Crawler Example (Part 2)*"

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                                .getHyperLink(hyperLink), depth+1))

    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

Custom collector converts a stream of futures into a single future that's triggered when all futures in stream complete.

See earlier lesson on "*Advanced Java Completable Future Features: Implementing FuturesCollector*"

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
         countImages(Options.instance().getJSuper()
                             .getHyperLink(hyperLink), depth+1))

    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

After all futures have completed, then create a new future that will trigger after all image counts are summed together.

The crawlLinksInPage() Method

- Return number of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,
                                            int depth) {
  return page
    .select("a[href]").stream()
    .map(hyperLink ->
      countImages(Options.instance().getJSuper()
                                  .getHyperLink(hyperLink), depth+1))

    .collect(FuturesCollector.toFuture())

    .thenApply(list -> list.stream().reduce(0, Integer::sum));
}
```

Returns a future, which is flattened by the thenComposeAsync() caller

The combineImage Counts() Method

The combineImageCounts() Method

- Helper method creates several lambdas variables to simplify implementation.

```
CompletableFuture<Integer> countImagesAsync(String pageUri,
                                            int depth) {
  CompletableFuture<Document> pageF = getStartPage(pageUri);

  CompletableFuture<Integer> imagesInPageF = pageF
    .thenApplyAsync(this::getImagesInPage)
    .thenApply(List::size);


  CompletableFuture<Integer> imagesInLinksF = pageF
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

Task 4: Async count number of images on this page and accessible via links on this page.

```
  return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

See previous lesson on "*Java CompletableFuture Image Crawler Example (Part 3)*"

The combineImageCounts() Method

- Helper method that combines image counts asynchronously

```
CompletableFuture<Integer> combineImageCounts
   (CompletableFuture<Integer> imagesInPageF,
    CompletableFuture<Integer> imagesInLinksF){
   return imagesInPageF
     .thenCombine(imagesInLinksF,
                  Integer::sum);
```

Add the number of images on this page to the number of images on hyperlinks accessible via this page

The combineImageCounts() Method

- Helper method that combines image counts asynchronously

```
CompletableFuture<Integer> combineImageCounts
   (CompletableFuture<Integer> imagesInPageF,
    CompletableFuture<Integer> imagesInLinksF){
  return imagesInPageF
    .thenCombine(imagesInLinksF,
                 Integer::sum);
```

When both futures complete, return a new future to a computation that combines and adds their results.

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

The combineImageCounts() Method

- Helper method that combines image counts asynchronously

```
CompletableFuture<Integer> combineImageCounts
   (CompletableFuture<Integer> imagesInPageF,
    CompletableFuture<Integer> imagesInLinksF){
  return imagesInPageF
    .thenCombineAsync(imagesInLinksF,
                      Integer::sum);
```

It's probably overkill to call thenCombineAsync() here!

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

The End

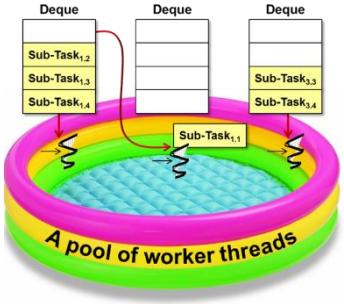
Java Completable Futures ImageStreamGang Example

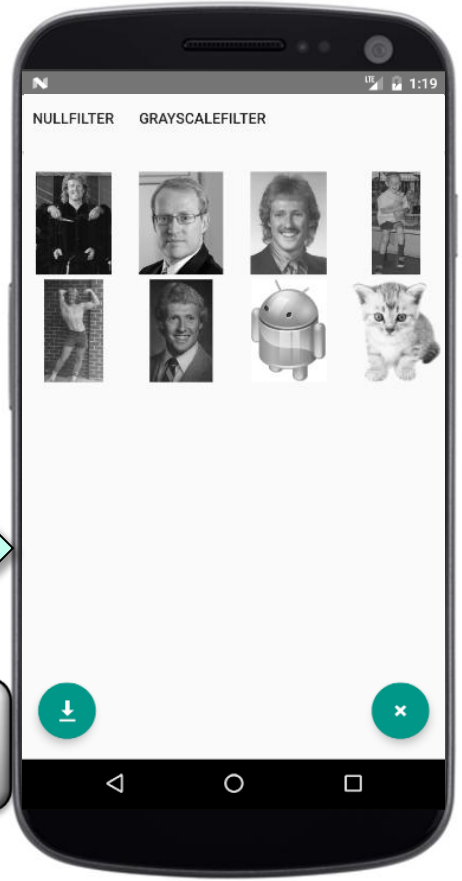
Introduction

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the design of the Java completable future version of the ImageStreamGang app



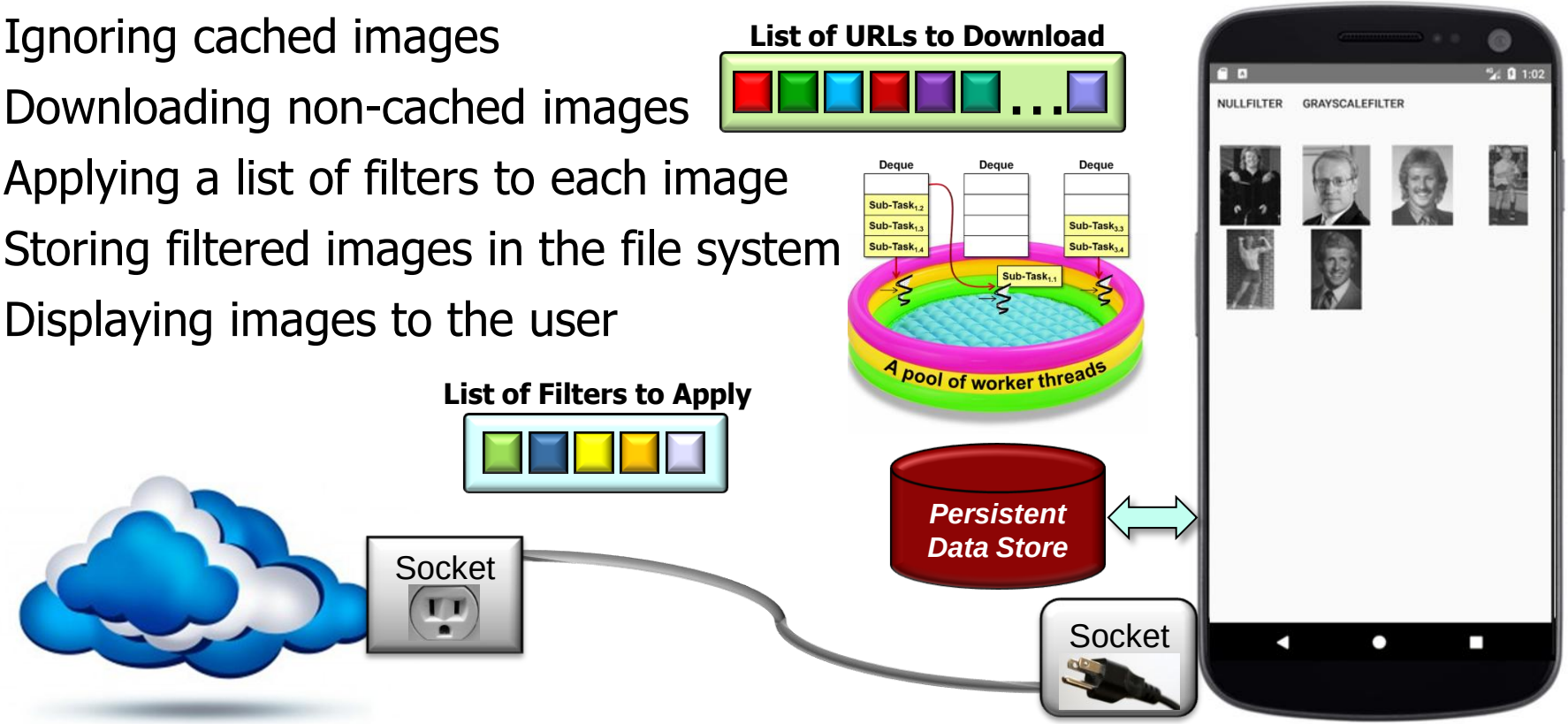


See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Overview of the Completable Futures ImageStreamGang

Overview of Completable Futures ImageStreamGang

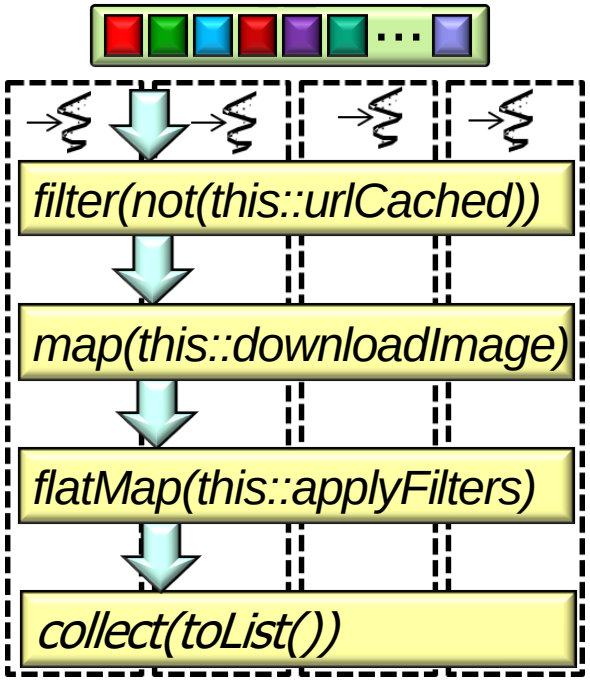
- This app applies several Java parallelism frameworks that do the following
  - Ignoring cached images
  - Downloading non-cached images
  - Applying a list of filters to each image
  - Storing filtered images in the file system
  - Displaying images to the user



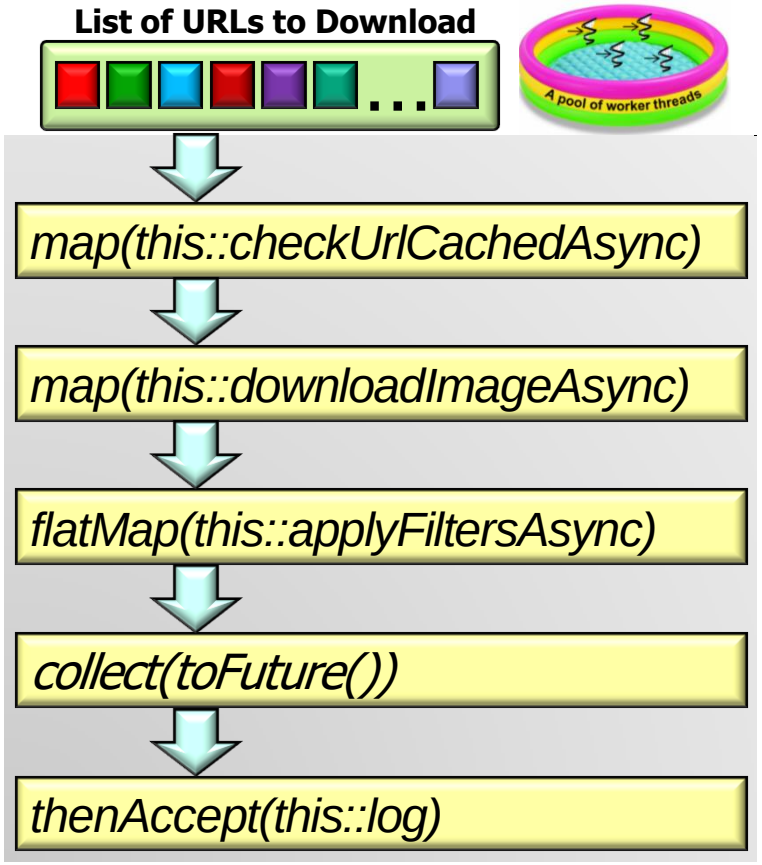
See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.



Parallel Streams

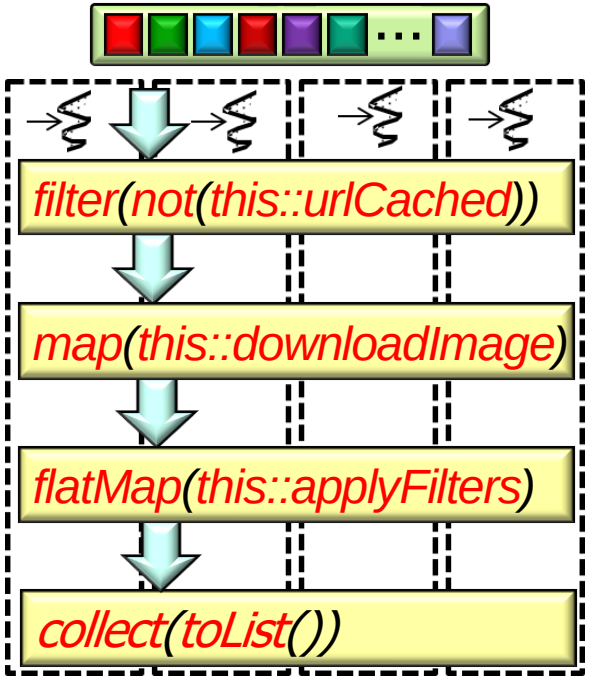


Completable Futures

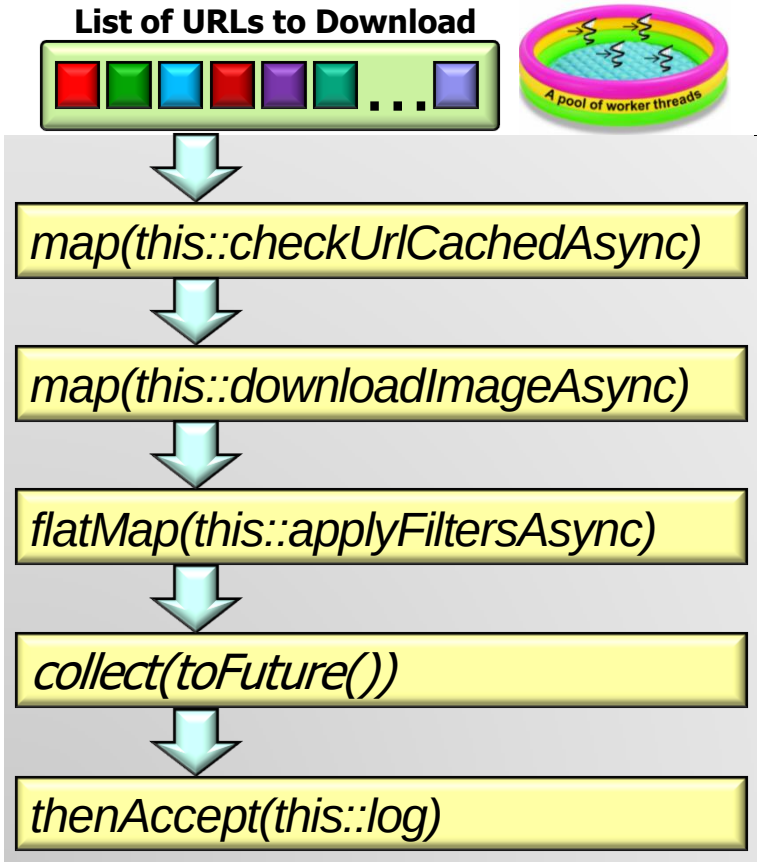
See earlier lesson on "*The Java Parallel ImageStreamGang Example*"

Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.



Parallel Streams

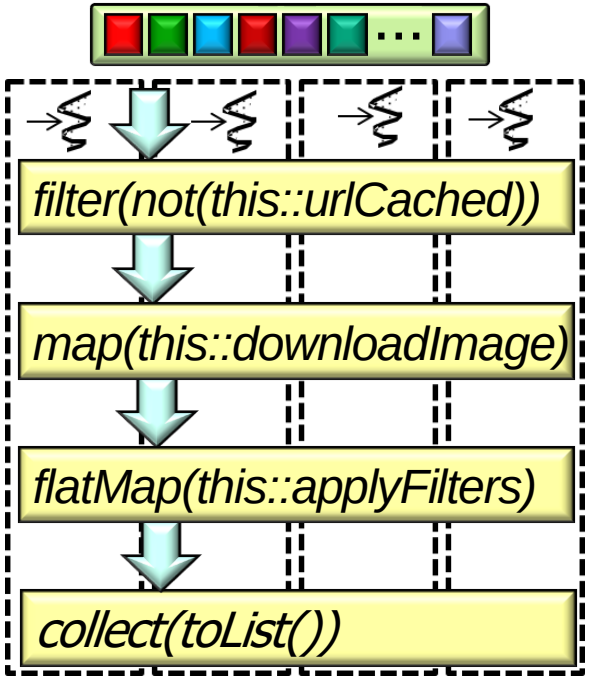


Completable Futures

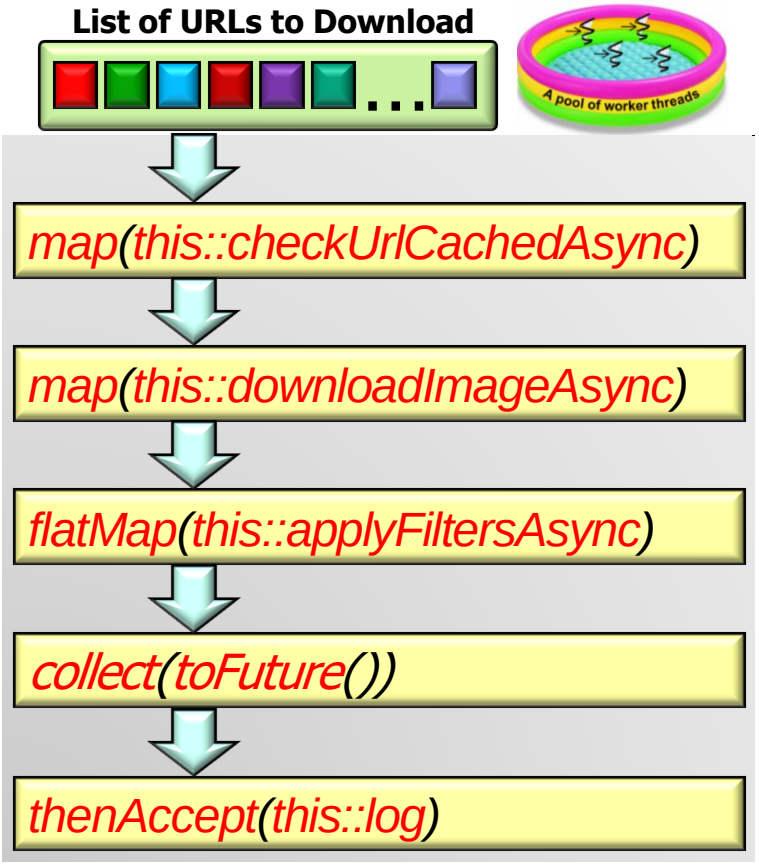
All behaviors in the parallel stream variant are synchronous.

Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.



Parallel Streams

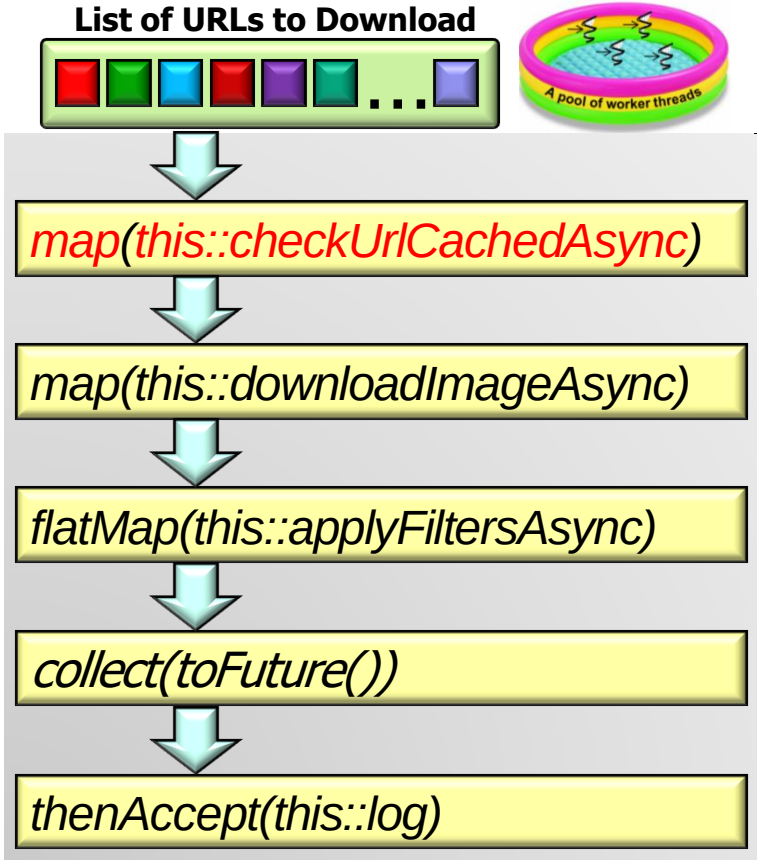


Completable Futures

All behaviors in the completable futures variant are asynchronous.

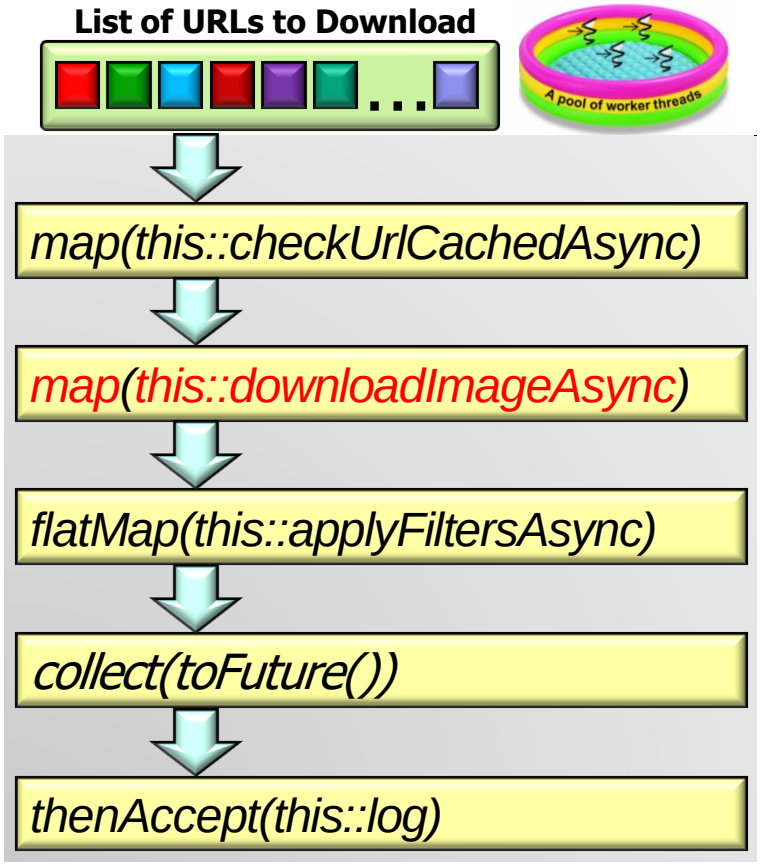
Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.
  - Ignore cached images *asynchronously*



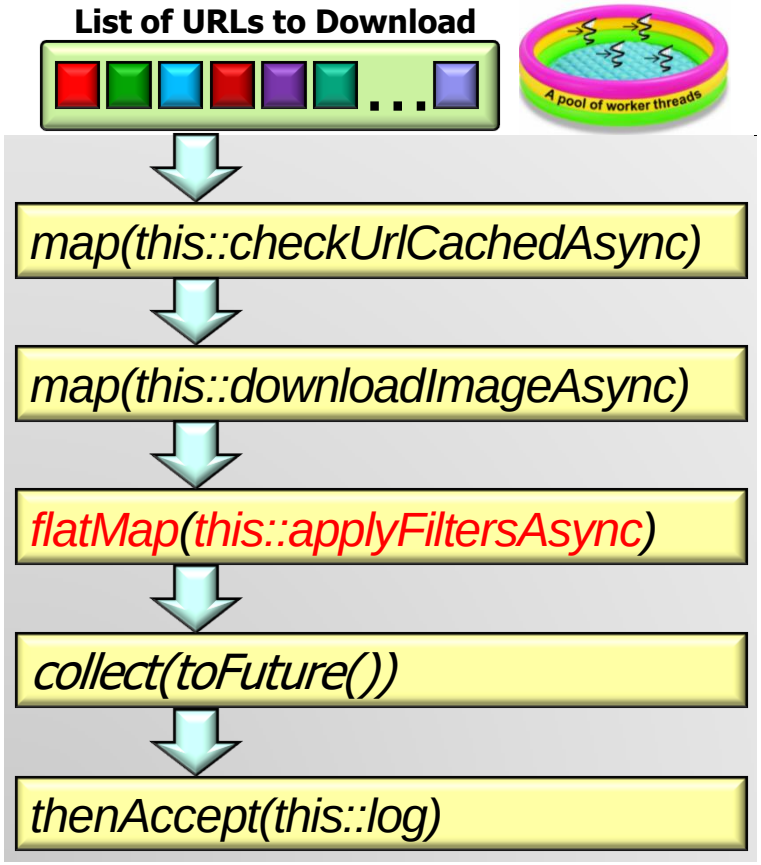
Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.
  - Ignore cached images *asynchronously*
  - Download non-cached images *asynchronously*



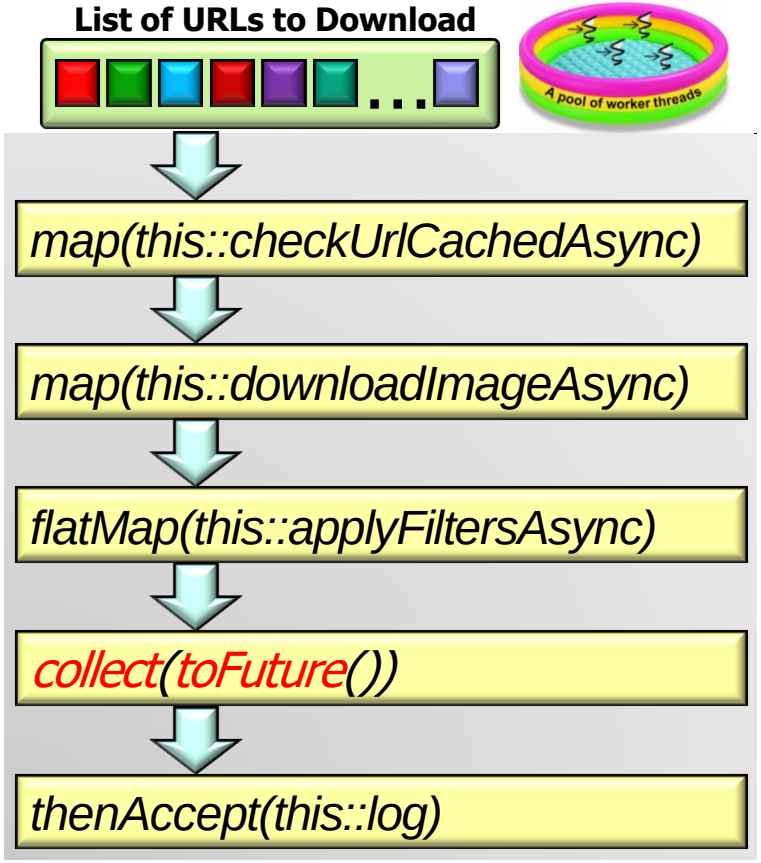
Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.
  - Ignore cached images *asynchronously*
  - Download non-cached images *asynchronously*
  - As downloads complete apply a list of filters and store filtered images in file system *asynchronously*



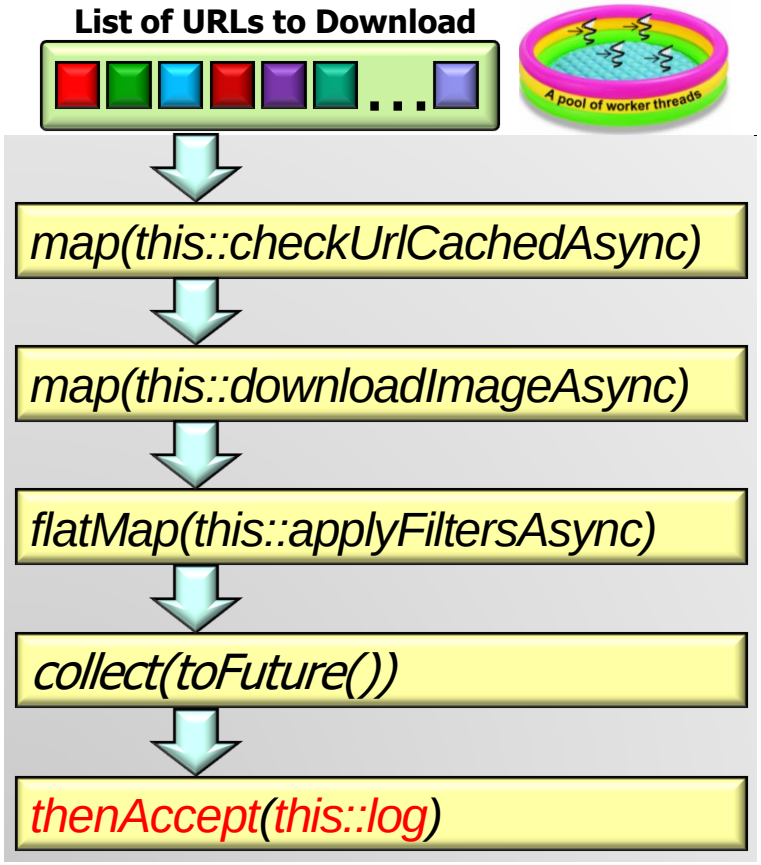
Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.
  - Ignore cached images *asynchronously*
  - Download non-cached images *asynchronously*
  - As downloads complete apply a list of filters and store filtered images in file system *asynchronously*
  - Trigger all the stream processing to run *asynchronously*



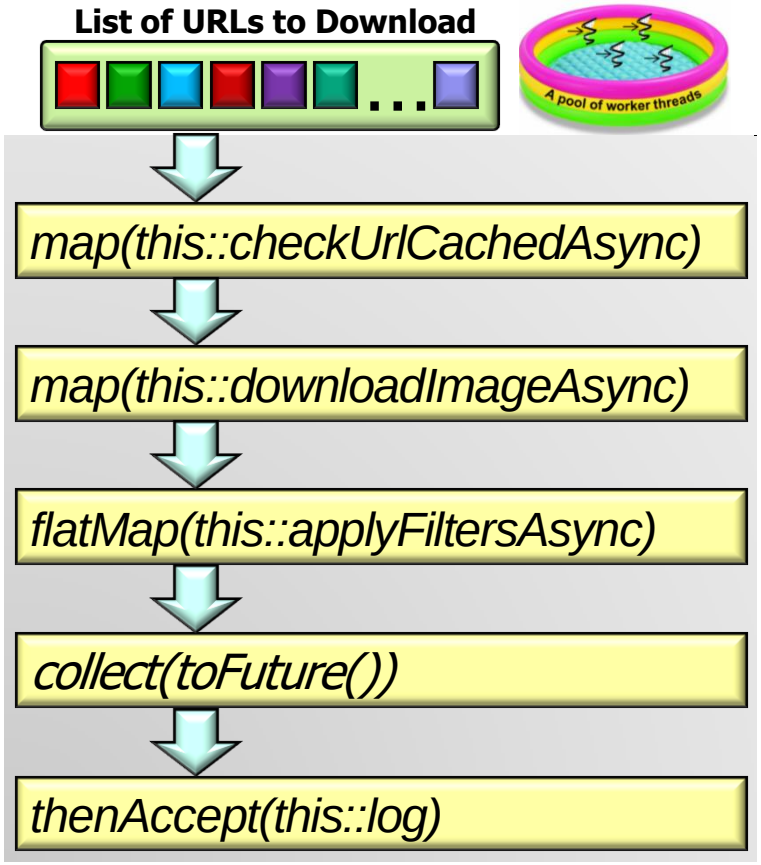
Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.
  - Ignore cached images *asynchronously*
  - Download non-cached images *asynchronously*
  - As downloads complete apply a list of filters and store filtered images in file system *asynchronously*
  - Trigger all the stream processing to run *asynchronously*
  - Get results of asynchronous computations



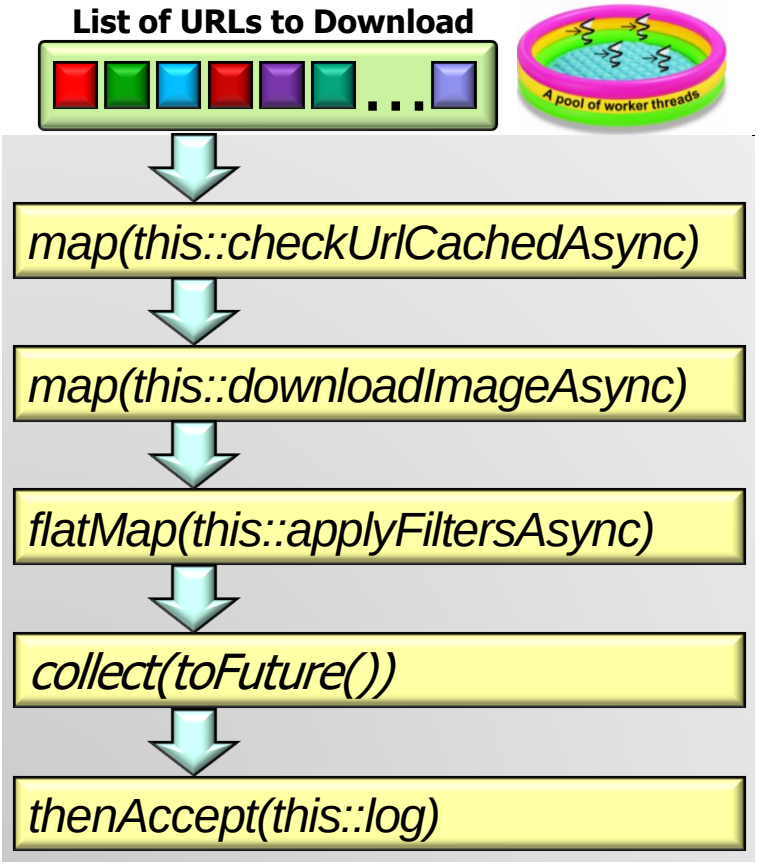
Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the earlier parallel streams variant.
  - Ignore cached images *asynchronously*
  - Download non-cached images *asynchronously*
  - As downloads complete apply a list of filters and store filtered images in file system *asynchronously*
  - Trigger all the stream processing to run *asynchronously*
  - Get results of asynchronous computations
    - Ultimately display images to user



Overview of Completable Futures ImageStreamGang

- Combining completable futures and streams helps to *efficiently* close the gap between the domain intent and the implementation.



The End

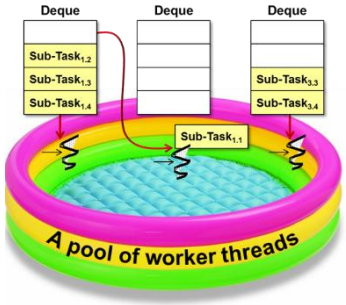
Java Completable Futures ImageStreamGang Example

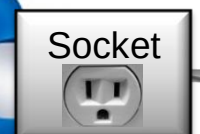
Applying Completable Futures

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the design of the Java completable future version of ImageStreamGang
- Know how to apply completable futures to ImageStreamGang





Applying Completable Futures to ImageStreamGang

Applying Completable Futures to ImageStreamGang

- Focus on processStream()

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

See imagestreamgang/streams/ImageStreamCompletableFuture1.java

Applying Completable Futures to ImageStreamGang

- Focus on processStream()

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                log(stream.flatMap
                    (Optional::stream),
                     urls.size()))
    .join();
```

Get the list of URLs input by the user

Applying Completable Futures to ImageStreamGang

- Focus on processStream()

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Combines a Java sequential stream with completable futures

Applying Completable Futures to ImageStreamGang

- Focus on processStream()

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Factory method creates a stream of URLs

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Asynchronously check if images have already been cached locally

map() converts a stream of URLs to a stream of futures to optional URLs.

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Asynchronously download an image at each given URL

map() converts URL futures (completed) to image futures (downloading).

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Asynchronously filter and store downloaded images on the local file system

flatMap() converts image futures (completed) to filtered image futures (xforming/storing).

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Trigger all intermediate operations and create a future used to wait for all async operations associated w/the stream of futures to complete

See lesson on "*Java CompletableFutures ImageStreamGang Example: StreamOfFuturesCollector*"

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

This lambda logs the results when all the futures in stream complete their async processing.

Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Block until all images have been downloaded, processed, and stored

This join() is the one and only call in this implementation strategy.

The End

Java Completable Futures ImageStreamGang Example

Applying Factory Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the design of the Java completable future version of ImageStreamGang
- Know how to apply completable futures to ImageStreamGang
  - Factory methods
    - supplyAsync()

<<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>

Applying Factory Methods in ImageStreamGang

Applying Factory Methods in ImageStreamGang

- Initiate an async check to see if images are cached locally

map() calls the behavior checkUrlCachedAsync()

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                log(stream.flatMap
                    (Optional::stream),
                     urls.size()))
    .join();
```

Applying Factory Methods in ImageStreamGang

- Initiate an async check to see if images are cached locally

Asynchronously check if a URL is already downloaded

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Applying Factory Methods in ImageStreamGang

- Initiate an async check to see if images are cached locally

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Returns a stream of completable futures to optional URLs, which have a value if the URL is not cached, or are empty if it is cached

Later behaviors simply ignore "empty" optional URL values.

Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally.

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync(URL url) {
  return CompletableFuture
    .supplyAsync(() ->
                 Optional.ofNullable(urlCached(url)
                   ? null
                   : url),
                 getExecutor());
}
```

See imagestreamgang/streams/ImageStreamCompletableFutureBase.java

Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally.

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync(URL url) {
  return CompletableFuture
    .supplyAsync(() ->
                 Optional.ofNullable(urlCached(url)
                   ? null
                   : url),
                 getExecutor());
}
```

This factory method registers an action that runs asynchronously.

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

Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally.

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync(URL url) {
  return CompletableFuture
    .supplyAsync(() ->
                 Optional.ofNullable(urlCached(url)
                   ? null
                   : url),
                 getExecutor());
}
```

supplyAsync() runs action in a worker thread from the common fork-join pool.

```
void initiateStream() {
  // Set the executor to the common fork-join pool.
  setExecutor(ForkJoinPool.commonPool());
  ...
}
```

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

Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally.

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync(URL url) {
  return CompletableFuture
    .supplyAsync(() ->
                 Optional.ofNullable(urlCached(url)
                   ? null
                   : url),
                 getExecutor());
}
```

ofNullable() is a factory method that returns an optional URL, which has a value if the URL is not cached, or is empty if it is already cached.

See docs.oracle.com/javase/8/docs/api/java/util/Optional.html#ofNullable

Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally.

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync(URL url) {
  return CompletableFuture
    .supplyAsync(() ->
                 Optional.ofNullable(urlCached(url)
                   ? null
                   : url),
                 getExecutor());
}
```

Returns true if the image has already been filtered before

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

See imagestreamgang/streams/ImageStreamGang.java

Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally.

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync(URL url) {
  return CompletableFuture
    .supplyAsync(() ->
                 Optional.ofNullable(urlCached(url)
                   ? null
                   : url),
                 getExecutor());
}
```

Returns true if image file already exists

```
boolean urlCached(URL url, String filterName) {
  File file = new File(getPath(), filterName);
  File imageFile = new File(file, getNameForUrl(url));
  return !imageFile.createNewFile();
}
```

See imagestreamgang/streams/ImageStreamGang.java

Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally.

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync(URL url) {
  return CompletableFuture
    .supplyAsync(() ->
                 Optional.ofNullable(urlCached(url)
                   ? null
                   : url),
                 getExecutor());
}
```

ClearlyBetter® SOLUTIONS

```
boolean urlCached(URL , String filterName) {
  File = new File(getPath(), filterName);
  File imageFile = new File(file, getNameForUrl(url));
  return !imageFile.createNewFile();
}
```

There are clearly better ways of implementing an image cache!

Java Completable Futures ImageStreamGang Example: Applying Factory Methods

The End

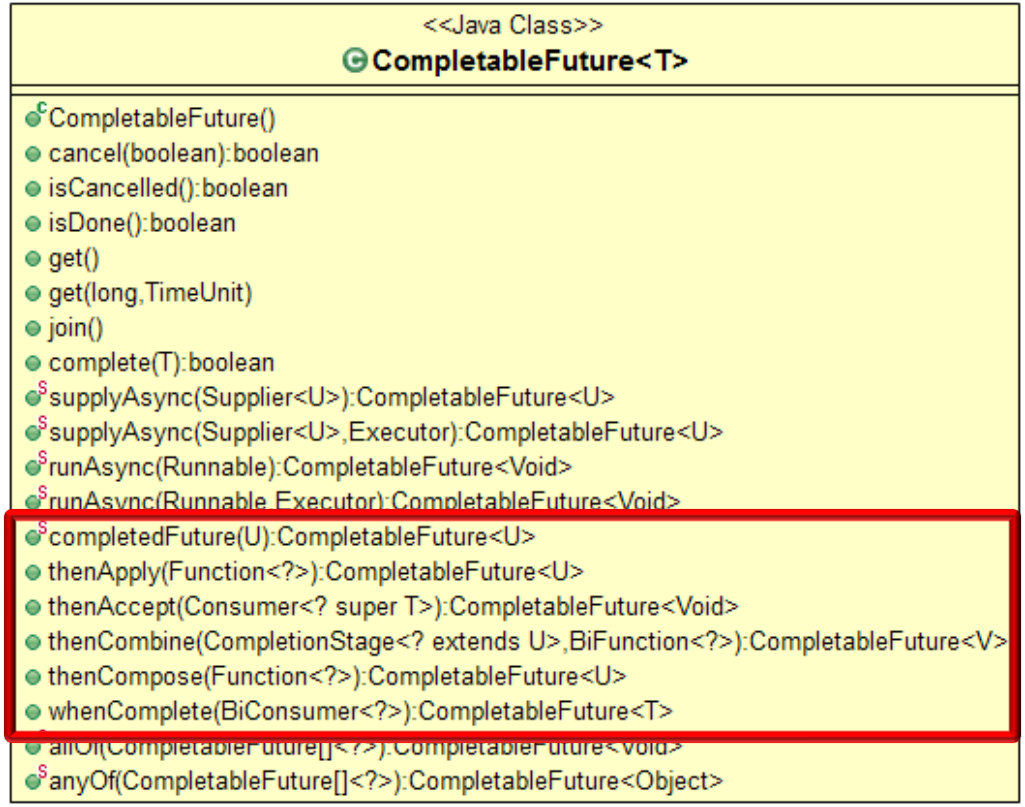
Java Completable Futures ImageStreamGang Example

Applying Completion Stage Methods (Part 1)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the design of the Java completable future version of ImageStreamGang
- Know how to apply completable futures to ImageStreamGang
  - Factory methods
  - Completion stage methods
    - downloadImageAsync()



Applying Completion Stage Methods in DownloadImageAsync

Applying Completion Stage Methods in DownloadImageAsync

- Asynchronously download an image at each given URL

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

map() calls the behavior downloadImageAsync()

Applying Completion Stage Methods in DownloadImageAsync

- Asynchronously download an image at each given URL

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Asynchronously downloads an image and stores it in memory.

Later behaviors simply ignore "empty" optional images.

Applying Completion Stage Methods in DownloadImageAsync

- Asynchronously download an image at each given URL

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Returns a stream of futures to optional images, which have a value if the image is being downloaded, or are empty if it is already cached

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
CompletableFuture<Optional<Image>> downloadImageAsync
                    (CompletableFuture<Optional<URL>> urlFuture) {
  return urlFuture
    .thenApplyAsync(urlOpt ->
                    urlOpt
                    .map(this::blockingDownload),
                    getExecutor());
}
```

Asynchronously download image when future completes

See imagestreamgang/streams/ImageStreamCompletableFuture1.java

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
CompletableFuture<Optional<Image>> downloadImageAsync
                    (CompletableFuture<Optional<URL>> urlFuture) {
  return urlFuture
    .thenApplyAsync(urlOpt ->
                      urlOpt
                      .map(this::blockingDownload),
                      getExecutor());
}
```

This completion stage method registers an action that's not executed immediately, but only after urlFuture completes.

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

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
CompletableFuture<Optional<Image>> downloadImageAsync
                    (CompletableFuture<Optional<URL>> urlFuture) {
  return urlFuture
    .thenApplyAsync(urlOpt ->
                      urlOpt
                      .map(this::blockingDownload),
                      getExecutor());
}
```

If a URL is present, then download it when urlFuture completes, and return an optional describing the result; otherwise return an empty optional.

See docs.oracle.com/javase/8/docs/api/java/util/Optional.html#map

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
CompletableFuture<Optional<Image>> downloadImageAsync
                    (CompletableFuture<Optional<URL>> urlFuture) {
  return urlFuture
    .thenApplyAsync(urlOpt ->
                    urlOpt
                    .map(this::blockingDownload),
                    getExecutor());
}
```

Asynchronously run blockingDownload() if urlOpt is non-empty

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
CompletableFuture<Optional<Image>> downloadImageAsync
                    (CompletableFuture<Optional<URL>> urlFuture) {
  return urlFuture
    .thenApplyAsync(urlOpt ->
                      urlOpt
                      .map(this::blockingDownload),
                      getExecutor());
}
```

Use the common fork-join pool and its ManagedBlocker mechanism

See earlier lesson on "*The Java Fork-Join Pool: The ManagedBlocker Interface*"

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
Image blockingDownload(URL url) {
        return BlockingTask
          .callInManagedBlock(() ->
                                  downloadImage(url));
}
```

Transform a URL into an Image by downloading each image via the URL

See imagestreamgang/streams/ImageStreamGang.java

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
Image blockingDownload(URL url) {
        return BlockingTask
          .callInManagedBlock(() ->
                                  downloadImage(url));
}
```

BlockingTask.callInManagedBlock() wraps the ManagedBlocker interface, which auto-expands the common fork-join pool to handle the blocking image download.

See imagestreamgang/utils/BlockingTask.java

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
Image blockingDownload(URL url) {
        return BlockingTask
          .callInManagedBlock(() ->
                                  downloadImage(url));
}
```

This blocking call actually downloads the image at the given URL.

```
Image downloadImage(URL url) {
   return new Image(url,
                    NetUtils.downloadContent(url));
}
```

See imagestreamgang/streams/ImageStreamGang.java

Applying Completion Stage Methods in DownloadImageAsync

- downloadImageAsync() uses the thenApplyAsync() method internally.

```
CompletableFuture<Optional<Image>> downloadImageAsync
                    (CompletableFuture<Optional<URL>> urlFuture) {
  return urlFuture
    .thenApplyAsync(urlOpt ->
                      urlOpt
                      .map(this::blockingDownload),
                      getExecutor());
}
```

Returns a future to an optional image that will complete when the image finishes downloading

Java Completable Futures ImageStreamGang Example:
Applying Completion Stage Methods in Image StreamGang (Part 1)

The End

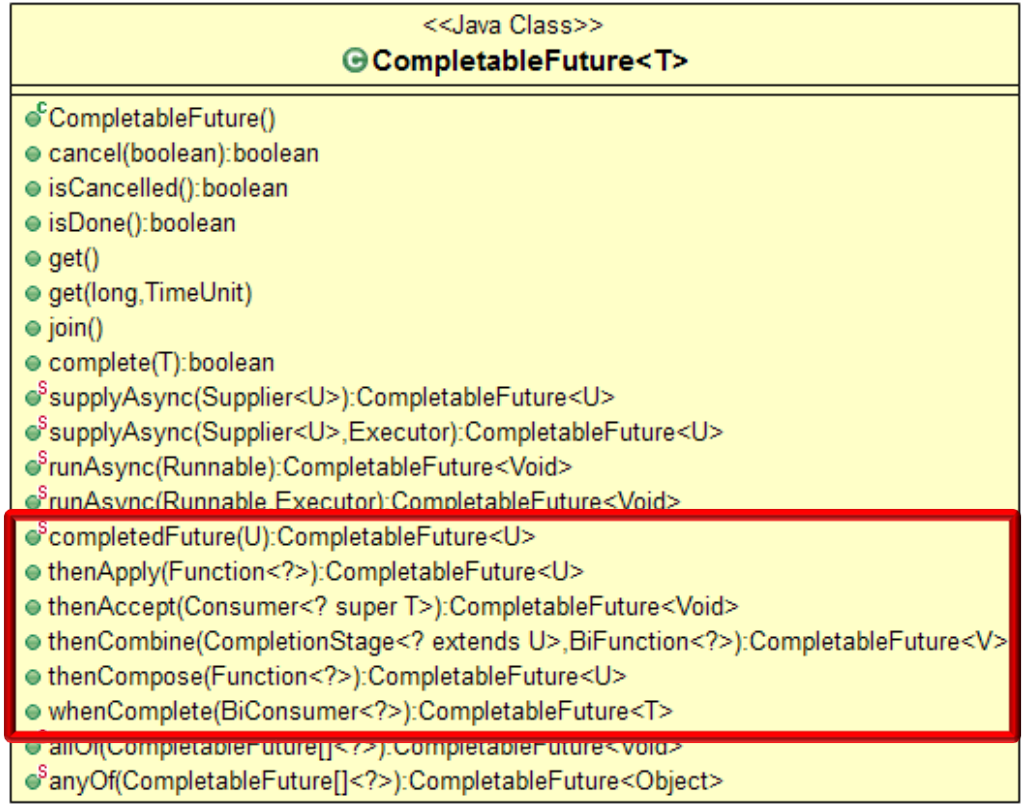
Java Completable Futures ImageStreamGang Example

Applying Completion Stage Methods (Part 2)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the design of the Java completable future version of ImageStreamGang
- Know how to apply completable futures to ImageStreamGang
  - Factory methods
  - Completion stage methods
    - downloadImageAsync()
    - applyFiltersAsync()



Applying Completion Stage Methods in ApplyFiltersAsync

Applying Completion Stage Methods in ApplyFiltersAsync

- Asynchronously filter and store downloaded images on the local file system

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

flatMap() calls behavior applyFiltersAsync()

Applying Completion Stage Methods in ApplyFiltersAsync

- Asynchronously filter and store downloaded images on the local file system

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Asynchronously filter images and store them into files1

Later operations ignore "empty" optional images.

Applying Completion Stage Methods in ApplyFiltersAsync

- Asynchronously filter and store downloaded images on the local file system

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

"Flatten" all filtered/stored images into a single output stream

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#flatMap

Applying Completion Stage Methods in ApplyFiltersAsync

- Asynchronously filter and store downloaded images on the local file system

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Returns a stream of futures to optional images, which have a value if the image is being filtered, or are empty if it is already cached

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));
}
```

Asynchronously filter images and then store them into files

See imagestreamgang/streams/ImageStreamCompletableFuture1.java

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));
}
```

Convert the list of filters into a stream

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));
}
```

Asynchronously apply a filter action after image completes downloading

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

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));
}
```

This completion stage method registers an action that's not executed immediately, but runs only after imageFuture completes.

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));}
```

If an image is present, then perform the action and return optional containing result; otherwise return an empty optional.

See docs.oracle.com/javase/8/docs/api/java/util/Optional.html#map

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));}
```

If image is non-empty, then asynchronously filter the image and store it in an output file.

See en.wikipedia.org/wiki/Decorator_pattern

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                 (filter, image).run()),
                         getExecutor()));}
```

thenApplyAsync() runs action in a worker thread from the common fork-join pool.

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                 (filter, image).run()),
                         getExecutor()));
}
```

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

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));
}
```

applyFiltersAsync() returns a stream of completable futures to optional images.

Applying Completion Stage Methods in ApplyFiltersAsync

- applyFiltersAsync() uses the thenApplyAsync() method internally.

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync
                (CompletableFuture<Optional<Image>> imageFuture) {
  return mFilters
    .stream()

    .map(filter -> imageFuture
         .thenApplyAsync(imageOpt ->
                         imageOpt
                         .map(image ->
                              makeFilterDecoratorWithImage
                                (filter, image).run()),
                         getExecutor()));
}
```

There is no terminal operation here, so a stream is returned!

Applying Completion Stage Methods in ApplyFiltersAsync

- Asynchronously filter and store downloaded images on the local file system

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

flatMap() merges the streams of futures returned by applyFilters Async() into just one stream.

This stream is processed by collect(), as discussed in the next part of the lesson.

Java Completable Futures ImageStreamGang Example:
Applying Completion Stage Methods in Image StreamGang (Part 2)

The End

Java Completable Futures ImageStreamGang Example

Applying Arbitrary-Arity Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the design of the Java completable future version of ImageStreamGang
- Know how to apply completable futures to ImageStreamGang
  - Factory methods
  - Completion stage methods
  - 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>
```

Applying Arbitrary-Arity Methods in ImageStreamGang

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

flatMap() outputs a stream of futures associated with processing that's running asynchronously.

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Provides a single means to await completion of a list of futures, before continuing with the program

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

collect() also triggers processing of all the intermediate operations.

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.
  - StreamOfFuturesCollector wraps "arbitrary-arity" allOf() method.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Return a future that completes when all futures in the stream complete

See AndroidGUI/app/src/main/java/livelessons/utils/StreamOfFuturesCollector.java

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.
  - StreamOfFuturesCollector wraps "arbitrary-arity" allOf() method.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Log the results after the final future completes

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.
  - StreamOfFuturesCollector wraps "arbitrary-arity" allOf() method.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                 (Optional::stream),
                  urls.size()))
    .join();
```

Remove empty optional values from the stream in Java 9+

See docs.oracle.com/javase/9/docs/api/java/util/Optional.html#flatMap

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.
  - StreamOfFuturesCollector wraps "arbitrary-arity" allOf() method.

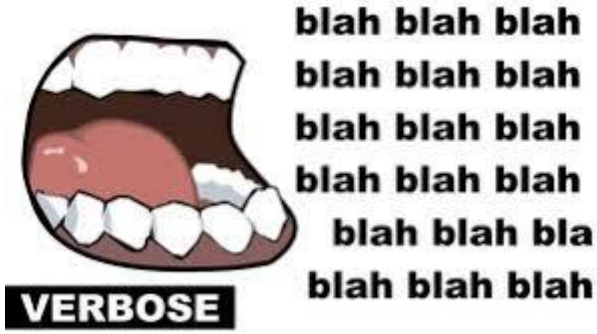
```
void processStream() {
  List<URL> urls = getInput();

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream -> log(stream
        .filter(Optional::isPresent)
        .map(Optional::get),
                urls.size()))
    .join();
```

Remove empty optional values from the stream in Java 8

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.
  - StreamOfFuturesCollector wraps "arbitrary-arity" allOf() method.

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

CompletableFuture<Stream<Image>>
  resultsFuture = urls
  .stream()
  .map(this::checkUrlCachedAsync)
  .map(this::downloadImageAsync)
  .flatMap(this::applyFiltersAsync)
  .collect(toFuture())
  .thenApply(stream -> log(stream
      .filter(Optional::isPresent)
      .map(Optional::get),
                urls.size()))
  .join();
```



Java 8 is more verbose..

See blog.codefx.org/java/java-9-optional

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.
  - StreamOfFuturesCollector wraps "arbitrary-arity" allOf() method.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Wait until all the async processing is completed

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

Applying Arbitrary-Arity Methods in ImageStreamGang

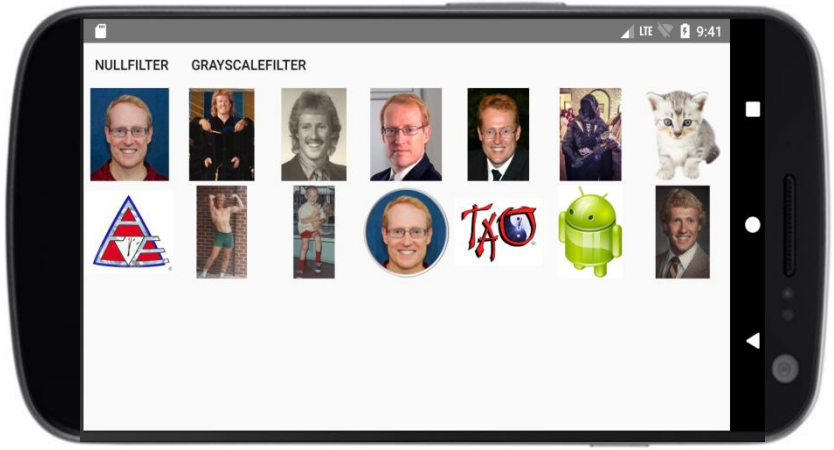
- collect() returns a future to a stream of futures to images being processed asynchronously.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

This is the one and only call to join() in this async stream pipeline!

Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a future to a stream of futures to images being processed asynchronously.




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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Images are displayed after all the async processing completes.

Java Completable Futures ImageStreamGang Example:
Applying Arbitrary-Arity Methods in Image StreamGang

The End

Java Completable Futures ImageStreamGang Example

StreamOfFuturesCollector

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

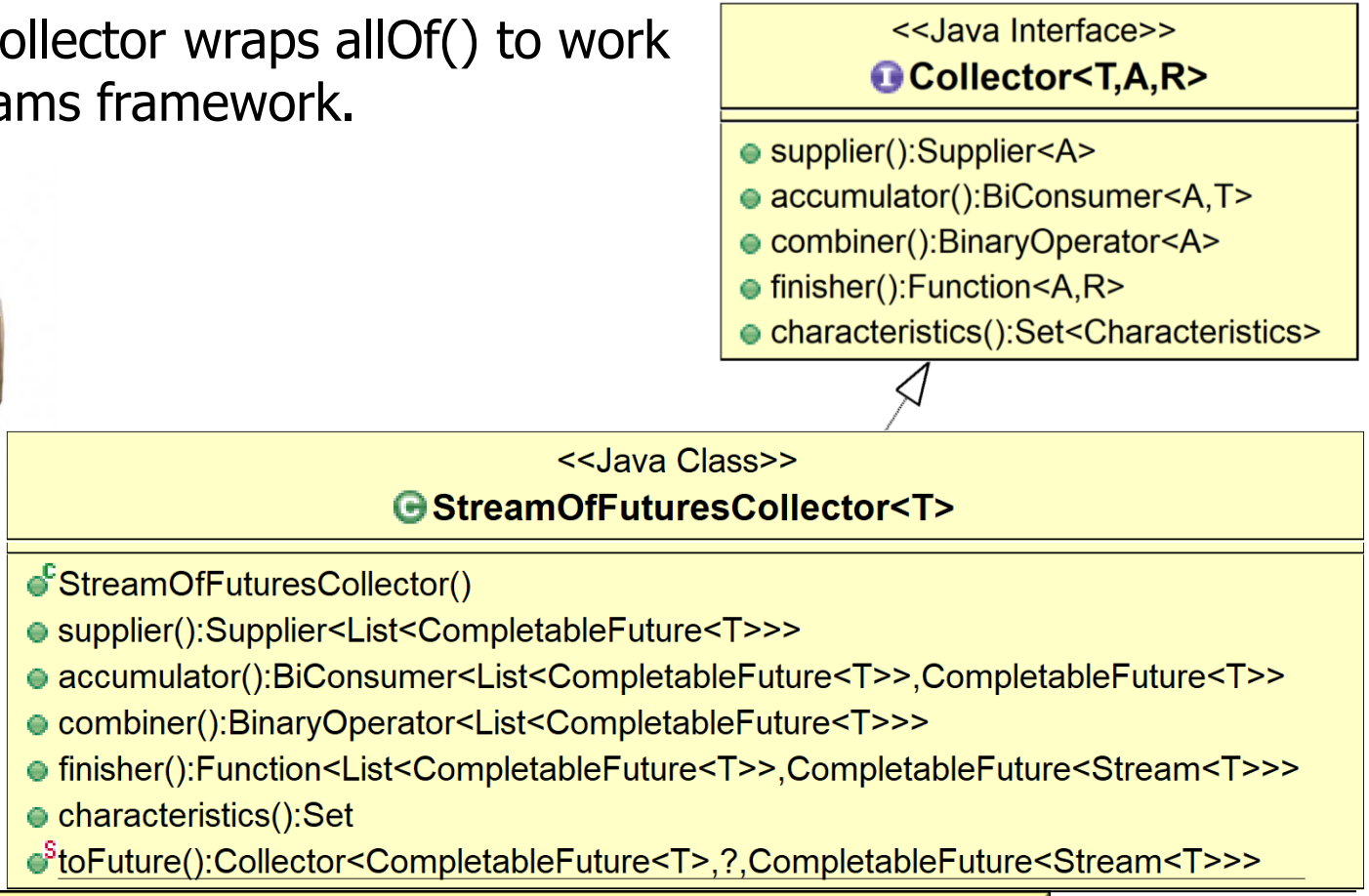
- Understand the design of the Java completable future version of ImageStreamGang.
- Know how to apply completable futures to ImageStreamGang
  - Factory methods
  - Completion stage methods
  - Arbitrary-arity methods
    - Wrap the allOf() method to work with the Java streams framework

```
<<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>
```

Implementing the Class StreamOfFuturesCollector

Implementing the Class StreamOfFuturesCollector

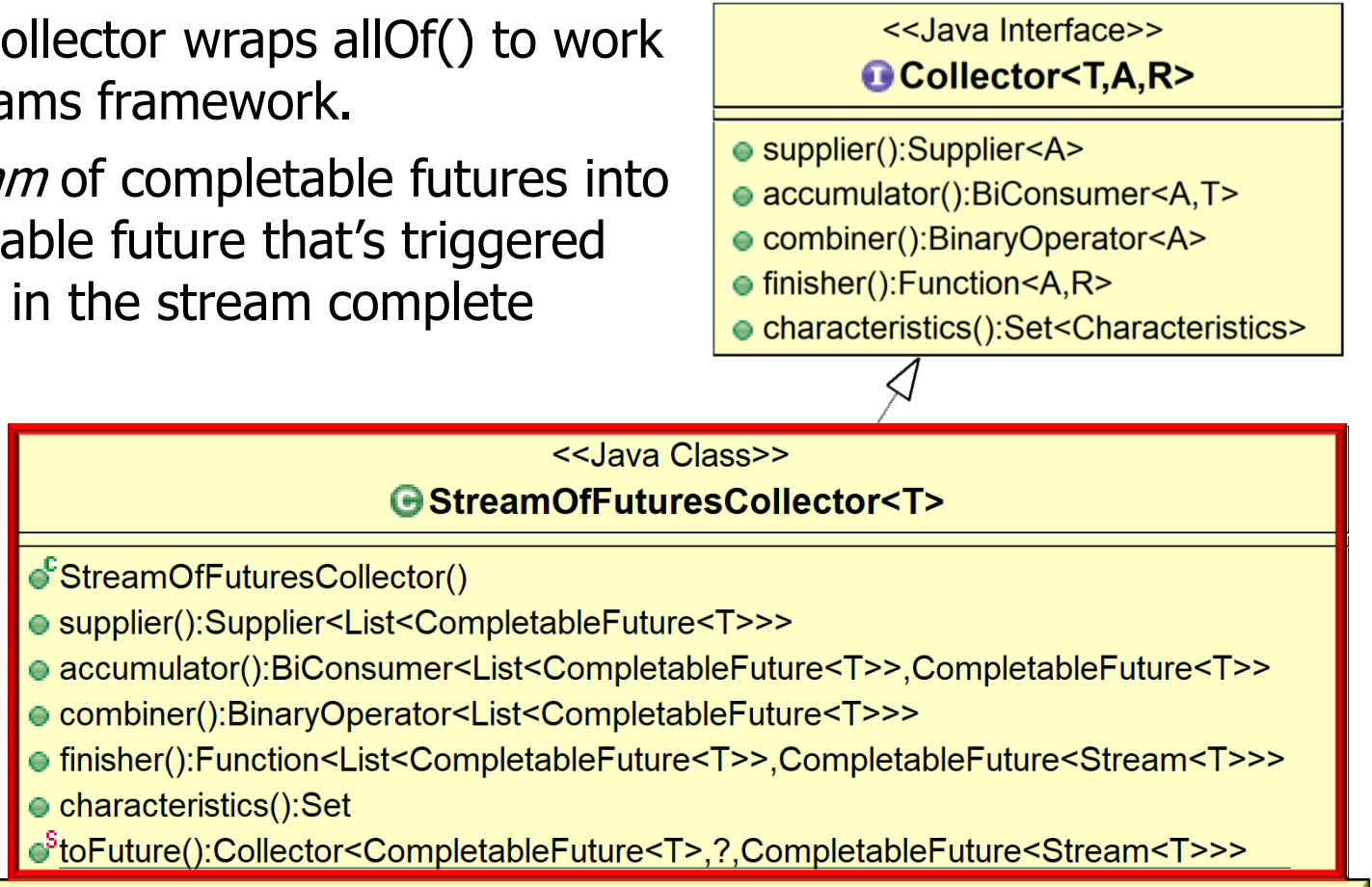
- StreamOfFuturesCollector wraps allOf() to work with the Java streams framework.



See livelessons/utils/StreamOfFuturesCollector.java

Implementing the Class StreamOfFuturesCollector

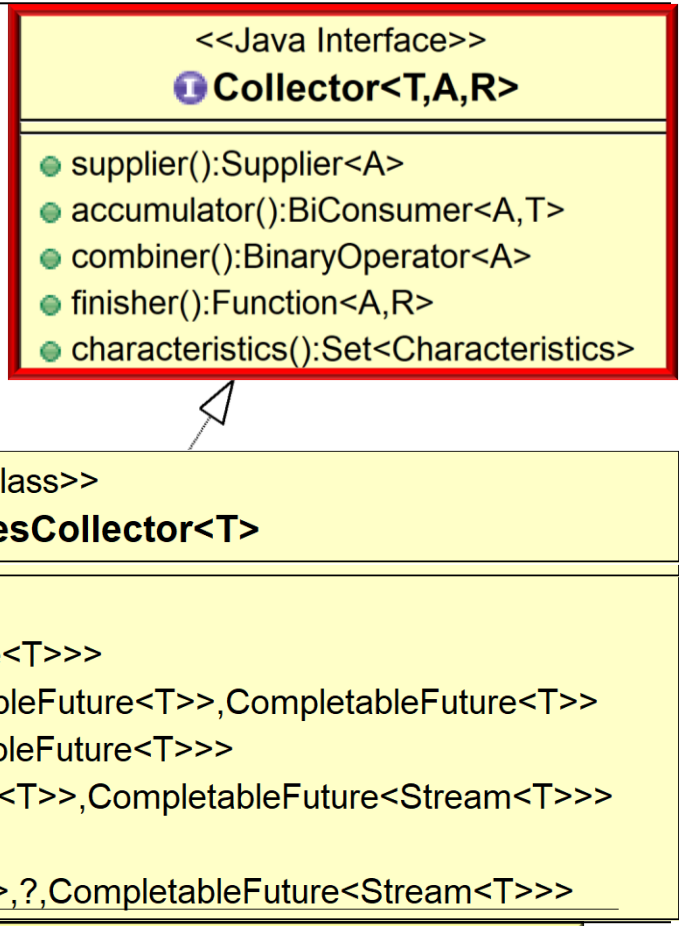
- StreamOfFuturesCollector wraps allOf() to work with the Java streams framework.
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete



StreamOfFuturesCollector is a nonconcurrent collector (supports parallel and sequential streams).

Implementing the Class StreamOfFuturesCollector

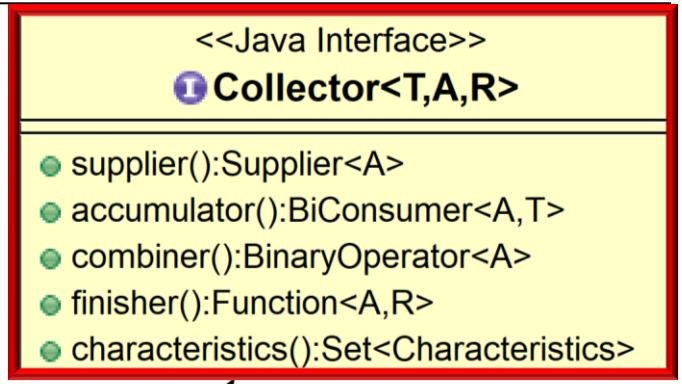
- StreamOfFuturesCollector wraps allOf() to work with the Java streams framework.
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete
  - Implements the Collector interface

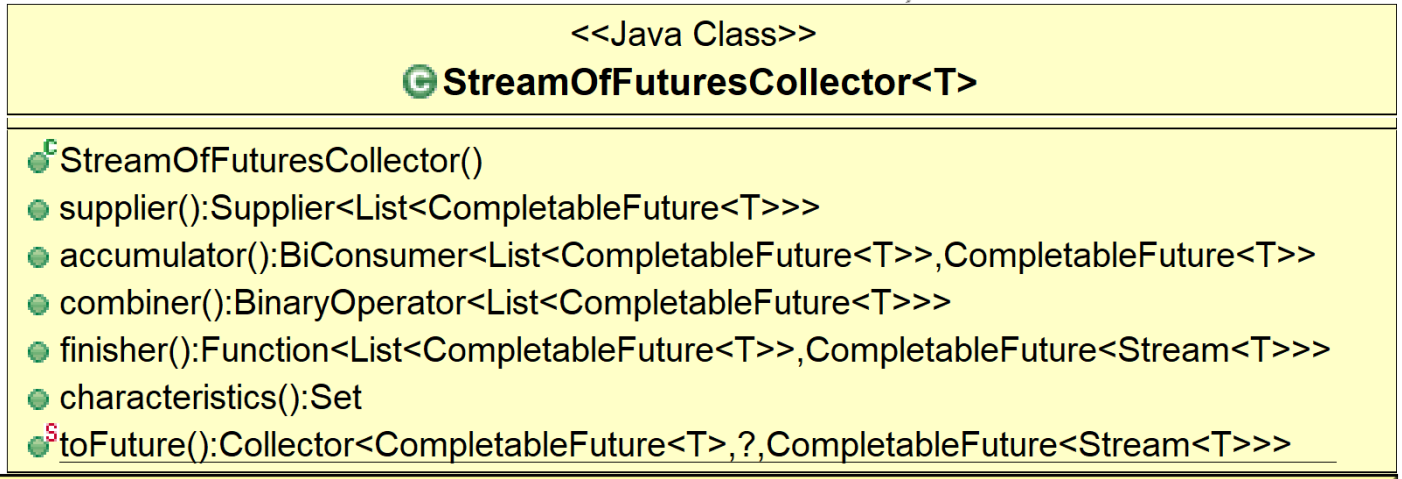


See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf() to work with the Java streams framework.
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete
  - Implements the Collector interface

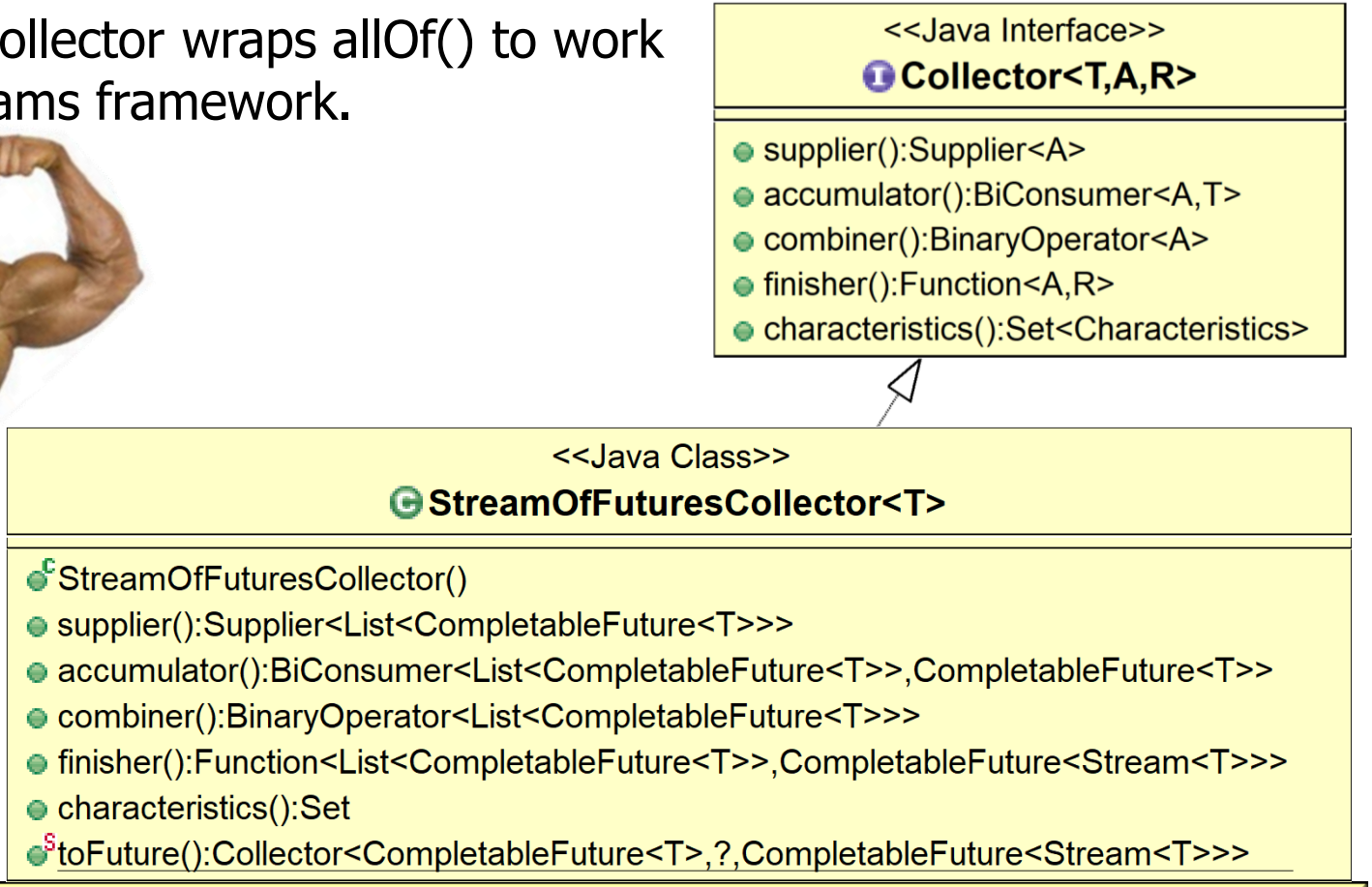




A collector accumulates input stream elements into a mutable result container.

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf() to work with the Java streams framework.



StreamOfFuturesCollector provides a powerful wrapper for some complex code!

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  ...
```

Implements a custom collector

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  ...
```

The type of input elements in the stream

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  ...
```

The mutable result container type

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  ...
```

The result type of final output of the collector

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  ...
```

The Stream<T> parameter differs from the List<T> parameter applied by the previous FuturesCollector.

See earlier lesson on "*Advanced Java Completable Future Features: Implementing FuturesCollector*"

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  public Supplier<List<CompletableFuture<T>>> supplier() {
    return ArrayList::new;
  }

  public BiConsumer<List<CompletableFuture<T>>,
                    CompletableFuture<T>> accumulator()
  { return List::add; }
  ...
```

This factory method returns a supplier used by the Java streams collector framework to create a new mutable array list container.

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  public Supplier<List<CompletableFuture<T>>> supplier() {
    return ArrayList::new;
  }
```

This mutable result container stores a list of completable futures of type T.

```
  public BiConsumer<List<CompletableFuture<T>>,
                    CompletableFuture<T>> accumulator()
  { return List::add; }
  ...
```

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
        implements Collector<CompletableFuture<T>,
                             List<CompletableFuture<T>>,
                             CompletableFuture<Stream<T>>> {
  public Supplier<List<CompletableFuture<T>>> supplier() {
    return ArrayList::new;
  }
```

This factory method returns a bi-consumer used by the Java streams collector framework to add a new completable future into the mutable array list container.

```
  public BiConsumer<List<CompletableFuture<T>>,
                    CompletableFuture<T>> accumulator()
  { return List::add; }
  ...
```

This method is only ever called in a single thread (so no locks are needed).

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public BinaryOperator<List<CompletableFuture<T>>> combiner() {
    return (List<CompletableFuture<T>> one,
            List<CompletableFuture<T>> another) -> {
        one.addAll(another);
        return one;
    };
  }
  ...
```

This factory method returns a binary operator that merges two partial array list results into a single array list (only relevant for parallel streams).

This method is only ever called in a single thread (so no locks are needed).

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<Stream<T>>> finisher(){
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))
```

This factory method returns a function used by the Java streams collector framework to transform the array list mutable result container to the completable future result type.

```
      .thenApply(v -> futures.stream()
                                .map(CompletableFuture::join));
  }
  ...
```

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<Stream<T>>> finisher(){
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                                .map(CompletableFuture::join));
  }
  ...
```

Reference to the mutable result container, which is an ArrayList

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<Stream<T>>> finisher(){
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))
```

Convert the list of futures to an array of futures and pass to allOf() to obtain a future that will complete when all futures complete

```
      .thenApply(v -> futures.stream()
                              .map(CompletableFuture::join));
  }
  ...
```

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<Stream<T>>> finisher(){
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))
      .thenApply(v -> futures.stream()
                                .map(CompletableFuture::join));
  }
  ...
```

When all futures have completed, get a single future to a stream of joined elements of type T.

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<Stream<T>>> finisher(){
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                                .map(CompletableFuture::join));
  }
  ...
```

Convert the array list of futures into a stream of futures

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<Stream<T>>> finisher(){
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                                .map(CompletableFuture::join));
  }
  ...
```

This call to join() will never block!

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<Stream<T>>> finisher(){
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                                .map(CompletableFuture::join));
  }
  ...
```

Return a future to a stream of elements of T

Implementing the Class StreamOfFuturesCollector

- toFuture() returns a future to a stream of futures to images that are being downloaded, filtered, and stored.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
    .join();
```

Provides a single means to await completion of a set of futures before continuing with the program

Implementing the Class StreamOfFuturesCollector

- toFuture() returns a future to a stream of futures to images that are being downloaded, filtered, and stored.

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

  CompletableFuture<Stream<Image>>
    resultsFuture = urls
    .stream()
    .map(this::checkUrlCachedAsync)
    .map(this::downloadImageAsync)
    .flatMap(this::applyFiltersAsync)
    .collect(toFuture())
    .thenApply(stream ->
                 log(stream.flatMap
                     (Optional::stream),
                      urls.size()))
```

thenApply() is called only after the future returned from collect() completes

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Set characteristics() {
    return Collections.singleton(Characteristics.UNORDERED);
  }

  public static <T> Collector<CompletableFuture<T>, ?,
                                CompletableFuture<Stream<T>>>
  toFuture() {
    return new StreamOfFuturesCollector<>();
  }
}
```

Returns a set indicating the characteristics of the StreamOfFutureCollector class

StreamOfFuturesCollector is thus a *nonconcurrent* collector.

Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector implements all methods in the Collector interface.

```
public class StreamOfFuturesCollector<T>
  ...
  public Set characteristics() {
    return Collections.singleton(Characteristics.UNORDERED);
  }

  public static <T> Collector<CompletableFuture<T>, ?,
                              CompletableFuture<Stream<T>>>
  toFuture() {
    return new StreamOfFuturesCollector<>();
  }
}
```

This static factory method creates a new StreamOfFuturesCollector.

Java Completable Futures ImageStreamGang Example: StreamOfFuturesCollector

The End

Evaluating the Pros of the Java Completable Futures Framework

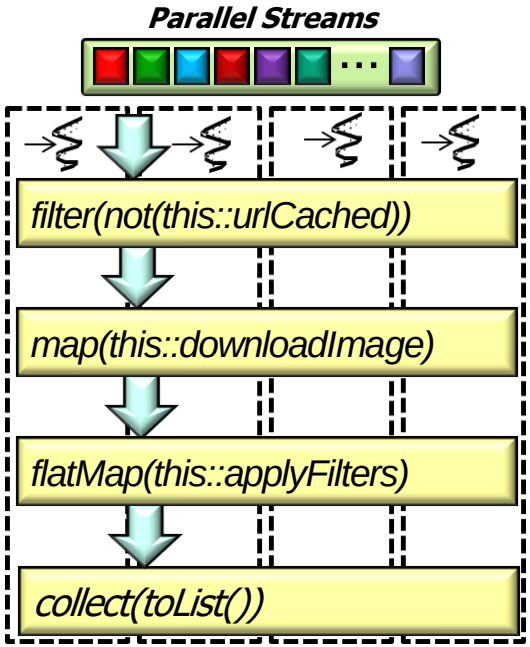
Douglas C. Schmidt

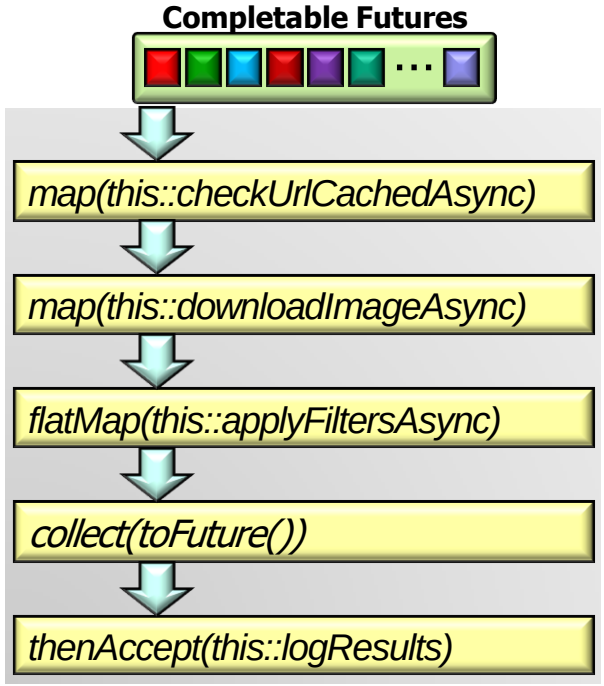
Learning Objectives in This Part of the Lesson

- Understand the pros of using the Java completable futures framework

Learning Objectives in This Part of the Lesson

- Understand the pros of using the Java completable futures framework
  - We evaluate the Java completable futures framework compared with the parallel streams framework.



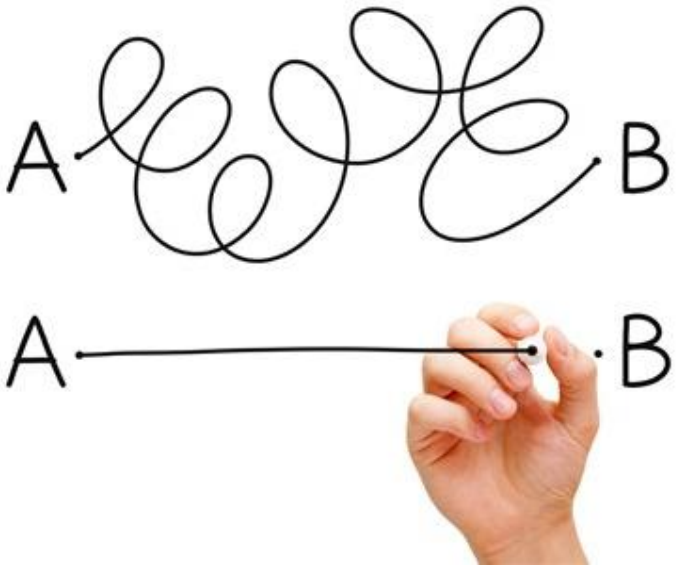


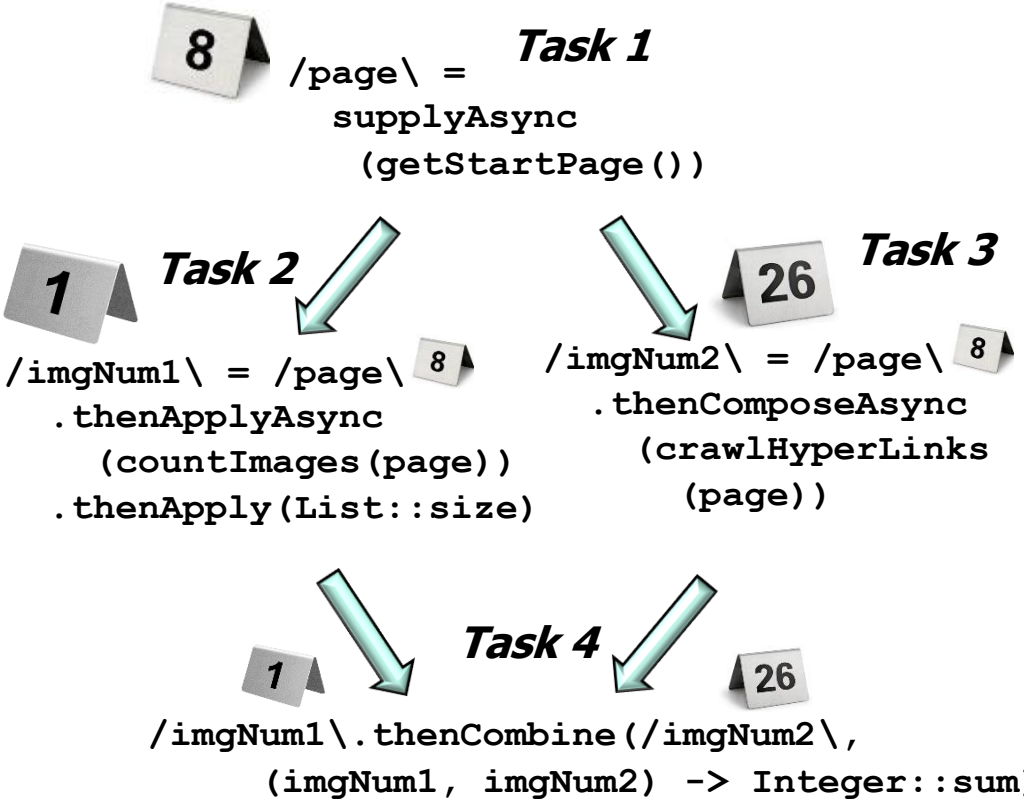
See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Pros of the Java Completable Futures Framework

Pros of the Java Completable Futures Framework

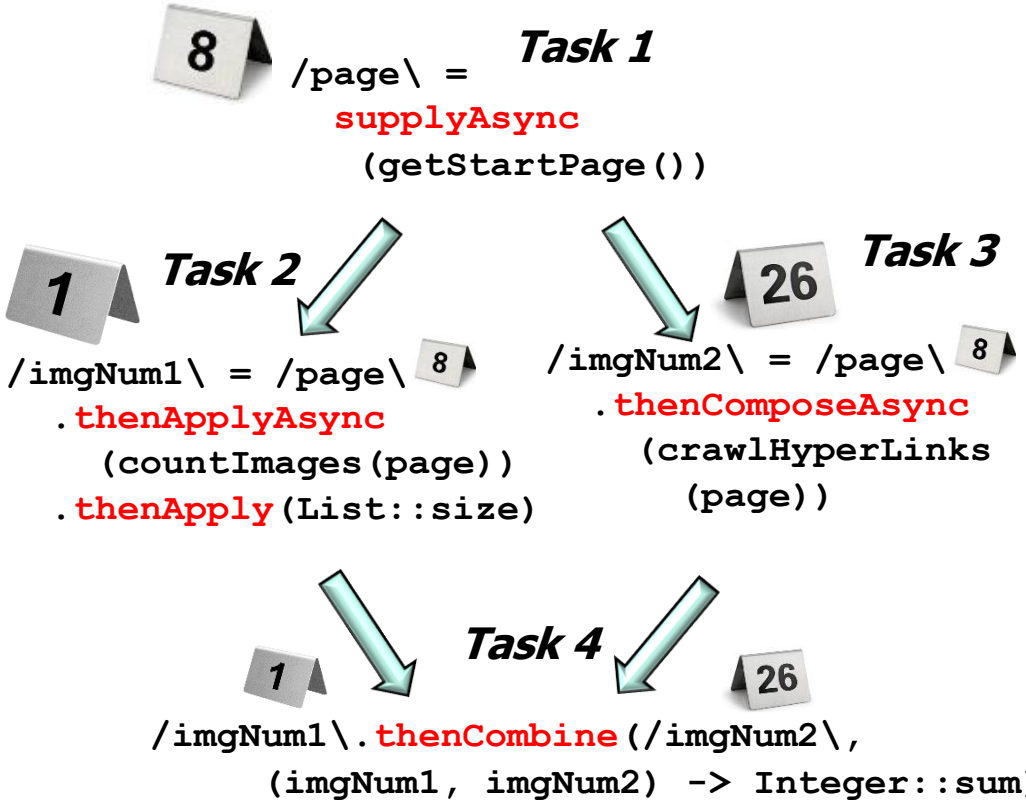
- Greatly simplifies programming of asynchronous operations





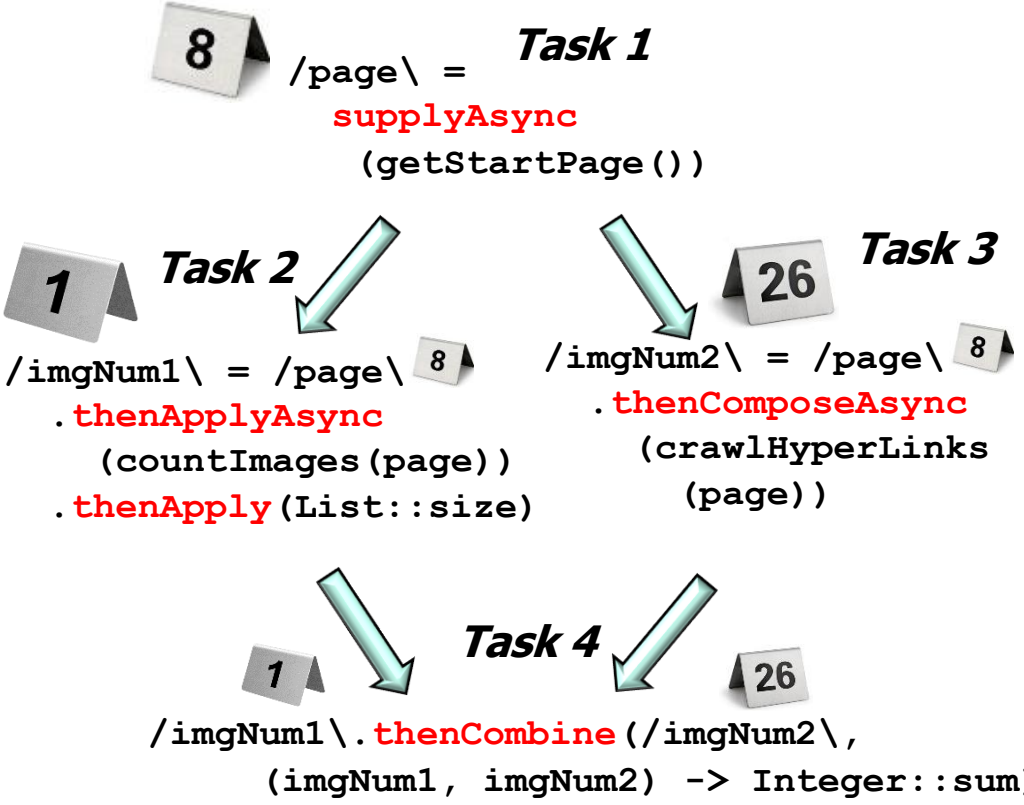
Pros of the Java Completable Futures Framework

- Greatly simplifies programming of asynchronous operations
  - Supports dependent actions that trigger upon completion of async operations



Pros of the Java Completable Futures Framework

- Greatly simplifies programming of asynchronous operations
  - Supports dependent actions that trigger upon completion of async operations
    - Async operations can be forked, chained, and joined in a relatively intuitive way



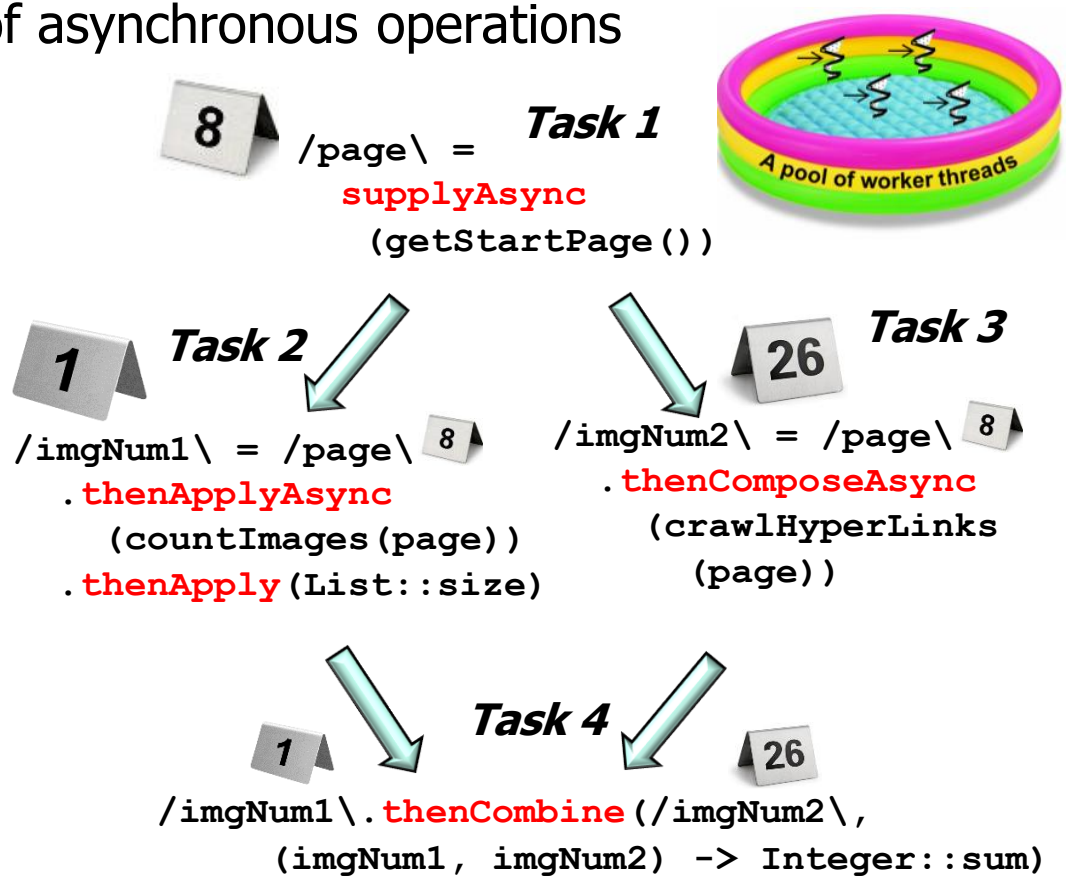
Pros of the Java Completable Futures Framework

- Greatly simplifies programming of asynchronous operations
  - Supports dependent actions that trigger upon completion of async operations
    - Async operations can be forked, chained, and joined in a relatively intuitive way
    - Enables async programs to appear like sync programs

```
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);
```

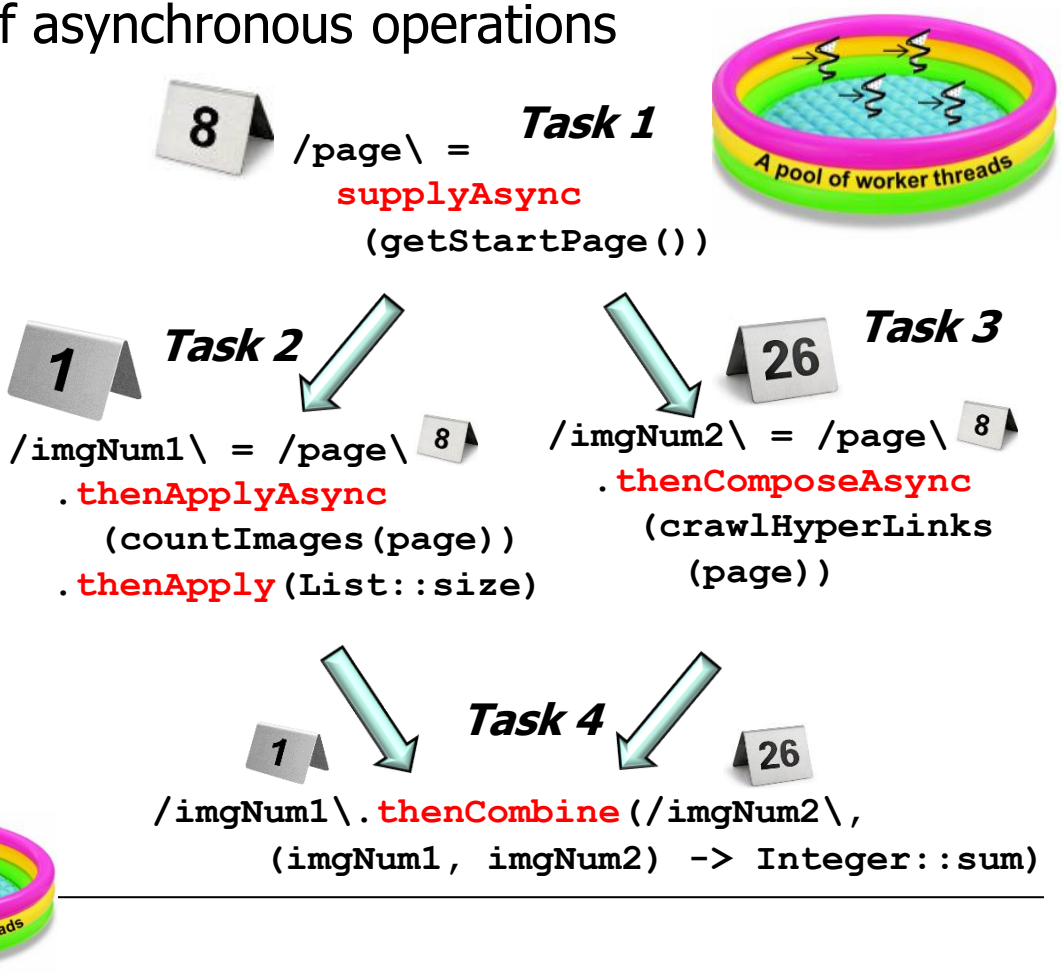
Pros of the Java Completable Futures Framework

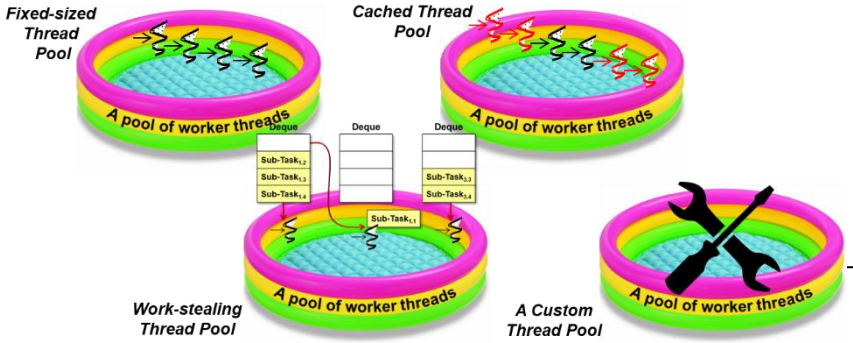
- Greatly simplifies programming of asynchronous operations
  - Supports dependent actions that trigger upon completion of async operations
  - Async operations run in parallel in a thread pool



Pros of the Java Completable Futures Framework

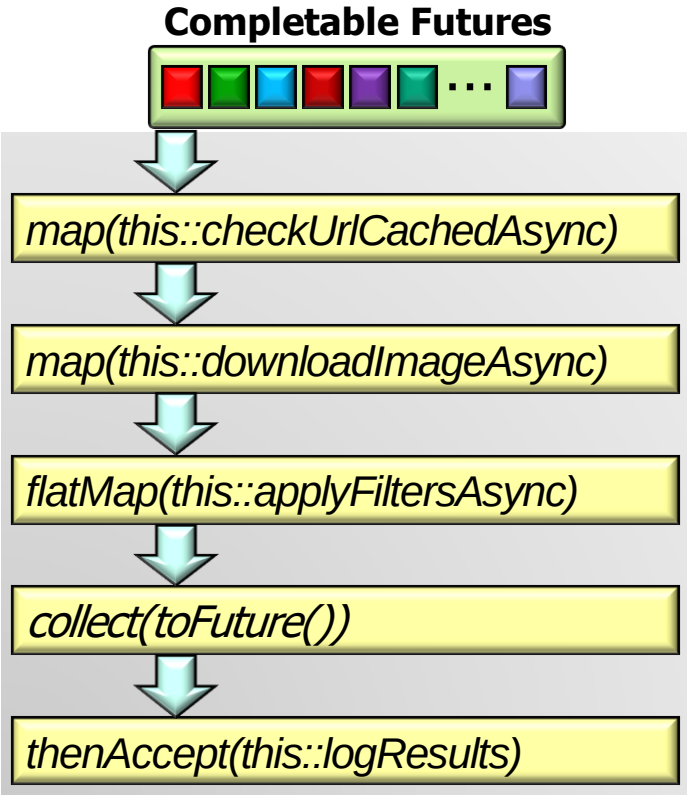
- Greatly simplifies programming of asynchronous operations
  - Supports dependent actions that trigger upon completion of async operations
  - Async operations run in parallel in a thread pool
    - Either a (common) fork-join pool or various types of pre- or user-defined thread pools





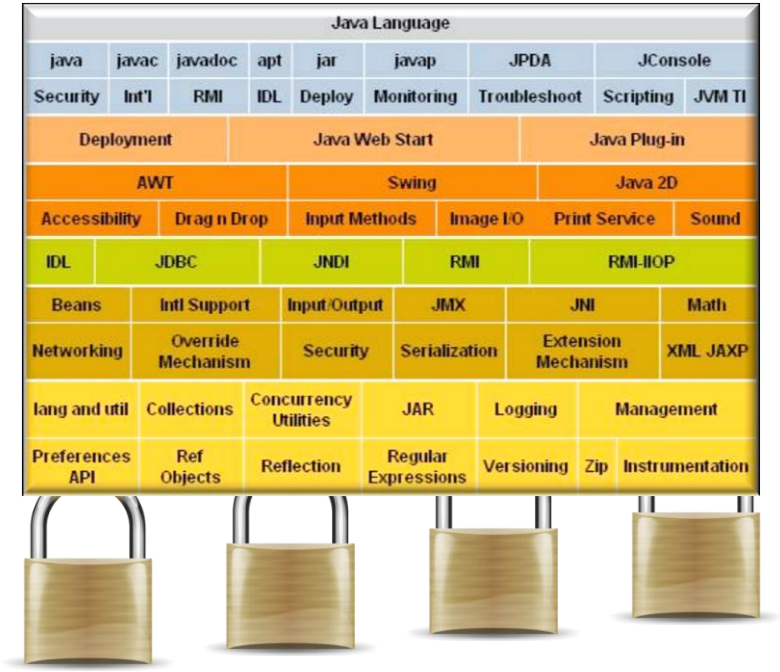
Pros of the Java Completable Futures Framework

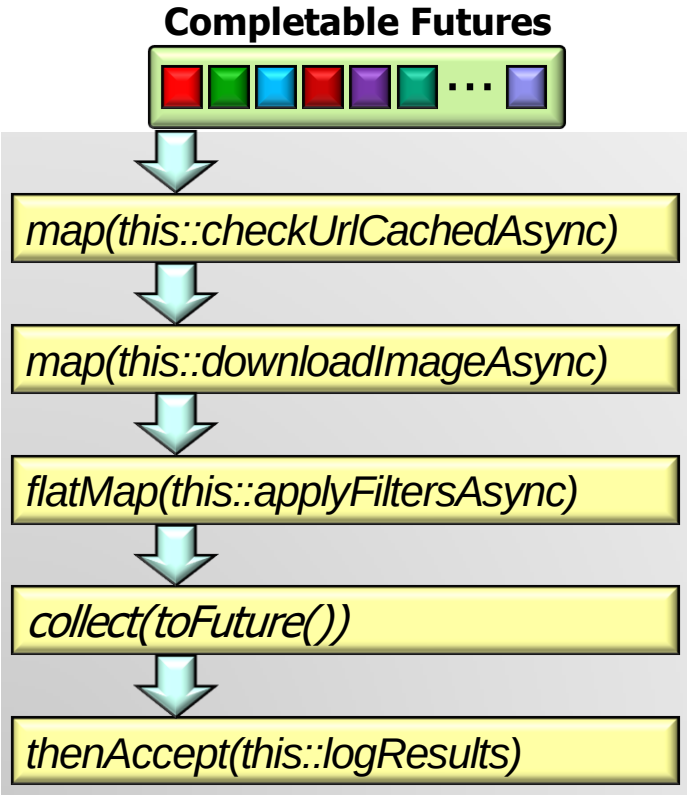
- No explicit synchronization or threading is required for completable futures.



Pros of the Java Completable Futures Framework

- No explicit synchronization or threading is required for completable futures.
  - Java libraries handle locking needed to protect shared mutable state.

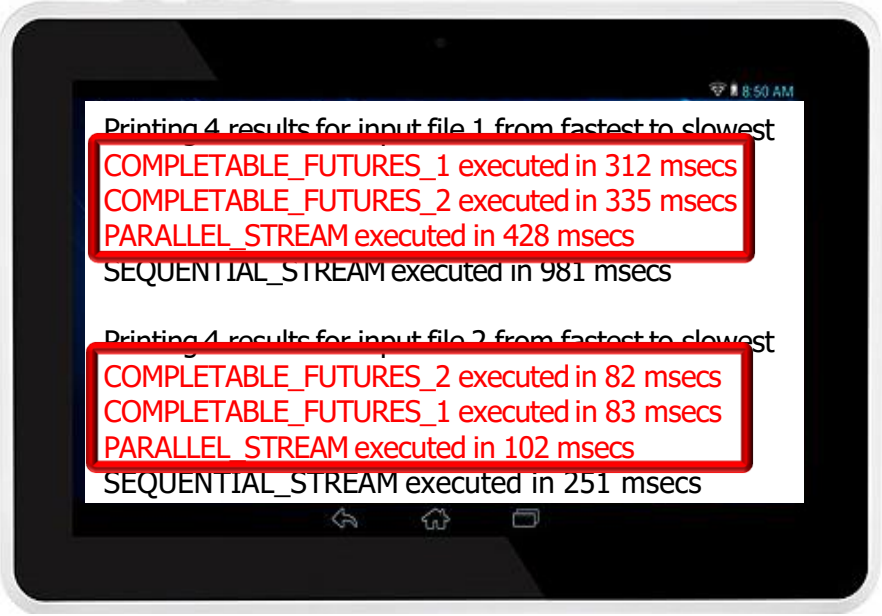


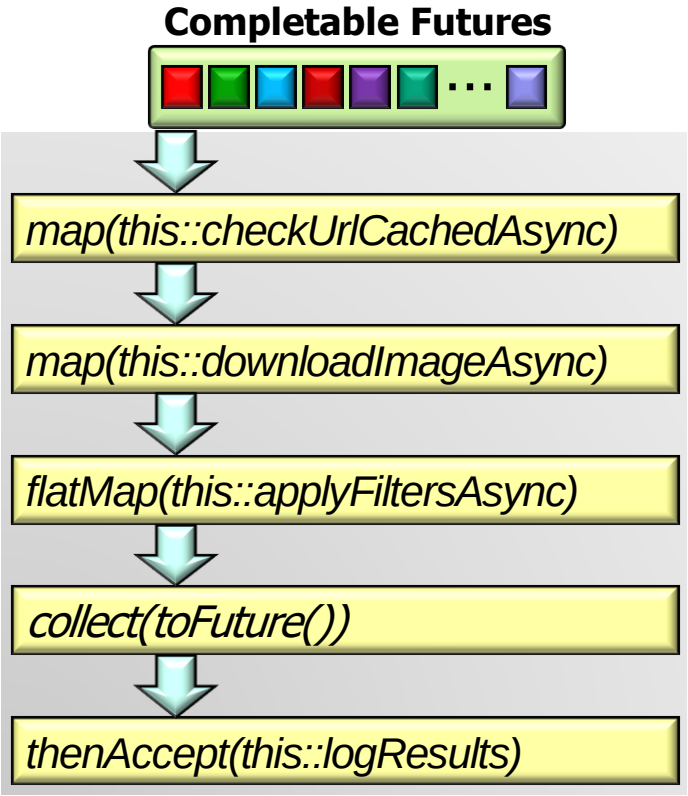


See docs.oracle.com/javase/tutorial/essential/concurrency/collections.html

Pros of the Java Completable Futures Framework

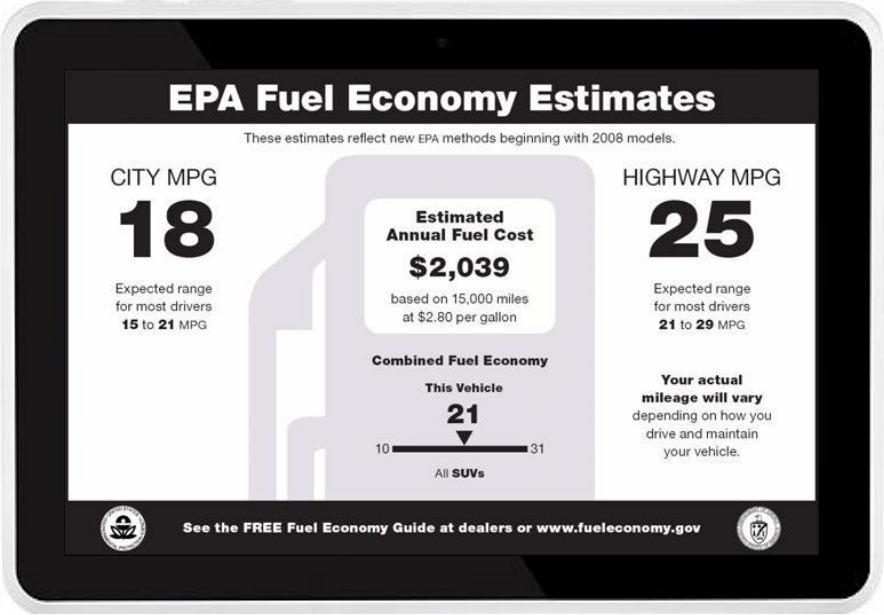
- Completable futures are often more efficient than parallel streams.

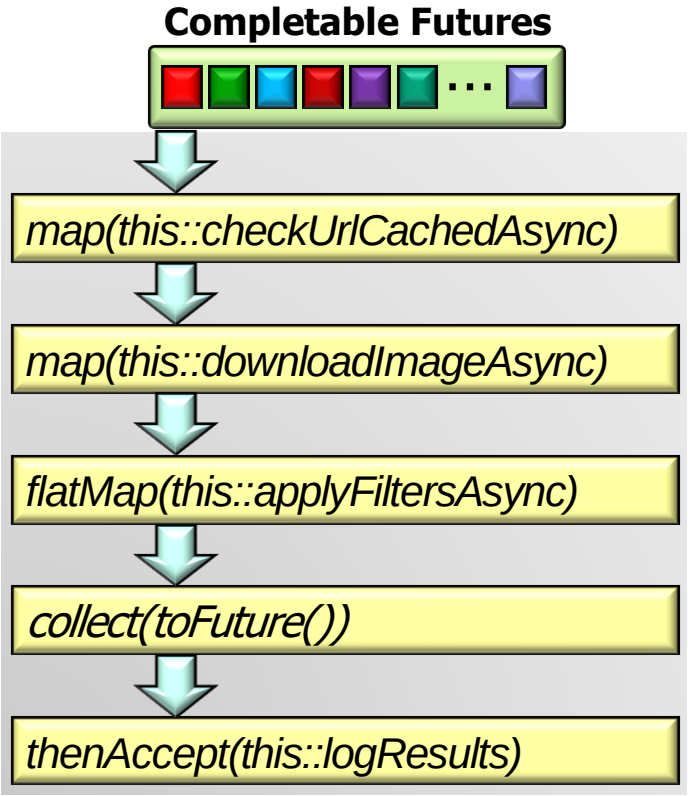




Pros of the Java Completable Futures Framework

- Completable futures are often more efficient than parallel streams.
  - Naturally, your mileage may vary…

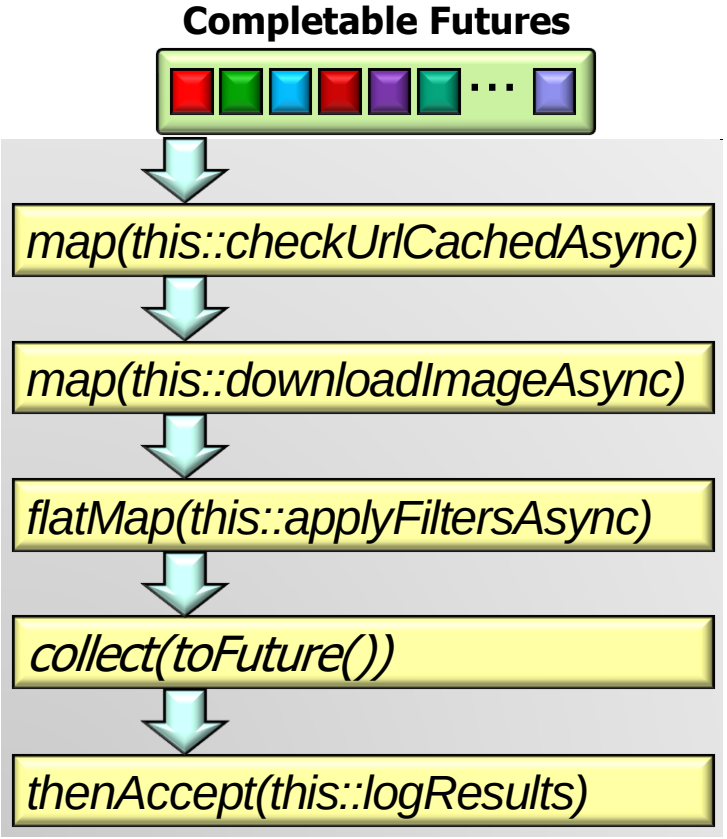




There's no substitute for benchmarking; e.g., java-performance.info/jmh!

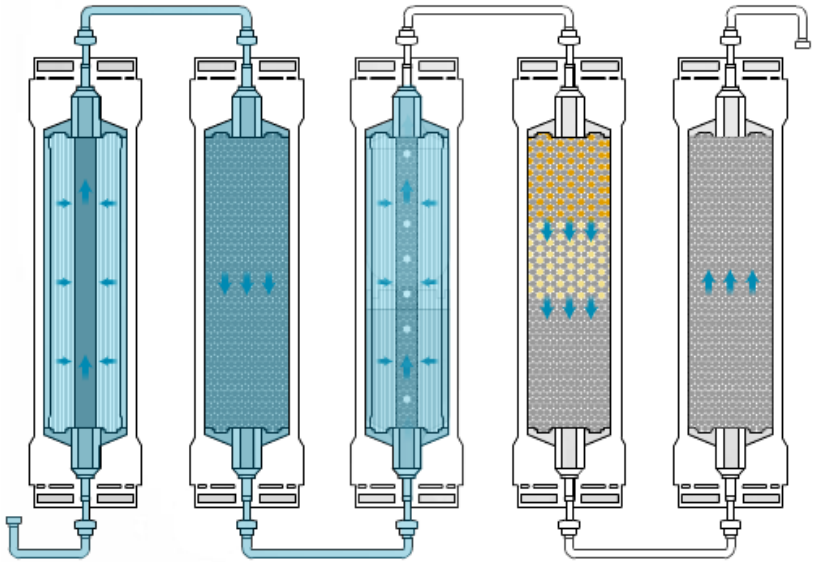
Pros of the Java Completable Futures Framework

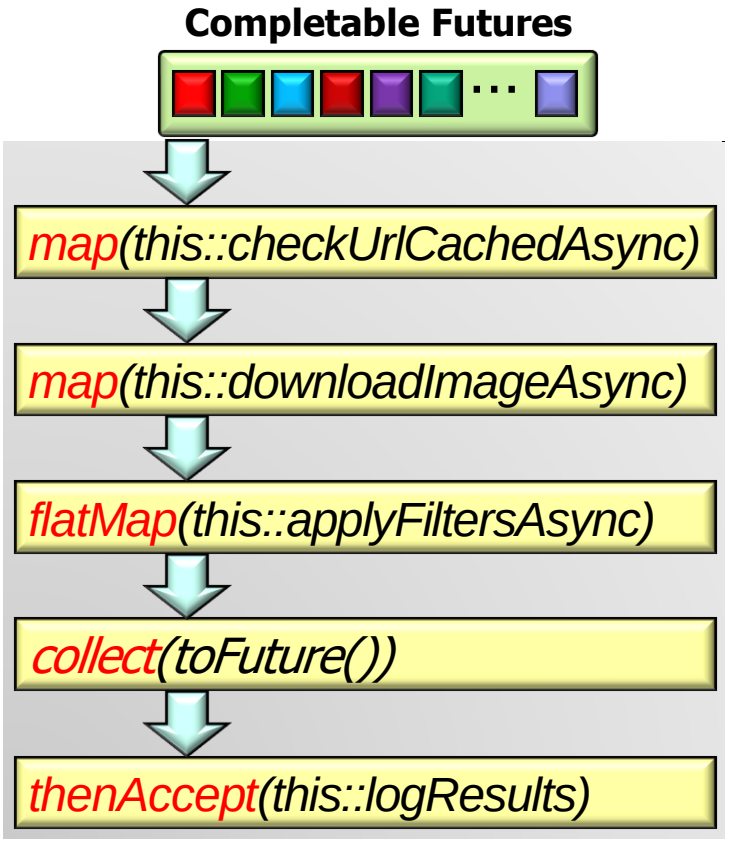
- Combining sequential streams and completable futures is often a win.



Pros of the Java Completable Futures Framework

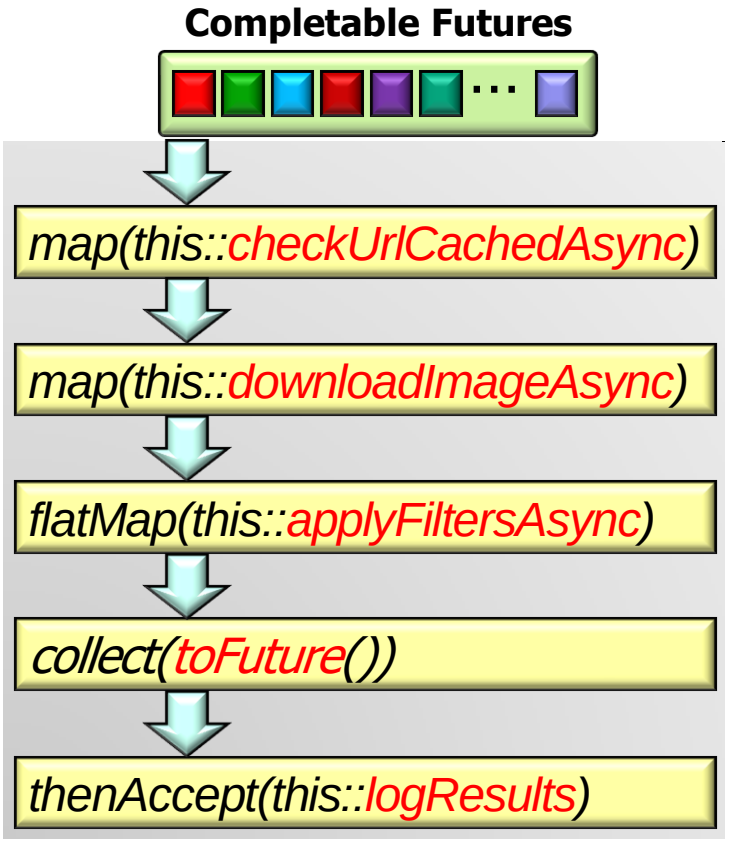
- Combining sequential streams and completable futures is often a win.
  - Streams guide the overall flow of control.





Pros of the Java Completable Futures Framework

- Combining sequential streams and completable futures is often a win.
  - Streams guide the overall flow of control and completable futures efficiently perform async operations in parallel.



Pros of the Java Completable Futures Framework

- Combining sequential streams and completable futures is often a win.
  - Streams guide the overall flow of control and completable futures efficiently perform async operations in parallel.
  - However, combining parallel streams and completable futures is overkill.

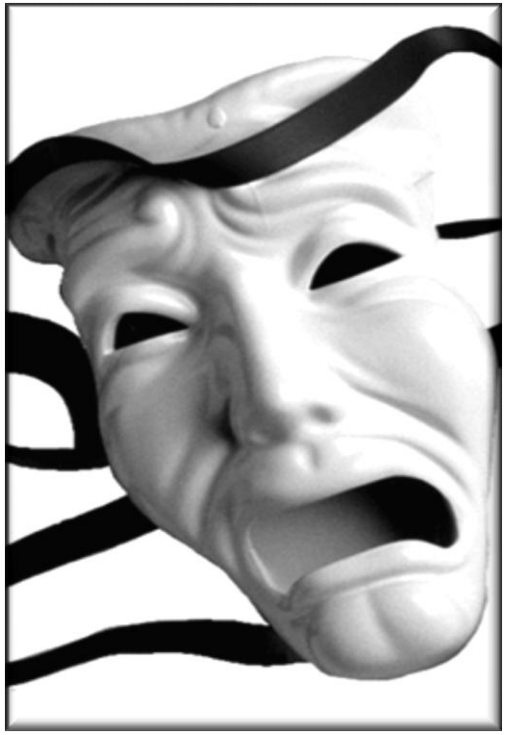
The End

Evaluating the Cons of the Java Completable Futures Framework

Douglas C. Schmidt

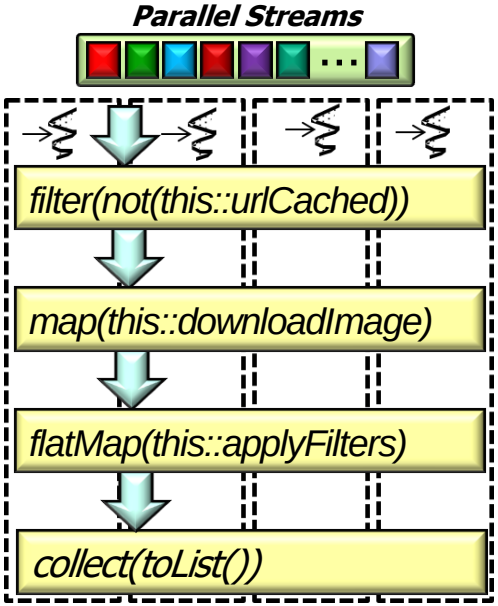
Learning Objectives in This Part of the Lesson

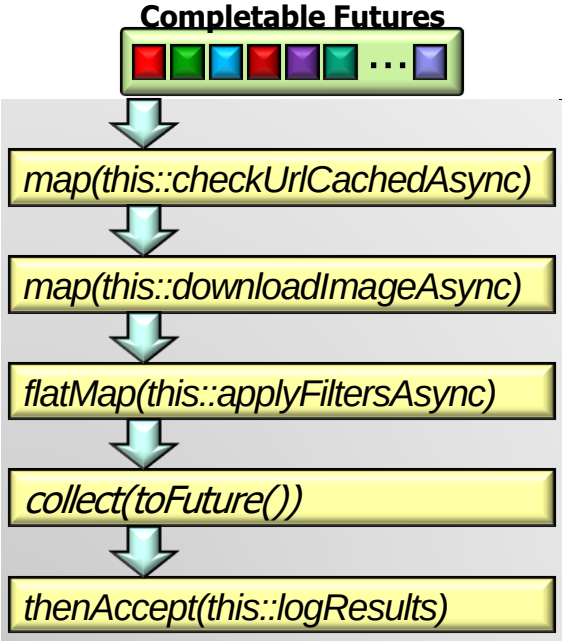
- Understand the pros of using the Java completable futures framework
- Understand the cons of using the Java completable futures framework



Learning Objectives in This Part of the Lesson

- Understand the pros of using the Java completable futures framework
- Understand the cons of using the Java completable futures framework
  - Again, we evaluate the Java completable futures framework compared with the parallel streams framework.





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

Cons of the Java Completable Futures Framework

Cons of the Java Completable Futures Framework

- It's easier to program Java parallel streams than completable futures.

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

  List<Image> images =
  urls
  .parallelStream()
  .filter(not(this::urlCached))
  .map(this::blockingDownload)
  .flatMap(this::applyFilters)
  .collect(toList());

  logResults(images); ...
```

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

  CompletableFuture<Stream<Image>>
  resultsFuture = urls
  .stream()
  .map(this::checkUrlCachedAsync)
  .map(this::downloadImageAsync)
  .flatMap(this::applyFiltersAsync)
  .collect(toFuture())
  .thenApply(this::logResults)
  .join(); ...
```

Cons of the Java Completable Futures Framework

- It's easier to program Java parallel streams than completable futures
  - The overall control flow is similar when using the Java streams framework

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

  List<Image> images =
  urls
  .parallelStream()
  .filter(not(this::urlCached))
  .map(this::blockingDownload)
  .flatMap(this::applyFilters)
  .collect(toList());

  logResults(images); ...
```

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

  CompletableFuture<Stream<Image>>
  resultsFuture = urls
  .stream()
  .map(this::checkUrlCachedAsync)
  .map(this::downloadImageAsync)
  .flatMap(this::applyFiltersAsync)
  .collect(toFuture())
  .thenApply(this::logResults)
  .join(); ...
```

Cons of the Java Completable Futures Framework

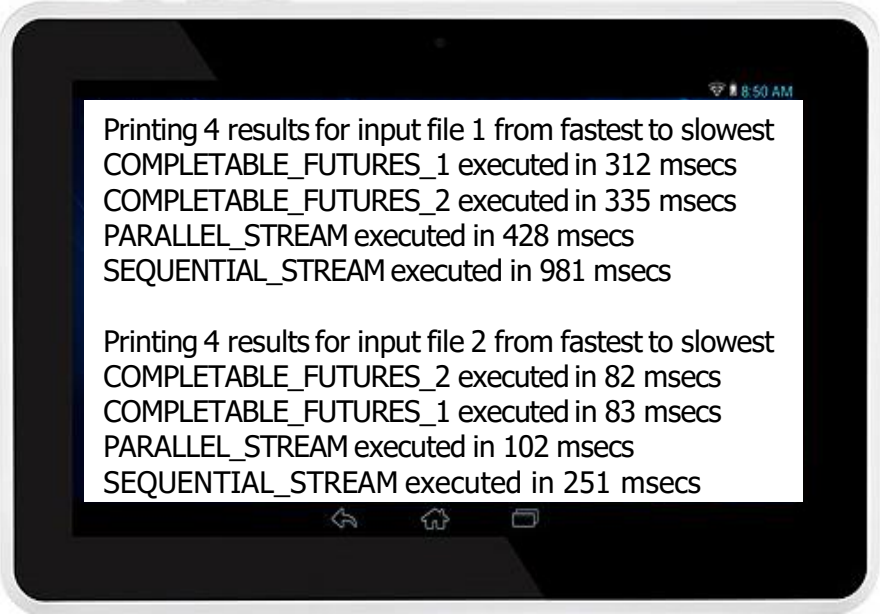
- It's easier to program Java parallel streams than completable futures
  - The overall control flow is similar when using the Java streams framework
  - However, async behaviors are more complicated than the sync behaviors!

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

  List<Image> images =
  urls
  .parallelStream()
  .filter(not(this::urlCached))
  .map(this::blockingDownload)
  .flatMap(this::applyFilters)
  .collect(toList());

  logResults(images); ...
```

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

  CompletableFuture<Stream<Image>>
  resultsFuture = urls
  .stream()
  .map(this::checkUrlCachedAsync)
  .map(this::downloadImageAsync)
  .flatMap(this::applyFiltersAsync)
  .collect(toFuture())
  .thenApply(this::logResults)
  .join(); ...
```

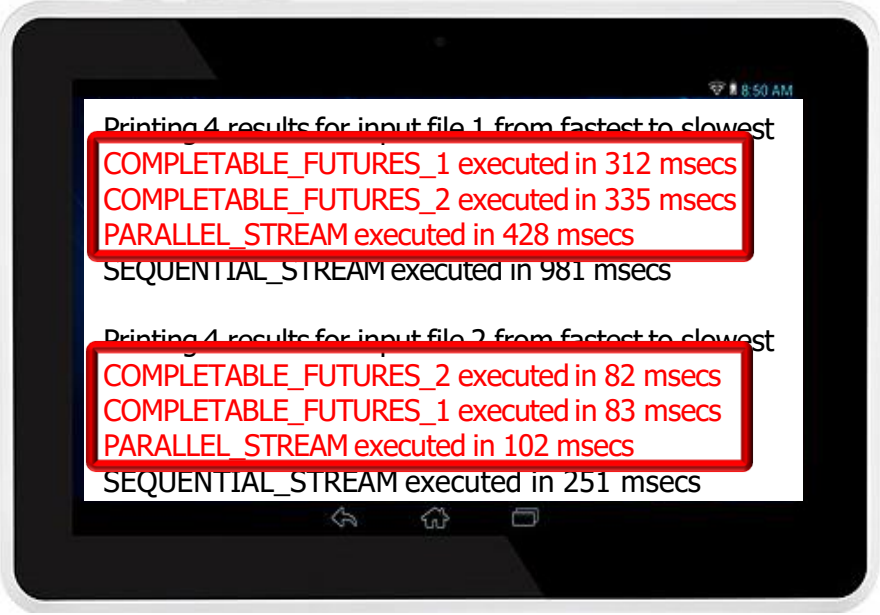
Cons of the Java Completable Futures Framework

- There is a tradeoff between computing performance and programmer productivity when choosing amongst these frameworks.



Cons of the Java Completable Futures Framework

- There is a tradeoff between computing performance and programmer productivity when choosing amongst these frameworks.
  - Completable futures are more efficient and scalable but are harder to program.



In general, asynchrony patterns aren't well understood by many developers

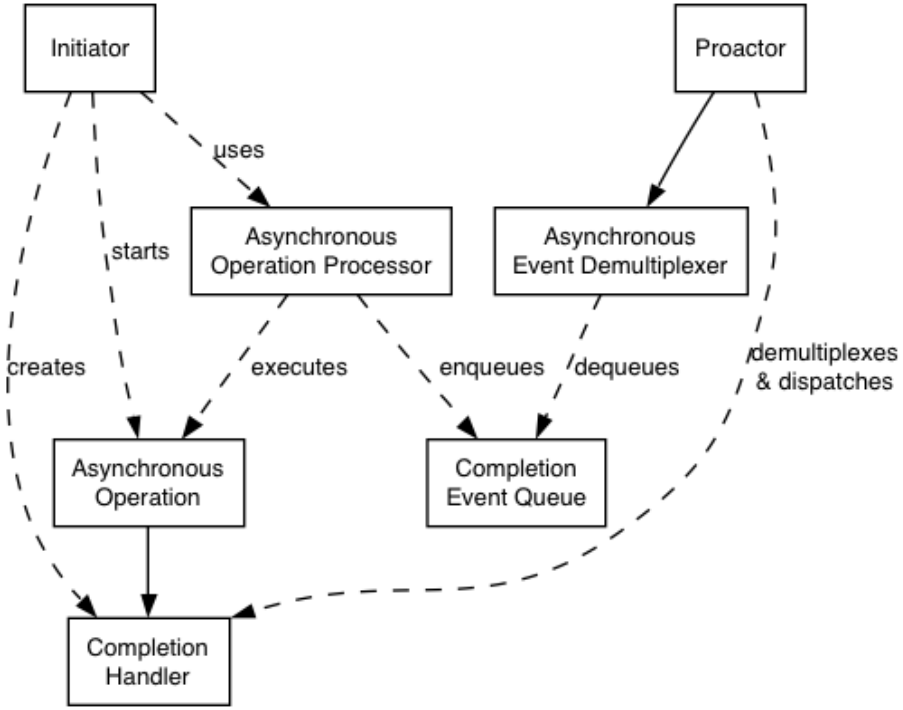
Cons of the Java Completable Futures Framework

- There is a tradeoff between computing performance and programmer productivity when choosing amongst these frameworks.
  - Completable futures are more efficient and scalable but are harder to program.
  - Parallel streams are easier to program, but are less efficient and scalable.

Use sequential streams for initial development and then trivially make them parallel!

Cons of the Java Completable Futures Framework

- As usual, it is essential to know the best practices and patterns needed to program completable futures effectively!



The End

Enhancements to the Java Completable Futures Framework

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the pros of using the Java completable futures framework
- Understand the cons of using the Java completable futures framework
- Be aware of enhancements to the Java completable futures framework

Enhancements to the Java Completable Futures Framework

Enhancements to the Java Completable Futures Framework

- Java 9 provides enhancements to the Java 8 completable future framework.

Methods	Params		
default Executor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
complete Async	Supplier<T>	Completable Future<T>	Complete *CompletableFuture* asynchro-nously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	Completable Future<T>	Resolves *CompletableFuture* exception-ally with *TimeoutException*, unless it is completed before the specified timeout
complete OnTimeout	T value, long timeout, TimeUnit unit	Completable Future<T>	Completes this CompletableFuture with the given value if not otherwise completed before the given timeout

See www.baeldung.com/java-9-completablefuture

Enhancements to the Java Completable Futures Framework

- Java 9 provides enhancements to the Java 8 completable future framework.

Methods	Params		
default Executor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
complete Async	Supplier<T>	Completable Future<T>	Complete *CompletableFuture* asynchronously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	Completable Future<T>	Resolves *CompletableFuture* exceptionally with *TimeoutException*, unless it is completed before the specified timeout
complete OnTimeout	T value, long timeout, TimeUnit unit	Completable Future<T>	Completes this CompletableFuture with the given value if not otherwise completed before the given timeout

See docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletableFuture.html#defaultExecutor

Enhancements to the Java Completable Futures Framework

- Java 9 provides enhancements to the Java 8 completable future framework.

Methods	Params		
default Executor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
complete Async	Supplier<T>	Completable Future<T>	Complete *CompletableFuture* asynchro-nously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	Completable Future<T>	Resolves *CompletableFuture* exception-ally with *TimeoutException*, unless it is completed before the specified timeout
complete OnTimeout	T value, long timeout, TimeUnit unit	Completable Future<T>	Completes this CompletableFuture with the given value if not otherwise completed before the given timeout

See docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletableFuture.html#completeAsync

Enhancements to the Java Completable Futures Framework

- Java 9 provides enhancements to the Java 8 completable future framework.

Methods	Params		
default Executor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
complete Async	Supplier<T>	Completable Future<T>	Complete *CompletableFuture* asynchronously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	Completable Future<T>	Resolves *CompletableFuture* exceptionally with *TimeoutException*, unless it is completed before the specified timeout
complete OnTimeout	T value, long timeout, TimeUnit unit	Completable Future<T>	Completes this CompletableFuture with the given value if not otherwise completed before the given timeout

See docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletableFuture.html#orTimeout

Enhancements to the Java Completable Futures Framework

- Java 9 provides enhancements to the Java 8 completable future framework.

Methods	Params		
default Executor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
complete Async	Supplier<T>	Completable Future<T>	Complete *CompletableFuture* asynchronously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	Completable Future<T>	Resolves *CompletableFuture* exceptionally with *TimeoutException*, unless it is completed before the specified timeout
complete OnTimeout	T value, long timeout, TimeUnit unit	Completable Future<T>	Completes this CompletableFuture with the given value if not otherwise completed before the given timeout

See docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletableFuture.html#completeOnTimeout

Applying orTimeout() and completeOnTimeout()

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   })...
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex27

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   }) . . .
```

Asynchronously find the best price for a flight

supplyAsync() uses the default executor (i.e., common fork-join pool)

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   })...
```

Asynchronously find the latest exchange rate

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   })...
```

Select the default rate if call runs longer than 2 seconds

See iteratrlearning.com/java9/2016/09/13/java9-timeouts-completablefutures.html

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   })...
```

Combine and convert the search results

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   }) . . .
```

Throws TimeoutException if the time period elapses

See 4comprehension.com/completablefuture-timeout

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   })...
```

This method is always called, w/ or w/out an exception being thrown

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   })...
```

Print results if async calls finished normally

Applying orTimeout() and CompleteOnTimeout()

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 2, SECONDS),
                this::convert)
   .orTimeout(3, SECONDS)
   .whenComplete((amount, ex) -> {
     if (amount != null)
       { System.out.println("The price is: " + amount + "GBP"); }
     else { System.out.println(ex.toString()); }
   })...
```

Print the exception if async call timed out

Applying completeAsync() to Big Fractions

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";


  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";
```

This string builder holds intermediate results

```
  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";
```

Define a pair of big fractions

```
  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";
```

Create a new empty completable future

```
  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";
```

Register an action that appends a string when run (in calling thread)

```
  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

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

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";
```

Arrange to execute a supplier lambda in common fork-join pool

```
  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

See docs.oracle.com/javase/9/docs/api/java/util/concurrent/CompletableFuture.html#completeAsync

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";


  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

This method sets all the processing in motion

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";


  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2);});
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

This supplier lambda is used to multiply two BigFraction objects

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";


  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2);});
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

These computations run concurrently

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";


  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

join() blocks until result is complete

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";


  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

Append the mixed fraction to sb

Applying completeAsync() to Big Fractions

- Using completeAsync() to multiply big fractions

```
void testFractionMultiplicationCompleteAsync() {
  StringBuilder sb = new StringBuilder(">> Starting test\n");
  String f1 = "62675744/15668936"; String f2 = "609136/913704";


  CompletableFuture<BigFraction> f = new CompletableFuture<>();
  f.thenRun(() -> sb.append("completeAsync() result = "));
  f.completeAsync(() -> {
      BigFraction bf1 = new BigFraction(f1);
      BigFraction bf2 = new BigFraction(f2);
      return bf1.multiply(bf2); });
  ...
  sb.append(f.join().toMixedString()); display(sb.toString());
}
```

Display output as a string

The End