Evaluating the Cons of the Java Completable Futures Framework

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

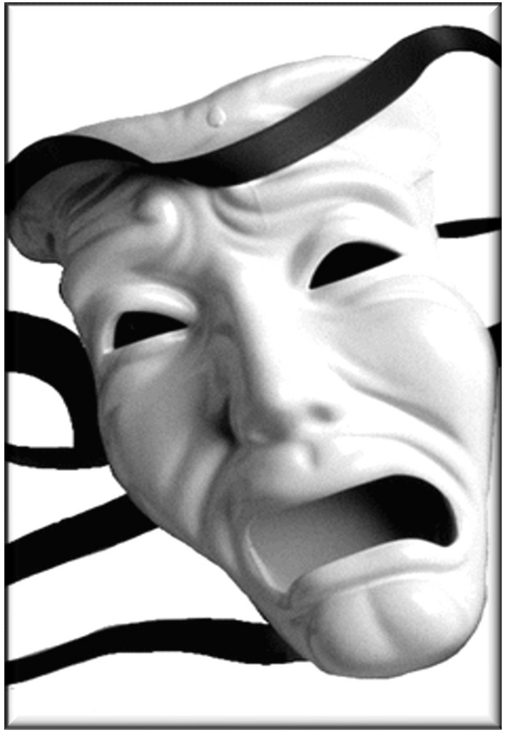
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

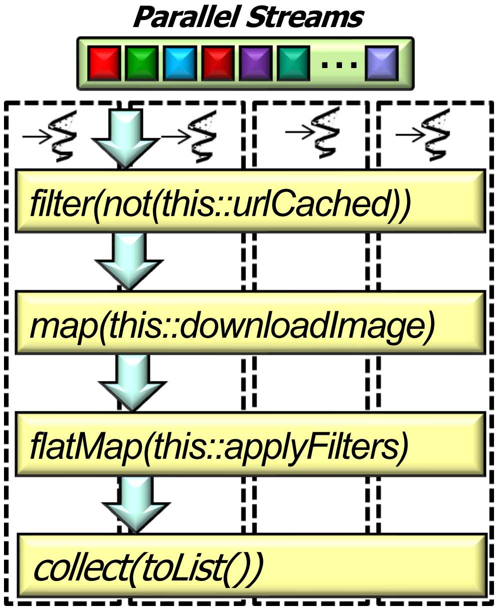
Learning Objectives in this Part of the Lesson

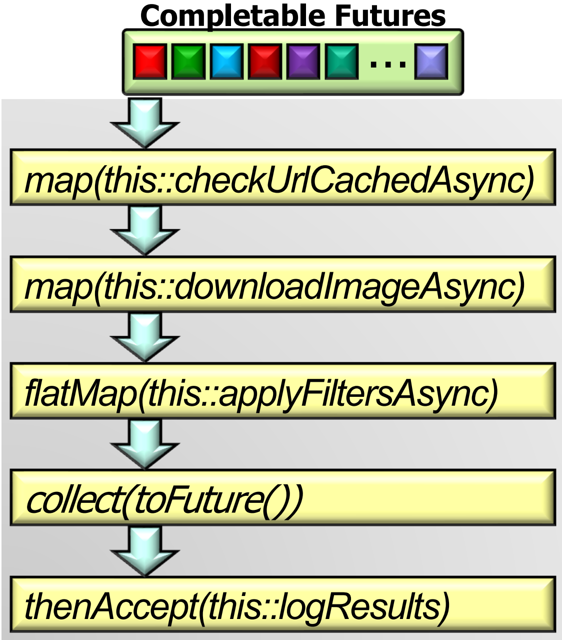
- Evaluate the pros of using the Java completable futures framework
- Evaluate the cons of using the Java completable futures framework



Learning Objectives in this Part of the Lesson

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




Completable Futures
...
map(this::checkUrlCachedAsync)
map(this::downloadImageAsync)
flatMap(this::applyFiltersAsync)
collect(toFuture())
thenAccept(this::logResults)

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)
  .mapMulti(this::applyFilters)
  .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)
  .mapMulti(this::applyFilters)
  .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)
  .mapMulti(this::applyFilters)
  .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)
  .mapMulti(this::applyFilters)
  .toList();

  logResults(images); ...
```

These behaviors use two-way synchronous operations & quickly discard cached images from consideration

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!

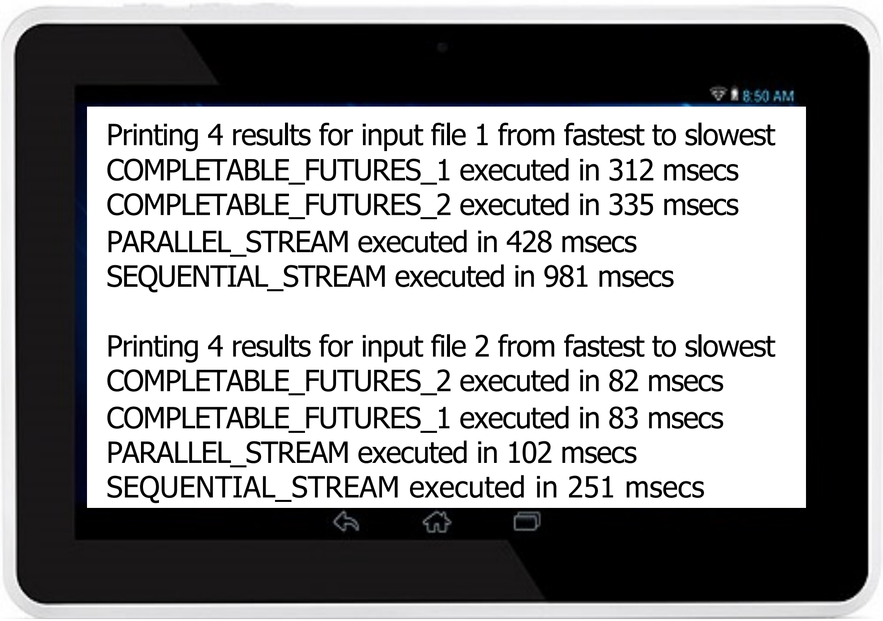
These behaviors use complex asynchronous operations & must propagate Optional cached images thru the stream

```
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's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks

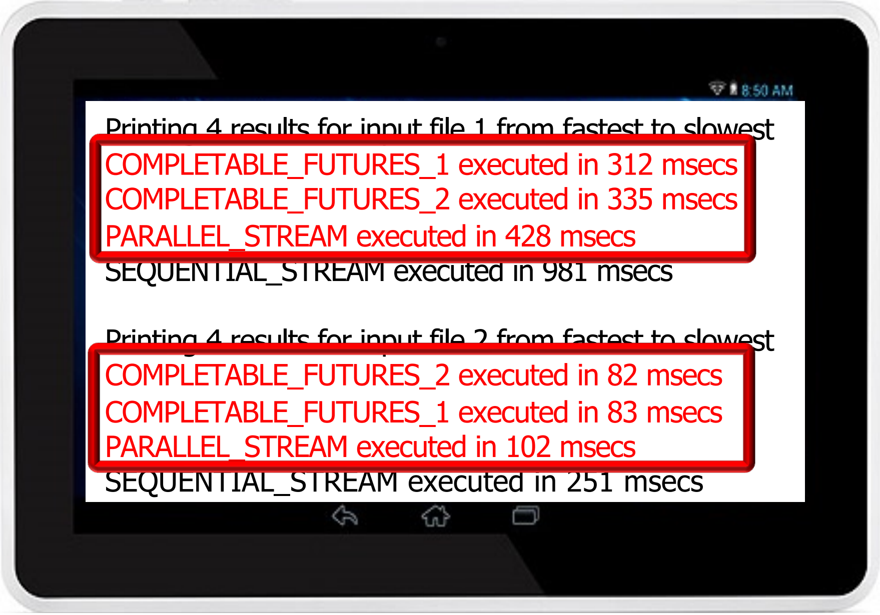


```
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 msecs
```

Cons of the Java Completable Futures Framework

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program

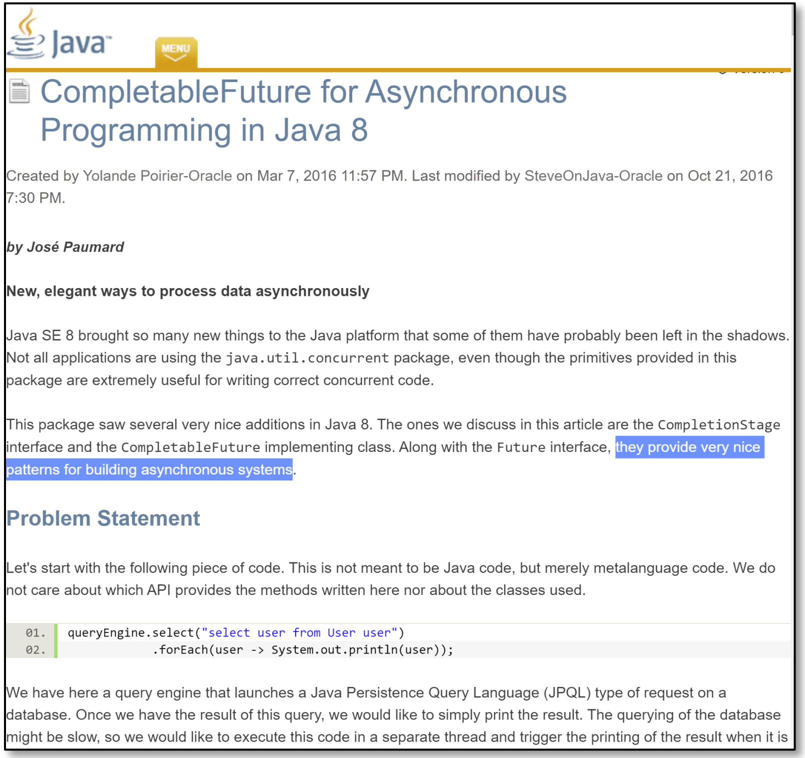


```
Printing 4 results for input file 1 from fastest to slowest
COMPLETABLE_FUTURES_1 executed in 312 msecs
COMPLETABLE_FUTURES_2 executed in 335 msecs
PARALLEL_STREAM executed in 428 msecs
SEQUENTIAL_STREAM executed in 981 msecs

Printing 4 results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_2 executed in 82 msecs
COMPLETABLE_FUTURES_1 executed in 83 msecs
PARALLEL_STREAM executed in 102 msecs
SEQUENTIAL_STREAM executed in 251 msecs
```

Cons of the Java Completable Futures Framework

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program
    - Asynchrony patterns aren't generally well understood by developers



CompletableFuture for Asynchronous Programming in Java 8

Created by Yolande Poirier-Oracle on Mar 7, 2016 11:57 PM. Last modified by SteveOnJava-Oracle on Oct 21, 2016 7:30 PM.

by José Paumard

New, elegant ways to process data asynchronously

Java SE 8 brought so many new things to the Java platform that some of them have probably been left in the shadows. Not all applications are using the `java.util.concurrent` package, even though the primitives provided in this package are extremely useful for writing correct concurrent code.

This package saw several very nice additions in Java 8. The ones we discuss in this article are the `CompletionStage` interface and the `CompletableFuture` implementing class. Along with the `Future` interface, they provide very nice patterns for building asynchronous systems.

Problem Statement

Let's start with the following piece of code. This is not meant to be Java code, but merely metalanguage code. We do not care about which API provides the methods written here nor about the classes used.

```
queryEngine.select("select user from User user")
              .forEach(user -> System.out.println(user));
```

We have here a query engine that launches a Java Persistence Query Language (JPQL) type of request on a database. Once we have the result of this query, we would like to simply print the result. The querying of the database might be slow, so we would like to execute this code in a separate thread and trigger the printing of the result when it is

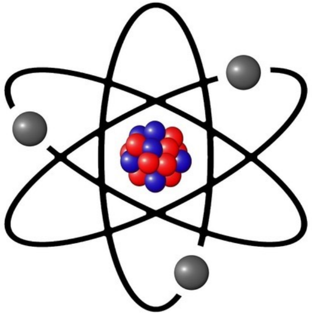
See community.oracle.com/docs/DOC-995305

Cons of the Java Completable Futures Framework

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

Cons of the Java Completable Futures Framework

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program
  - Parallel streams are easier to program, but are less efficient & scalable
    - Use sequential streams for initial development & then trivially make them parallel!



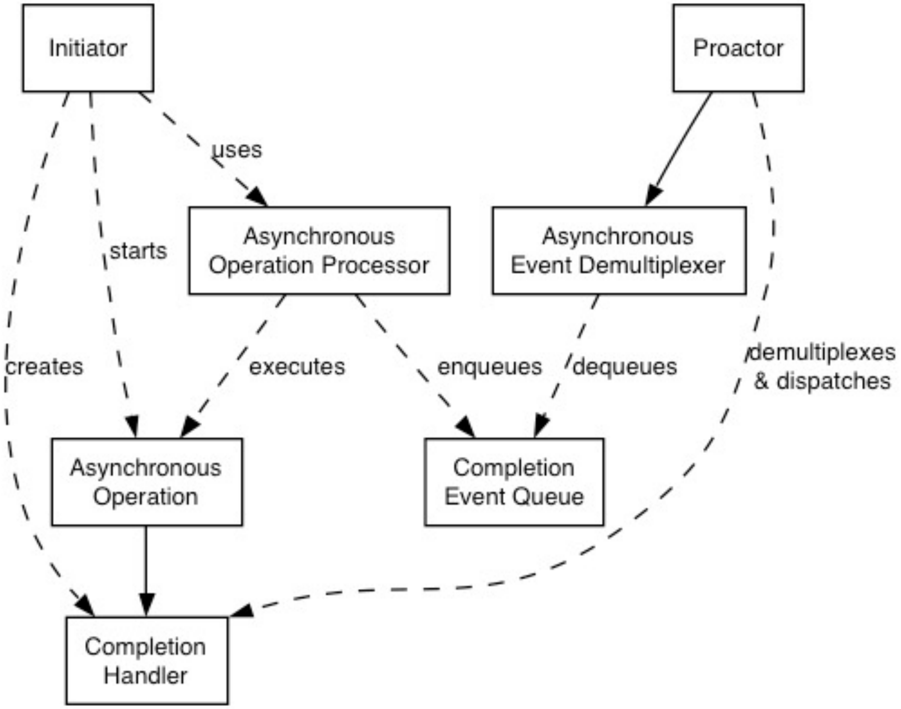
```
List<List<SearchResults>>
processStream() {
  return getInput()
    .stream()
    .map(this::processInput)
    .toList();
}
```

```
List<List<SearchResults>>
processStream() {
  return getInput()
    .parallelStream()
    .map(this::processInput)
    .toList();
}
```

Converting sequential to parallel streams only require minuscule changes!

Cons of the Java Completable Futures Framework

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



End of Evaluating the Cons of the Java Completable Futures Framework