How Java Completable Futures Overcome Limitations of Java Futures

Douglas C. Schmidt

<u>d.schmidt@vanderbilt.edu</u>

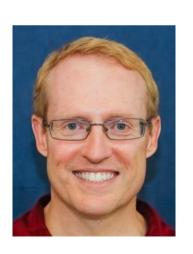
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems

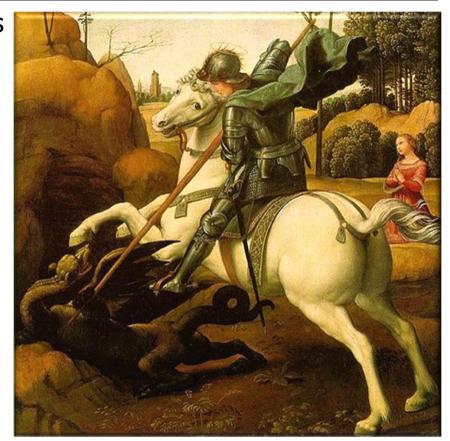
Vanderbilt University Nashville, Tennessee, USA





Learning Objectives in this Part of the Lesson

 Recognize how Java completable futures overcome limitations with Java futures



See earlier lesson on "Overview of the Java Completable Futures Framework"

 The completable futures framework overcomes Java future limitations



- The completable futures framework CompletableFuture<...> future = overcomes lava future limitations
 - *Can* be completed explicitly



```
new Thread (() -> {
  future.complete(...);
}).start();
      After complete() is done
     calls to join() will unblock
System.out.println(future.join());
```

new CompletableFuture<>();

- The completable futures framework CompletableFuture overcomes Java future limitations .supplyAsync(re
 - Can be completed explicitly
 - *Can* be chained fluently to handle async results efficiently & cleanly

```
.supplyAsync(reduceFraction)
```

- .thenAccept(System.out::println);

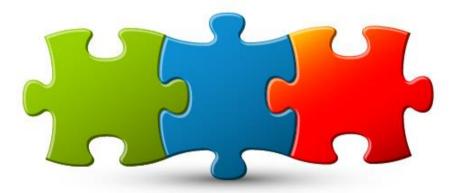
The action of each "completion stage" is triggered when the previous stage's future completes asynchronously

- The completable futures framework CompletableFuture overcomes Java future limitations
 - Can be completed explicitly
 - Can be chained fluently to handle async results efficiently & cleanly
 - Async programming thus looks more like sync programming!

- .supplyAsync(reduceFraction)
 - .thenApply(BigFraction

::toMixedString)

.thenAccept(System.out::println);



- The completable futures framework CompletableFuture<List overcomes Java future limitations
 CompletableFuture<List Overcomes Java future limitations
 - Can be completed explicitly
 - *Can* be chained fluently to handle async results efficiently & cleanly
 - Can be triggered reactively/ efficiently as a collection of futures w/out undue overhead



```
<BigFraction>> futureToList =
  Stream
    .generate(generator)
    .limit(sMAX FRACTIONS)
    .map(reduceFractions)
    .collect(FuturesCollector
             .toFutures());
futureToList
  .thenAccept(printList);
```

Create a single future that will be triggered when a group of other futures all complete

- The completable futures framework CompletableFuture<List overcomes Java future limitations
 <BigFraction>> future
 - Can be completed explicitly
 - *Can* be chained fluently to handle async results efficiently & cleanly
 - Can be triggered reactively/ efficiently as a collection of futures w/out undue overhead



```
<BigFraction>> futureToList =
  Stream
    .generate(generator)
    .limit(sMAX FRACTIONS)
    .map(reduceFractions)
    .collect(FuturesCollector
             .toFutures());
futureToList
  .thenAccept(printList);
```

Print out the results after all async fraction reductions have completed

- The completable futures framework CompletableFuture<List overcomes Java future limitations
 CompletableFuture<List Overcomes Java future limitations
 - Can be completed explicitly
 - *Can* be chained fluently to handle async results efficiently & cleanly
 - Can be triggered reactively/ efficiently as a collection of futures w/out undue overhead



```
<BigFraction>> futureToList =
  Stream
    .generate(generator)
    .limit(sMAX FRACTIONS)
    .map(reduceFractions)
    .collect(FuturesCollector
             .toFutures());
futureToList
  .thenAccept(printList);
```

Java completable futures can also be combined with Java sequential streams

End of How Java Completable Futures Overcome Limitations of Java Futures