Enhancements to 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

- 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		
defaultExecutor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
completeAsync	Supplier<T>	CompletableFuture<T>	Complete *CompletableFuture* asynchronously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	CompletableFuture<T>	Resolves *CompletableFuture* exceptionally with *TimeoutException*, unless it is completed before the specified timeout
completeOnTimeout	T value, long timeout, TimeUnit unit	CompletableFuture<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		
defaultExecutor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
completeAsync	Supplier<T>	CompletableFuture<T>	Complete *CompletableFuture* asynchronously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	CompletableFuture<T>	Resolves *CompletableFuture* exceptionally with *TimeoutException*, unless it is completed before the specified timeout
completeOnTimeout	T value, long timeout, TimeUnit unit	CompletableFuture<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		
defaultExecutor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
completeAsync	Supplier<T>	CompletableFuture<T>	Complete *CompletableFuture* asynchronously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	CompletableFuture<T>	Resolves *CompletableFuture* exceptionally with *TimeoutException*, unless it is completed before the specified timeout
completeOnTimeout	T value, long timeout, TimeUnit unit	CompletableFuture<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		
defaultExecutor	()	Executor	Returns default *Executor* used for methods that don't specify an *Executor*
completeAsync	Supplier<T>	CompletableFuture<T>	Complete *CompletableFuture* asynchronously using value given by the *Supplier*
orTimeout	long timeout, TimeUnit unit	CompletableFuture<T>	Resolves *CompletableFuture* exceptionally with *TimeoutException*, unless it is completed before the specified timeout
completeOnTimeout	T value, long timeout, TimeUnit unit	CompletableFuture<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() & completeOnTimeout()

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
                 .supplyAsync
                   (() -> queryExchangeRateFor("GBP"))
                 .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

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

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
                 .supplyAsync
                   (() -> queryExchangeRateFor("GBP"))
                 .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

Asynchronously find the best price for a flight

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

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
                 .supplyAsync
                   (() -> queryExchangeRateFor("GBP"))
                 .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

Asynchronously find the latest exchange rate

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
                 .supplyAsync
                   (() -> queryExchangeRateFor("GBP"))
                 .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

Select the default rate if call runs longer than 1 seconds

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

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
   .supplyAsync(
      () -> findBestPrice("LDN - NYC"))
   .thenCombine(CompletableFuture
                  .supplyAsync
                    (() -> queryExchangeRateFor("GBP"))
                  .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
   });
```

Combine & convert the search results

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
                .supplyAsync
                  (() -> queryExchangeRateFor("GBP"))
                .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

Throws TimeoutException if the time period elapses

See 4comprehension.com/completablefuture-timeout

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
               .supplyAsync
                 (() -> queryExchangeRateFor("GBP"))
               .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

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

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
                 .supplyAsync
                   (() -> queryExchangeRateFor("GBP"))
                 .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

Print results if async calls finished normally

Applying the orTimeout() & completeOnTimeout() Methods

- This example asynchronously performs web services within a bounded time

```
CompletableFuture
  .supplyAsync(
     () -> findBestPrice("LDN - NYC"))
  .thenCombine(CompletableFuture
                 .supplyAsync
                   (() -> queryExchangeRateFor("GBP"))
                 .completeOnTimeout(DEFAULT_RATE, 1, 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.getMessage()); }
  });
```

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";

 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 string builder holds intermediate results

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());
}
```

Define a pair of 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";
```

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";

  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());
}
```

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

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

End of Enhancements to the Java Completable Futures Framework