Overview of Advanced Java 8
CompletableFuture Features (Part 3)

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 advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
  - Completion stage methods used to chain together actions that perform async result processing & composition
  - Apply completion stage methods to BigFractions

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8
Learning Objectives in this Part of the Lesson

- Understand advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
- Completion stage methods used to chain together actions that perform async result processing & composition
  - Apply completion stage methods to BigFractions
- Know how to handle runtime exceptions
Applying Completable Future Completion Stage Methods
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```java
static void testFractionMultiplications1() {
    ...
    Stream.generate(() -> makeBigFraction(new Random(), false))
        .limit(sMAX_FRACTIONS)
        .map(reduceAndMultiplyFractions)
        .collect(FuturesCollector.toFuture())
        .thenAccept(ex8::sortAndPrintList);
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```java
static void testFractionMultiplications1() {
    ...
    Stream.generate(() -> makeBigFraction(new Random(), false))
        .limit(sMAX_FRACTIONS)
        .map(reduceAndMultiplyFraction)
        .collect(FuturesCollector.toFuture())
        .thenAccept(ex8::sortAndPrintList);
}
```
We show key completion stage methods via the `testFractionMultiplications1()` method that multiplies big fractions using a stream of `CompletableFuture` objects.

```java
BigFraction makeBigFraction(Random random, boolean reduced) {
    BigInteger numerator =
        new BigInteger(150000, random);

    BigInteger denominator =
        numerator.divide(BigInteger.
            .valueOf(random.nextInt(10) + 1));

    return BigFraction.valueOf(numerator,
                                denominator,
                                reduced);
}
```

Factory method that creates a large & random big fraction.
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures.

```java
BigFraction makeBigFraction(Random random, boolean reduced) {
    BigInteger numerator =
        new BigInteger(150000, random);

    BigInteger denominator =
        numerator.divide(BigInteger.valueOf(random.nextInt(10) + 1));

    return BigFraction.valueOf(numerator,
        denominator,
        reduced);
}
```

Make a random numerator uniformly distributed over range 0 to \(2^{150000} - 1\)

See docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#BigInteger
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```java
BigFraction makeBigFraction(Random random, boolean reduced) {
    BigInteger numerator =
        new BigInteger(150000, random);

    BigInteger denominator =
        numerator.divide(BigInteger.valueOf(random.nextInt(10) + 1));

    return BigFraction.valueOf(numerator,
                               denominator,
                               reduced);
}
```

Make a denominator by dividing the numerator by random # between 1 & 10
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```java
BigFraction makeBigFraction(Random random, boolean reduced) {
    BigInteger numerator =
        new BigInteger(150000, random);

    BigInteger denominator =
        numerator.divide(BigInteger.valueOf(random.nextInt(10) + 1));

    return BigFraction.valueOf(numerator, denominator, reduced);
}
```

Return a BigFraction w/the numerator & denominator
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures:

```java
static void testFractionMultiplications1() {
    ...
    Stream.generate(() -> makeBigFraction(new Random(), false))
        .limit(sMAX_FRACTIONS)
        .map(reduceAndMultiplyFraction)
        .collect(FuturesCollector.toFuture())
        .thenAccept(ex8::sortAndPrintList);
}
```

Reduce & multiply all these big fractions asynchronously.
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures.

```java
static void testFractionMultiplications1() {
    Function<BigFraction, CompletableFuture<BigFraction>>
        reduceAndMultiplyFraction = unreducedFrac ->
        CompletableFuture.supplyAsync(() -> BigFraction.reduce(unreducedFrac)).
            thenCompose(reducedFrac -> CompletableFuture.supplyAsync(() ->
                reducedFrac.multiply(sBigFraction)));
}
```

Lambda that asynchronously reduces/multiplies a big fraction.
ApplyingCompletableFutureCompletionStageMethods

- We show key completion stage methods via the `testFractionMultiplications1()` method that multiplies big fractions using a stream of CompletableFutures

```java
static void testFractionMultiplications1() {
    Function<BigFraction, CompletableFuture<BigFraction>>
    reduceAndMultiplyFraction = unreducedFrac ->
    CompletableFuture
    .supplyAsync(() -> BigFraction.reduce(unreducedFrac))
    .thenCompose(reducedFrac -> CompletableFuture
    .supplyAsync(() -> reducedFrac
    .multiply(sBigFraction)
    .multiply(sBigFraction)));
```

Asynchronously reduce a big fraction
Applying Completetable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletetableFutures.

```java
static void testFractionMultiplications1() {
  Function<BigFraction, CompletableFuture<BigFraction>>
  reduceAndMultiplyFraction = unreducedFrac ->
  CompletableFuture.supplyAsync(() -> BigFraction.reduce(unreducedFrac))
    .thenCompose(reducedFrac -> CompletableFuture.supplyAsync(() -> reducedFrac.multiply(sBigFraction)).

  Asynchronously multiply big fractions
```

...
Applying CompletableFuture Completion Stage Methods

- We show key completion stage methods via the `testFractionMultiplications1()` method that multiplies big fractions using a stream of CompletableFutures.

```java
static void testFractionMultiplications1() {
    Function<BigFraction, CompletableFuture<BigFraction>>
    reduceAndMultiplyFraction = unreducedFrac ->
    CompletableFuture.supplyAsync(() -> BigFraction.reduce(unreducedFrac))
        .thenCompose(reducedFrac -> CompletableFuture.supplyAsync(() ->
            reducedFrac.multiply(sBigFraction)));
```

`thenCompose()` acts like `flatMap()` to ensure one level of `CompletableFuture` nesting.
Applying CompletableFuture Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFuture.

```java
static void testFractionMultiplications2() {
    Function<BigFraction, CompletableFuture<BigFraction>>
    reduceAndMultiplyFraction = unreducedFrac ->
    CompletableFuture.supplyAsync(() ->
        BigFraction.reduce(unreducedFrac)).thenApplyAsync(reducedFrac ->
        reducedFrac.multiply(sBigFraction));

    // ... Asynchronously multiply big fractions
}
```

thenApplyAsync() provides a way to avoid calling supplyAsync() again.
We show key completion stage methods via the `testFractionMultiplications1()` method that multiplies big fractions using a stream of CompletableFuture:

```java
static void testFractionMultiplications1() {
    ...
    Stream.generate(() -> makeBigFraction(new Random(), false))
        .limit(sMAX_FRACTIONS)
        .map(reduceAndMultiplyFraction)
        .collect(FuturesCollector.toFuture())
        .thenAccept(ex8::sortAndPrintList);
}
```

Outputs a stream of completable futures to async operations on big fractions.
We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures.

```java
static void testFractionMultiplications1() {
    ...
    Stream.generate(() -> makeBigFraction(new Random(), false))
        .limit(sMAX_FRACTIONS)
        .map(reduceAndMultiplyFraction)
        .collect(FuturesCollector.toFuture())
        .thenAccept(ex8::sortAndPrintList);
}
```

Return a future to a list of big fractions being reduced & multiplied asynchronously.

FuturesCollector is a non-concurrent collector covered in Part 4 of this lesson.
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```java
static void testFractionMultiplications1() {
    ...
    Stream.generate(() -> makeBigFraction(new Random(), false))
        .limit(sMAX_FRACTIONS)
        .map(reduceAndMultiplyFraction)
        .collect(FuturesCollector.toFuture())
        .thenAccept(ex8::sortAndPrintList);
}
```

Sort & print results when all async computations complete
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```java
static void sortAndPrintList(List<BigFraction> list) {
    CompletableFuture<List<BigFraction>> quickSortF = CompletableFuture.supplyAsync(() -> quickSort(list));
    CompletableFuture<List<BigFraction>> mergeSortF = CompletableFuture.supplyAsync(() -> mergeSort(list));

    quickSortF.acceptEither(mergeSortF, sortedList -> sortedList.forEach(frac -> display(frac.toMixedString())));
}
```

Sorts & prints a list of reduced fractions
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures.

```java
static void sortAndPrintList(List<BigFraction> list) {
    CompletableFuture<List<BigFraction>> quickSortF = CompletableFuture.supplyAsync(() -> quickSort(list));
    CompletableFuture<List<BigFraction>> mergeSortF = CompletableFuture.supplyAsync(() -> mergeSort(list));
    quickSortF.acceptEither(mergeSortF, sortedList -> sortedList.forEach(frac -> display(frac.toMixedString())));
}...
```

Asynchronously apply quick sort & merge sort!
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures.

```java
static void sortAndPrintList(List<BigFraction> list) {

    CompletableFuture<List<BigFraction>> quickSortF =
    CompletableFuture.supplyAsync(() -> quickSort(list));

    CompletableFuture<List<BigFraction>> mergeSortF =
    CompletableFuture.supplyAsync(() -> mergeSort(list));

    quickSortF.acceptEither(mergeSortF, sortedList ->
    sortedList.forEach(frac -> display(frac.toMixedString()));
}
```

Select whichever result finishes first.
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures.

```java
static void sortAndPrintList(List<BigFraction> list) {

    CompletableFuture< List< BigFraction >> quickSortF =
    CompletableFuture.supplyAsync(() -> quickSort(list));

    CompletableFuture< List< BigFraction >> mergeSortF =
    CompletableFuture.supplyAsync(() -> mergeSort(list));

    quickSortF.acceptEither(mergeSortF, sortedList ->
                         sortedList.forEach(frac -> display(frac.toMixedString())));

    ...}
```
Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures.

```java
static void sortAndPrintList(List<BigFraction> list) {
    CompletableFuture<List<BigFraction>> quickSortF =
        CompletableFuture.supplyAsync(() -> quickSort(list));

    CompletableFuture<List<BigFraction>> mergeSortF =
        CompletableFuture.supplyAsync(() -> mergeSort(list));

    quickSortF.acceptEither(mergeSortF, sortedList ->
        sortedList.forEach(frac -> display(frac.toMixedString())));
}
```

Otherwise, the action runs in the thread in which the previous stage ran.
Applying CompletableFuture Completion Stage Methods

• We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void sortAndPrintList(List<BigFraction> list) {

    CompletableFuture<List<BigFraction>> quickSortF =
        CompletableFuture.supplyAsync(() -> quickSort(list));

    CompletableFuture<List<BigFraction>> mergeSortF =
        CompletableFuture.supplyAsync(() -> mergeSort(list));

    quickSortF.acceptEither(mergeSortF, sortedList ->
        sortedList.forEach(frac -> display(frac.toMixedString())));
    ...
```

acceptEither() does not cancel the second future after the first one completes
Handling Runtime Exceptions in Completion Stages
Handling Runtime Exceptions in Completion Stages

- Completion stage methods handle runtime exceptions

### Handling Runtime Exceptions in Completion Stages

- Completion stage methods handle runtime exceptions

<table>
<thead>
<tr>
<th>Methods</th>
<th>Params</th>
<th>Returns</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>when Complete</td>
<td>Bi Consumer</td>
<td>Completable Future with result of earlier stage or throws exception</td>
<td>Handle outcome of a stage, whether a result value or an exception</td>
</tr>
<tr>
<td>(Async)</td>
<td></td>
<td>Completable Future with result of earlier stage or throws exception</td>
<td></td>
</tr>
<tr>
<td>handle (Async)</td>
<td>Bi Function</td>
<td>Completable Future with result of BiFunction</td>
<td>Handle outcome of a stage &amp; return new value</td>
</tr>
<tr>
<td>exceptionally</td>
<td>Function &lt;Void&gt;</td>
<td>Completable Future</td>
<td>When exception occurs, replace exception with result value</td>
</tr>
</tbody>
</table>

See [community.oracle.com/docs/DOC-995305](community.oracle.com/docs/DOC-995305)
Handling Runtime Exceptions in Completion Stages

• This example shows three ways to handle exceptions w/completable futures

```java
CompletableFuture.supplyAsync(() ->
    BigFraction.valueOf(100, denominator))
```

...
Handling Runtime Exceptions in Completion Stages

- Using the handle() method to handle exceptional or normal completions

```java
CompletableFuture.supplyAsync(() ->
    BigFraction.valueOf(100, denominator))
    .handle((fraction, ex) -> {
        if (fraction == null)
            return BigFraction.ZERO;
        else
            return fraction.multiply(sBigReducedFraction);
    })
    .thenAccept(fraction ->
        System.out.println(fraction.toMixedString()));
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#handle](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#handle)
• Using the handle() method to handle exceptional or normal completions

CompletableFuture
  .supplyAsync(() ->
      BigFraction.valueOf(100, denominator))

  .handle((fraction, ex) -> {
    if (fraction == null)
      return BigFraction.ZERO;
    else
      return fraction.multiply(sBigReducedFraction);
  })

  .thenAccept(fraction ->
      System.out.println(fraction.toMixedString()));

The exception path
Using the handle() method to handle exceptional or normal completions

CompletableFuture
    .supplyAsync(() ->
        BigFraction.valueOf(100, denominator))
    .handle((fraction, ex) -> {
        if (fraction == null)
            return BigFraction.ZERO;
        else
            return fraction.multiply(sBigReducedFraction);
    })
    .thenAccept(fraction ->
        System.out.println(fraction.toMixedString()));
Handling Runtime Exceptions in Completion Stages

- Using the `handle()` method to handle exceptional or normal completions

```java
CompletableFuture.supplyAsync(() ->
    BigFraction.valueOf(100, denominator))
    .handle((fraction, ex) -> {
        if (fraction == null)
            return BigFraction.ZERO;
        else
            return fraction.multiply(sBigReducedFraction);
    })
    .thenAccept(fraction ->
        System.out.println(fraction.toMixedString()));
```

`handle()` must return a value (& can thus change the return value)
Handling Runtime Exceptions in Completion Stages

- Using the exceptionally() method to handle exceptional or normal completions

```java
CompletableFuture.supplyAsync(() ->
    BigFraction.valueOf(100, denominator))
    .thenApply(fraction ->
        fraction.multiply(sBigReducedFraction))
    .exceptionally(ex -> BigFraction.ZERO)
    .thenAccept(fraction ->
        System.out.println(fraction.toMixedString()));
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#exceptionally](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#exceptionally)
Handling Runtime Exceptions in Completion Stages

- Using the exceptionally() method to handle exceptional or normal completions

```java
CompletableFuture
    .supplyAsync(() ->
        BigFraction.valueOf(100, denominator))
    .thenApply(fraction ->
        fraction.multiply(sBigReducedFraction))
    .exceptionally(ex -> BigFraction.ZERO)
    .thenAccept(fraction ->
        System.out.println(fraction.toMixedString()));
```

- `exceptionally()` is akin to a catch in a Java try/catch block, i.e., control xfers to it
Handling Runtime Exceptions in Completion Stages

- Using the `whenComplete()` method to perform a exceptional or normal action

```java
CompletableFuture.supplyAsync(() -> 
    BigFraction.valueOf(100, denominator))
    .thenApply(fraction -> 
        fraction.multiply(sBigReducedFraction))
    .whenComplete((fraction, ex) -> {
        if (fraction != null)
            System.out.println(fraction.toMixedString());
        else
            System.out.println(ex.getMessage());
    });
```

*Called under both normal & exception conditions*

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#whenComplete](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#whenComplete)
Handling Runtime Exceptions in Completion Stages

- Using the `whenComplete()` method to perform a exceptional or normal action

```java
CompletableFuture
    .supplyAsync(() ->
        BigFraction.valueOf(100, denominator))

    .thenApply(fraction ->
        fraction.multiply(sBigReducedFraction))

    .whenComplete((fraction, ex) -> {
        if (fraction != null)
            System.out.println(fraction.toMixedString());
        else
            System.out.println(ex.getMessage());
    });
```

Handle the normal case
Handling Runtime Exceptions in Completion Stages

- Using the whenComplete() method to perform a exceptional or normal action

```java
CompletableFuture
    .supplyAsync(() ->
        BigFraction.valueOf(100, denominator))

    .thenApply(fraction ->
        fraction.multiply(sBigReducedFraction))

    .whenComplete((fraction, ex) -> {
        if (fraction != null)
            System.out.println(fraction.toMixedString());
        else // ex != null
            System.out.println(ex.getMessage());
    });
```

Handle the exceptional case
Handling Runtime Exceptions in Completion Stages

- Using the whenComplete() method to perform a exceptional or normal action

```java
CompletableFuture.supplyAsync(() -> 
    BigFraction.valueOf(100, denominator))
    .thenApply(fraction -> 
        fraction.multiply(sBigReducedFraction))
    .whenComplete((fraction, ex) -> {
        if (fraction != null) 
            System.out.println(fraction.toMixedString());
        else // ex != null
            System.out.println(ex.getMessage());
    });
```

whenComplete() is like Java 8 streams peek(): it has a side-effect & doesn’t change the return value

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#peek](docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#peek)
End of Overview of Advanced Java 8 Completable Future Features (Part 3)