Advanced Java CompletableFuture Features: Handling Runtime Exceptions

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

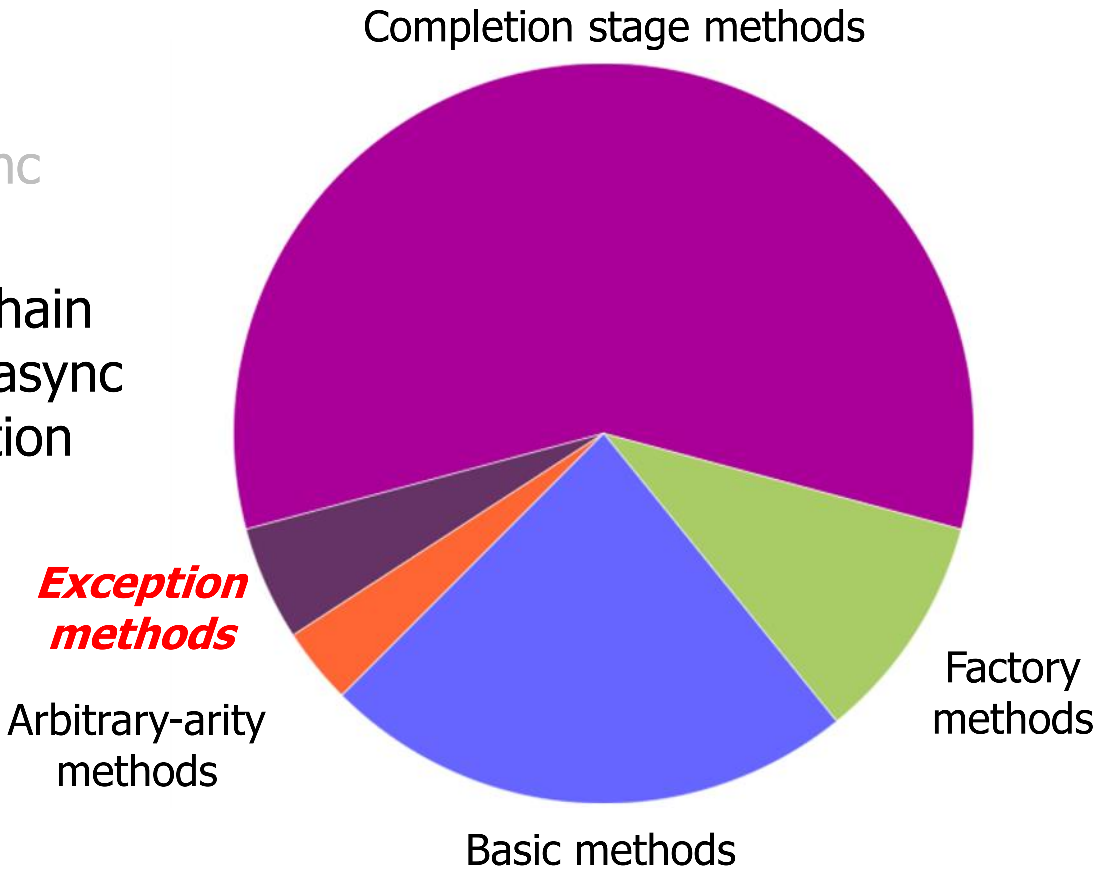
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

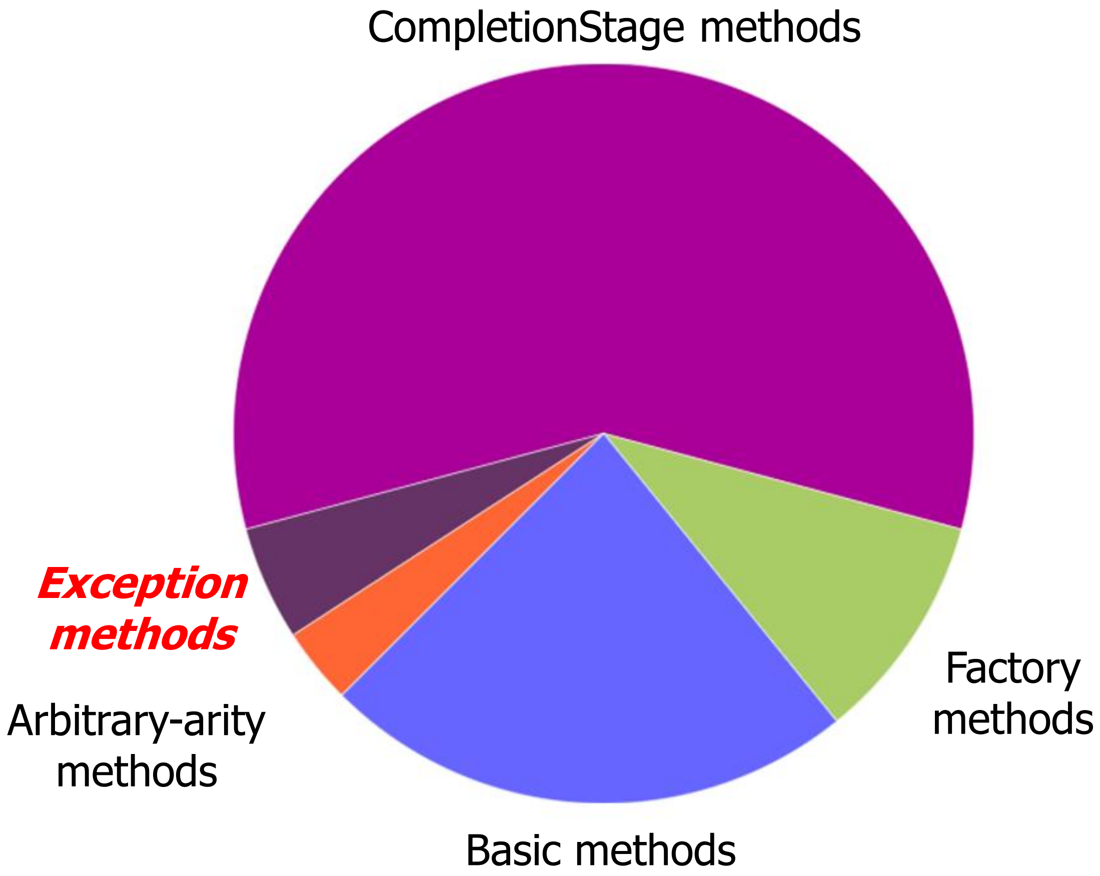
- Understand advanced features of completable futures, e.g.
  - Factory methods initiate async computations
  - Completion stage methods chain together actions to perform async result processing & composition
    - Method grouping
    - Single stage methods
    - Two stage methods (and)
    - Two stage methods (or)
    - Apply these methods
    - Handle runtime exceptions



Handling Runtime Exceptions in Completion Stages

Handling Runtime Exceptions in Completion Stages

- Completion stage methods handle runtime exceptions



See www.jesperdj.com/2015/09/26/the-future-is-completable-in-java-8

Handling Runtime Exceptions in Completion Stages

- Completion stage methods handle runtime exceptions

Methods	Params	Returns	Behavior
whenComplete (Async)	BiConsumer	CompletableFuture with result of earlier stage or throws exception	Handle outcome of a stage, whether a result value or an exception
handle (Async)	BiFunction	CompletableFuture with result of BiFunction	Handle outcome of a stage & return new value
exceptionally	Function	CompletableFuture<T>	When exception occurs, replace exception with result value

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

Handling Runtime Exceptions in Completion Stages

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

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

   ...
```

An exception will occur if denominator param is 0!

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

Handling Runtime Exceptions in Completion Stages

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

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

  ...
```

An unhandled exception will terminate a program!

See rollbar.com/guides/java-throwing-exceptions

Handling Runtime Exceptions in Completion Stages

- 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()));
```

Handle outcome of the previous stage (always called, regardless of whether exception's thrown)

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

Handling Runtime Exceptions in Completion Stages

- 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()));
```

These values are mutually exclusive

Handling Runtime Exceptions in Completion Stages

- 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

Handling Runtime Exceptions in Completion Stages

- 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 "normal" path

Handling Runtime Exceptions in Completion Stages

- 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()));
```

handle() must return a value (& can thus change the return value)

Handling Runtime Exceptions in Completion Stages

- 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()));
```

Display result as a mixed fraction

Handling Runtime Exceptions in Completion Stages

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

```
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

Handling Runtime Exceptions in Completion Stages

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

An exception occurs if denominator is 0!

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

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

  .exceptionally(ex -> BigFraction.ZERO)


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

Handling Runtime Exceptions in Completion Stages

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

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

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

  .exceptionally(ex -> BigFraction.ZERO)

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

Handle case where denominator != 0 (skipped if exception is thrown)

Handling Runtime Exceptions in Completion Stages

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

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

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

   .exceptionally(ex -> BigFraction.ZERO)

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

Handle case where denominator == 0 & exception is thrown (otherwise skipped)

exceptionally() is akin to catch() in a Java try/catch block, i.e., control xfers to it

Handling Runtime Exceptions in Completion Stages

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

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

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

  .exceptionally(ex -> BigFraction.ZERO)


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

Convert an exception to a 0 result

Handling Runtime Exceptions in Completion Stages

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

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

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

   .exceptionally(ex -> BigFraction.ZERO)

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

Display result as a mixed fraction

Handling Runtime Exceptions in Completion Stages

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

```
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

Handling Runtime Exceptions in Completion Stages

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

```
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());
    });
```

These values are mutually exclusive

Handling Runtime Exceptions in Completion Stages

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

```
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

```
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

```
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 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

End of Advanced Java CompletableFuture Features: Handling Runtime Exceptions