Basic Java Completable Future Features

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the basic features in the Java completable futures framework

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage<T>, Future<T>

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.

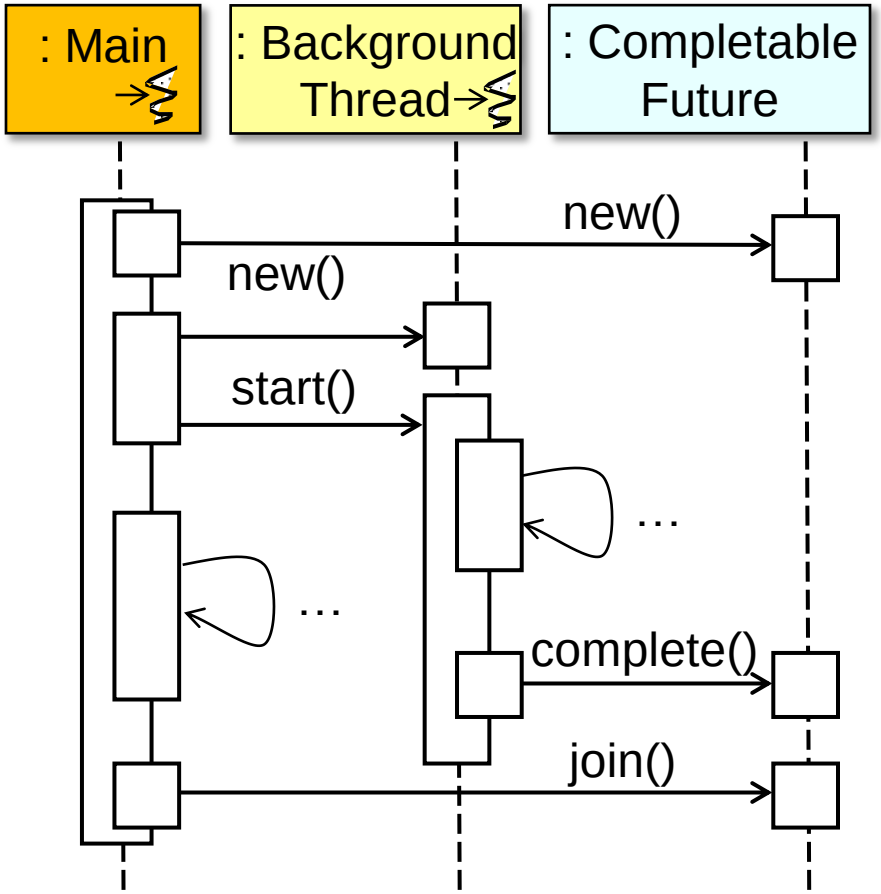
When two or more threads attempt to complete, completeExceptionally, or cancel a CompletableFuture, only one of them succeeds.

In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

Basic Completable
Future Features

Basic Completable Future Features

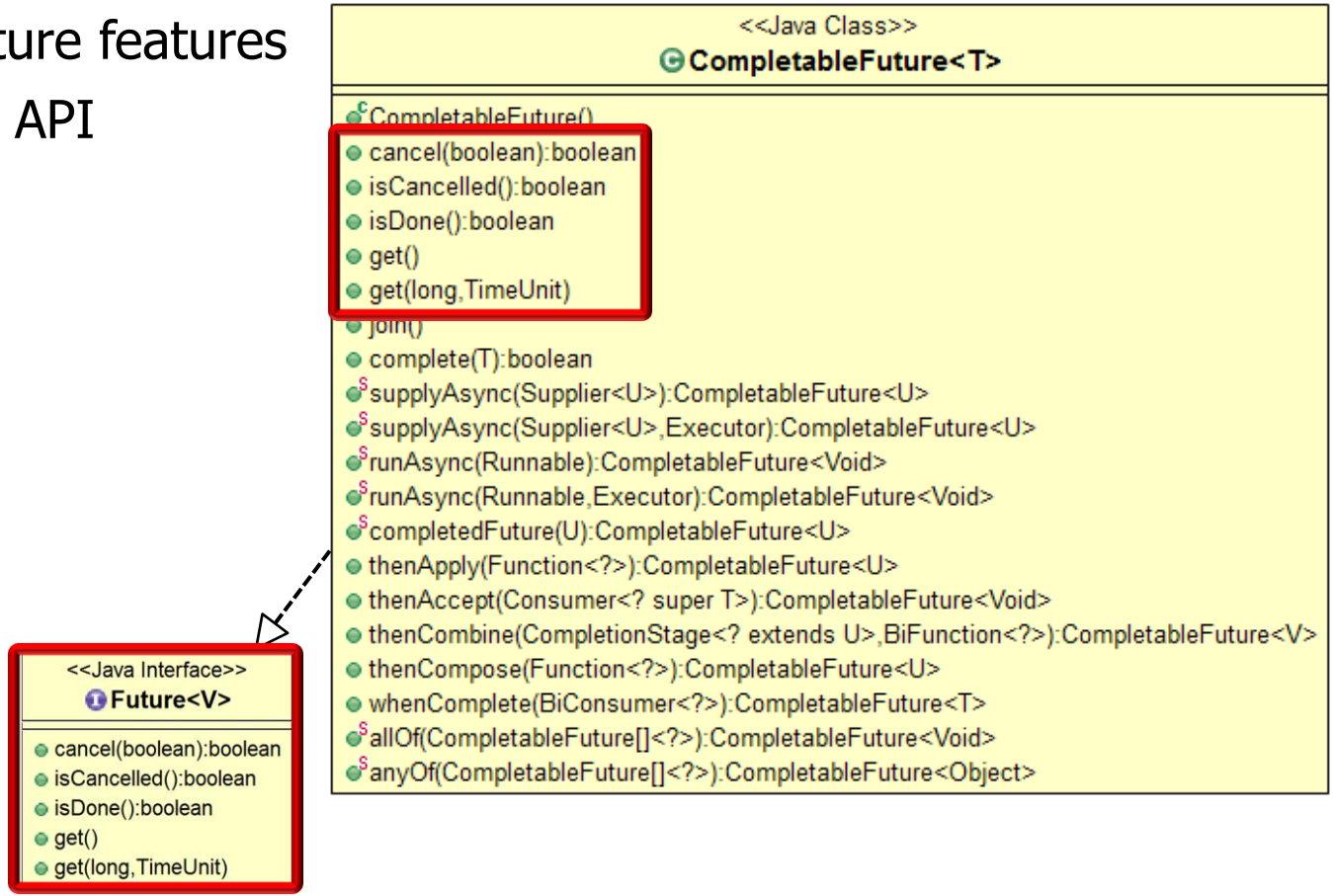
- Basic completable future features



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

Basic Completable Future Features

- Basic completable future features
  - Support the Future API



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
    - Can (time-) block and poll

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

ForkJoinTask<BigFraction> f =
  commonPool().submit(() -> {
    BigFraction bf1 =
      new BigFraction(f1);
    BigFraction bf2 =
      new BigFraction(f2);
    return bf1.multiply(bf2);
  });
...
BigFraction result = f.get();
// f.get(10, MILLISECONDS);
// f.get(0, 0);
```

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

Basic Completable Future Features

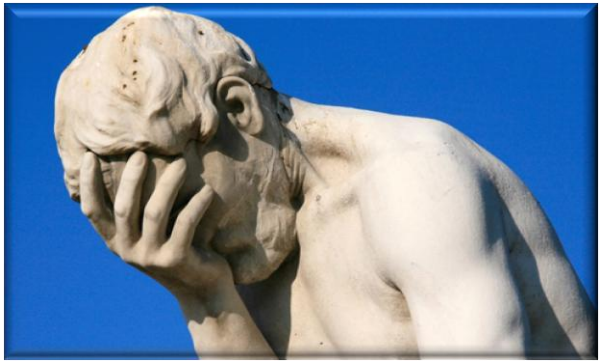
- Basic completable future features
  - Support the Future API
    - Can (time-) block and poll
    - Can be cancelled and tested if cancelled/done

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

ForkJoinTask<BigFraction> f =
   commonPool().submit(() -> {
     BigFraction bf1 =
       new BigFraction(f1);
     BigFraction bf2 =
       new BigFraction(f2);
     return bf1.multiply(bf2);
   });
...
if (!(f.isDone()
       || !f.isCancelled()))
   f.cancel();
```

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

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
    - Can (time-) block and poll
    - Can be cancelled and tested if cancelled/done
      - cancel() doesn't interrupt the computation by default

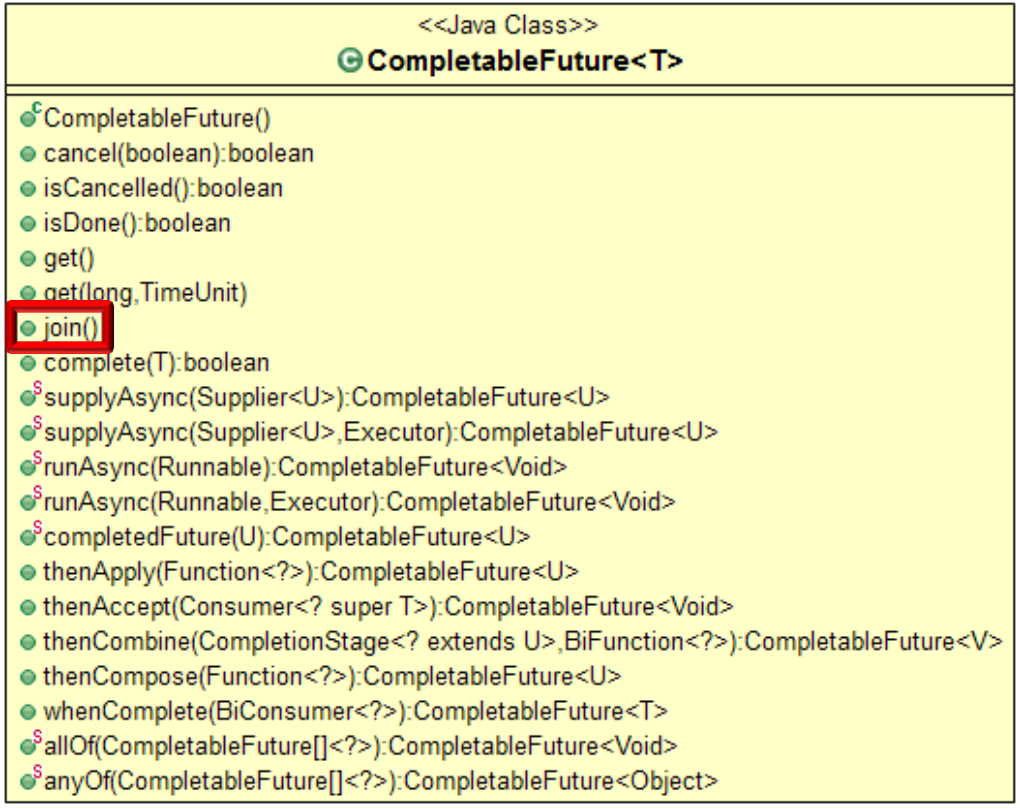


```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

ForkJoinTask<BigFraction> f =
  commonPool().submit(() -> {
    BigFraction bf1 =
      new BigFraction(f1);
    BigFraction bf2 =
      new BigFraction(f2);
    return bf1.multiply(bf2);
  });
...
if (!(f.isDone()
      || !f.isCancelled()))
  f.cancel();
```

See www.nurkiewicz.com/2015/03/completablefuture-cant-be-interrupted.html

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method

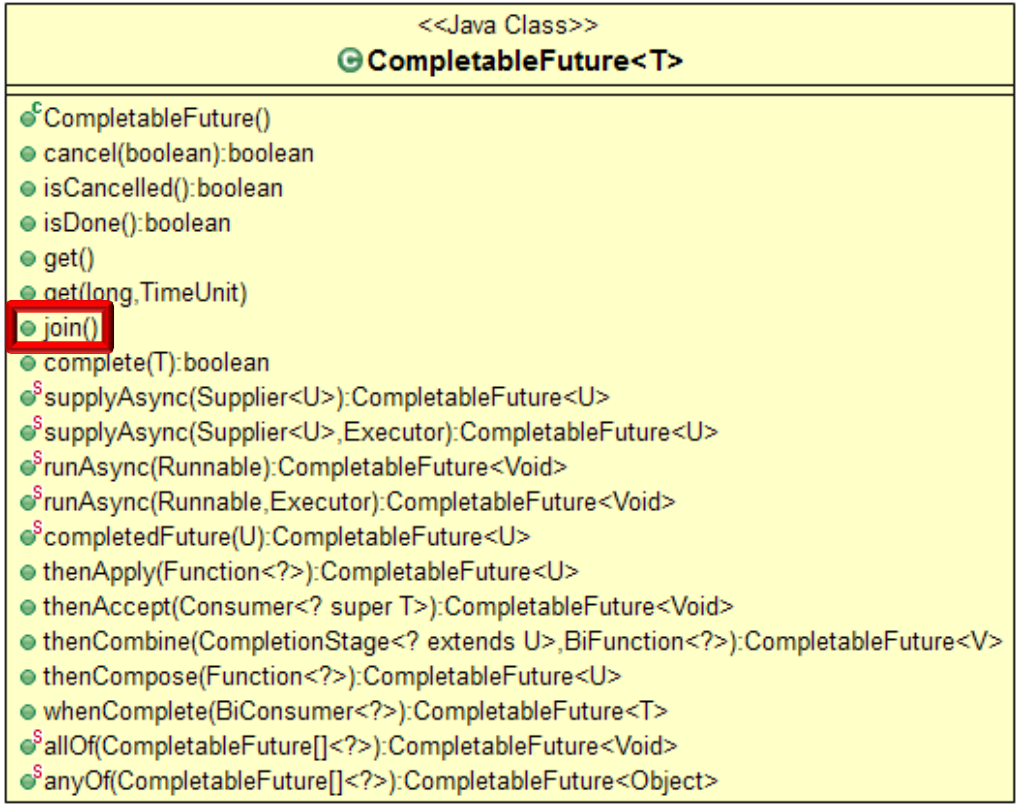


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#join

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
    - Behaves like get() *without* using checked exceptions

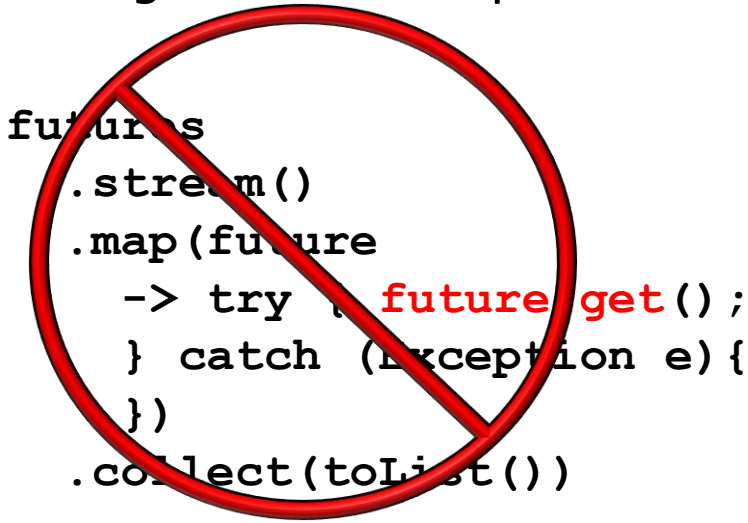
```
futures
   .stream()
   .map(CompletableFuture
        ::join)
   .collect(toList())
```



Completable future::join can be used as a method reference in a Java stream.

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
    - Behaves like get() *without* using checked exceptions



```
futures
  .stream()
  .map(future
    -> try { future.get();
    } catch (Exception e){
    })
  .collect(toList())
```

<<Java Class>>

CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Mixing checked exceptions and Java streams is ugly.

Basic Completable Future Features

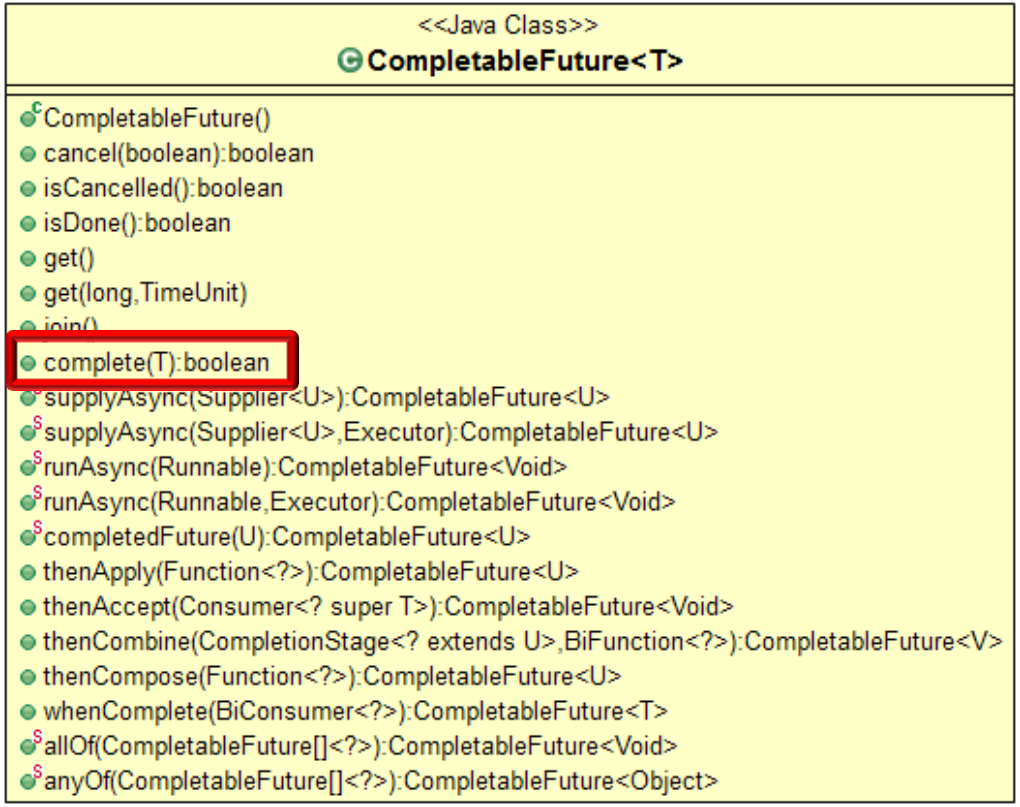
- Basic completable future features
  - Support the Future API
  - Define a join() method
    - Behaves like get() *without* using checked exceptions
    - There is no timed version of join()

<<Java Class>>
CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly



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

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly
    - Sets result returned by get()/join() to a given value

```
CompletableFuture<...> future =
  new CompletableFuture<>();

new Thread (() -> {
  ...
  future.complete(...);
}).start();

...
System.out.println(future.join());
```

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly
    - Sets result returned by get()/join() to a given value

```
CompletableFuture<...> future =
  new CompletableFuture<>();

new Thread (() -> {
  ...
  future.complete(...);
}).start();

...
System.out.println(future.join());
```

Create an incomplete future

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

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly
    - Sets result returned by get()/join() to a given value

```
CompletableFuture<...> future =
   new CompletableFuture<>();

new Thread (() -> {
   ...
   future.complete(...);
}).start();

...
System.out.println(future.join());
```

Create/start a new thread that runs concurrently with the main thread

See docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly
    - Sets result returned by get()/join() to a given value

```
CompletableFuture<...> future =
   new CompletableFuture<>();

new Thread (() -> {
   ...
   future.complete(...);
}).start();

...
System.out.println(future.join());
```

After complete() is done calls to join() will unblock.

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

Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly
    - Sets result returned by get()/join() to a given value

```
CompletableFuture<...> future =
  new CompletableFuture<>();

final CompletableFuture<Long> zero
  = CompletableFuture
      .completedFuture(0L);

new Thread (() -> {
  ...
  future.complete(zero.join());
}).start();

...
System.out.println(future.join());
```

A completable future can be initialized to a value/constant.

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

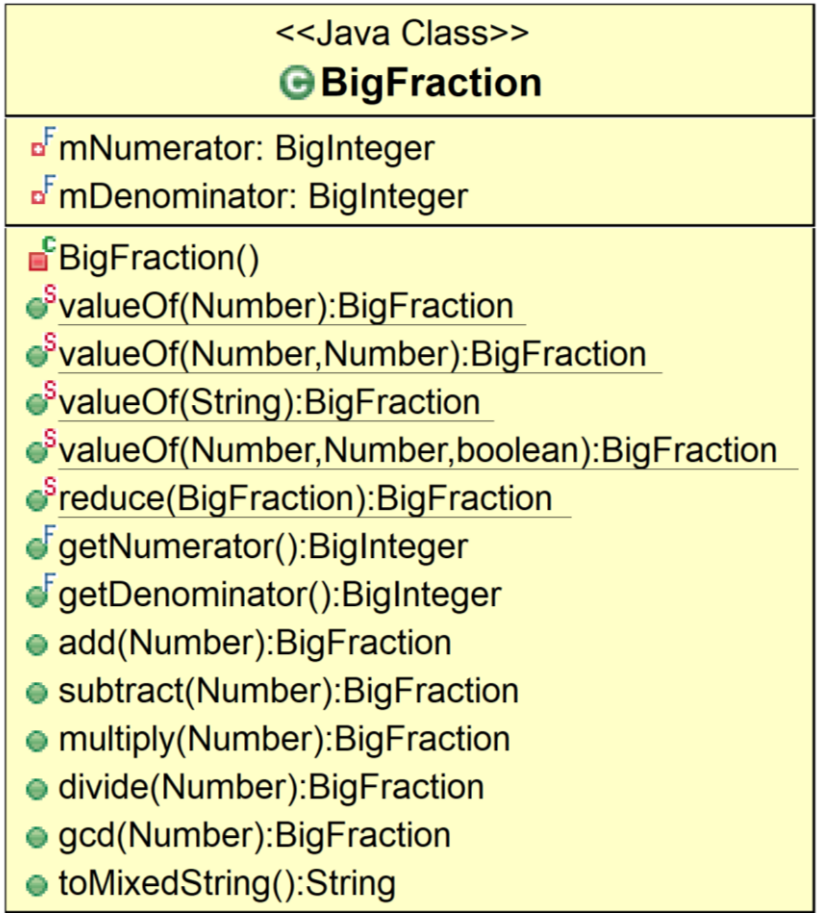
The End

Applying Basic Java Completable Future Features

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand the basic features in the Java completable futures framework
- Know how to apply these basic features to operate on big fractions



See earlier lesson on "*Programming with Java Futures*"

Learning Objectives in This Part of the Lesson

- Understand the basic features in the Java completable futures framework
- Know how to apply these basic features to operate on big fractions
- Recognize limitations with these basic features

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

`CompletionStage<T>, Future<T>`

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

A `Future` that may be explicitly completed (setting its value and status), and may be used as a `CompletionStage`, supporting dependent functions and actions that trigger upon its completion.

When two or more threads attempt to `complete`, `completeExceptionally`, or `cancel` a CompletableFuture, only one of them succeeds.

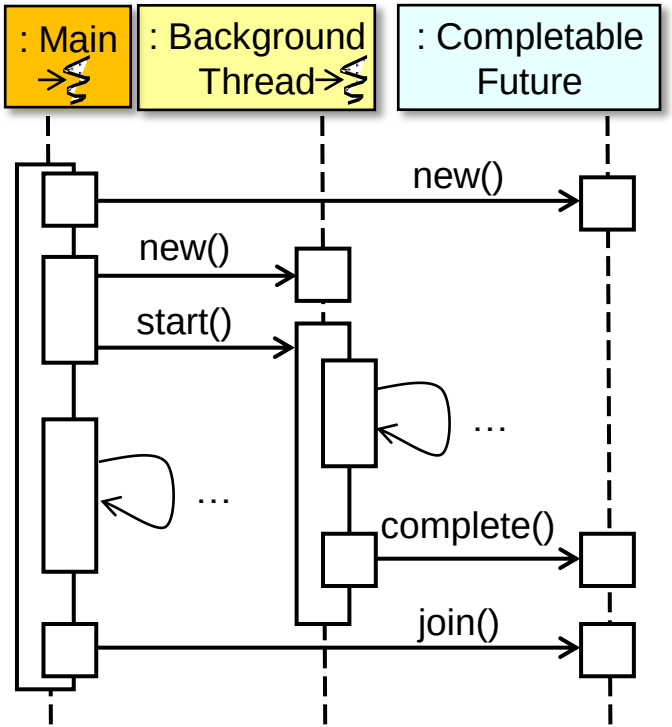
In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface `CompletionStage` with the following policies:

Applying Basic Completable Future Features

Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```



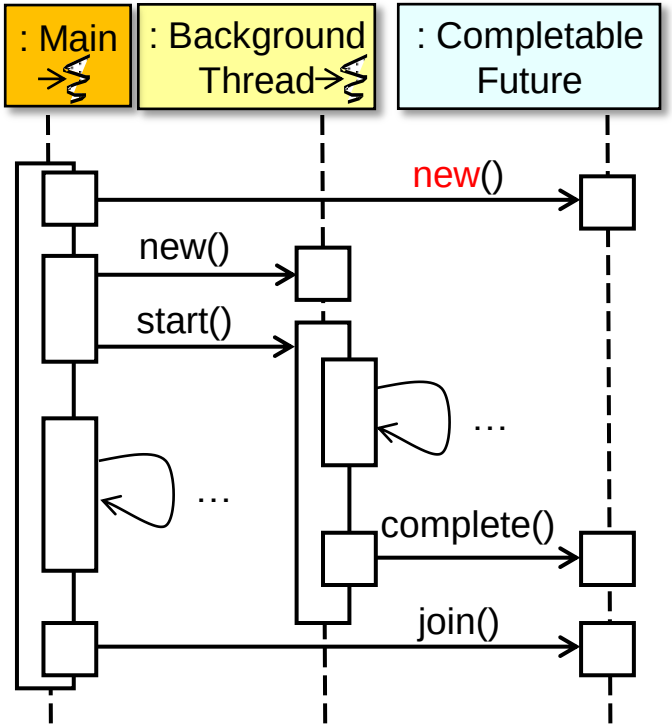
See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```

Make "empty" future

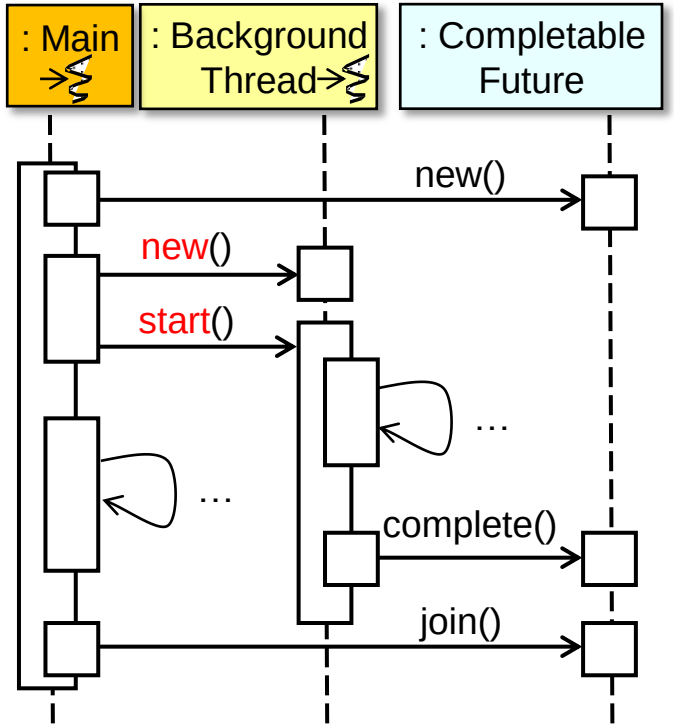


Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```

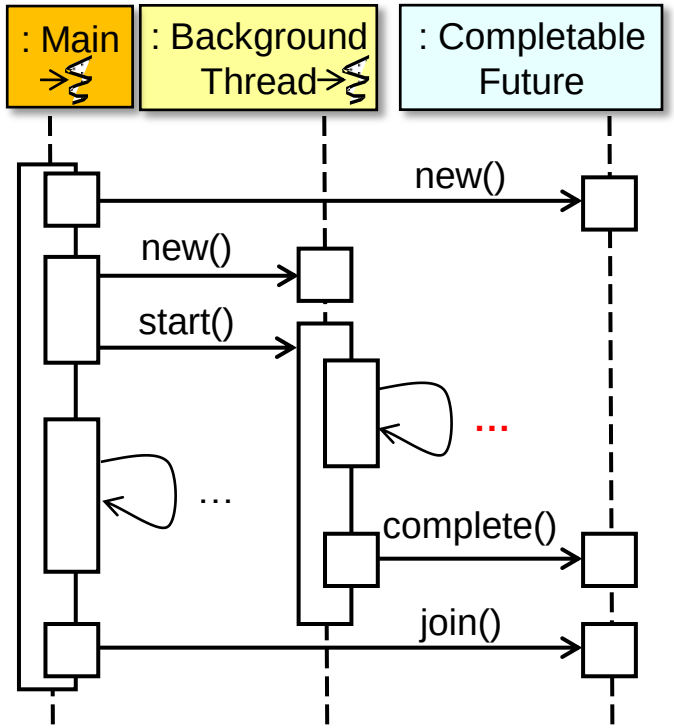
Start computation in a background thread



Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```



The computation multiplies BigFractions (via BigIntegers)

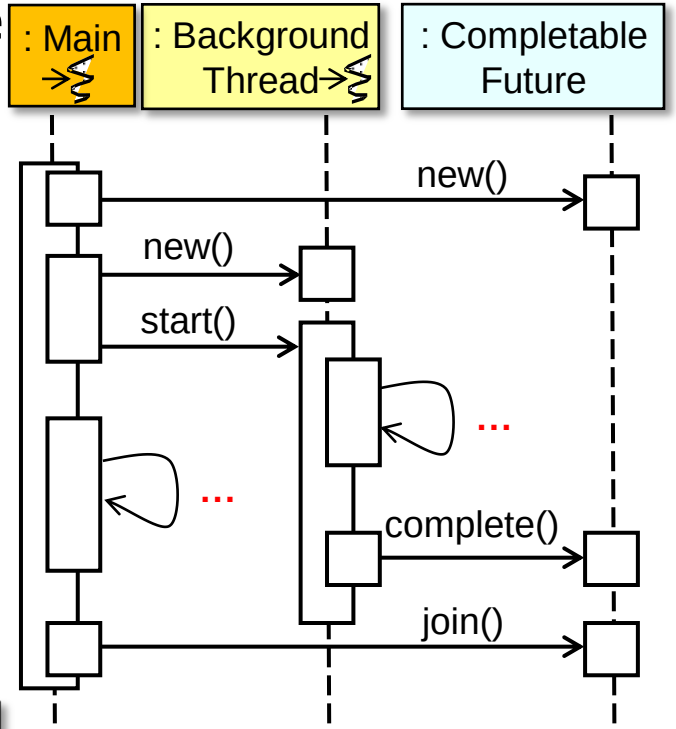
See docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html

Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```

These computations run concurrently.

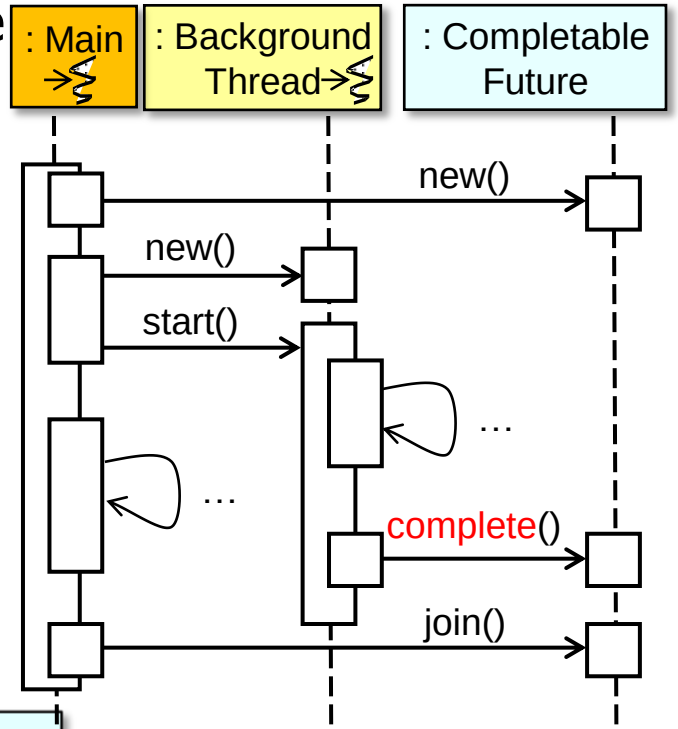


Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```

Explicitly complete the future with result

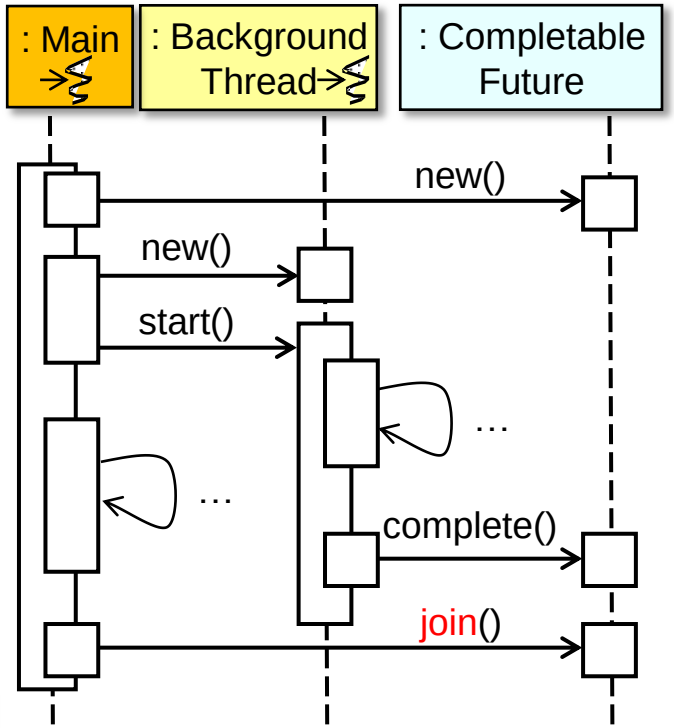


Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```

join() blocks until result is computed.

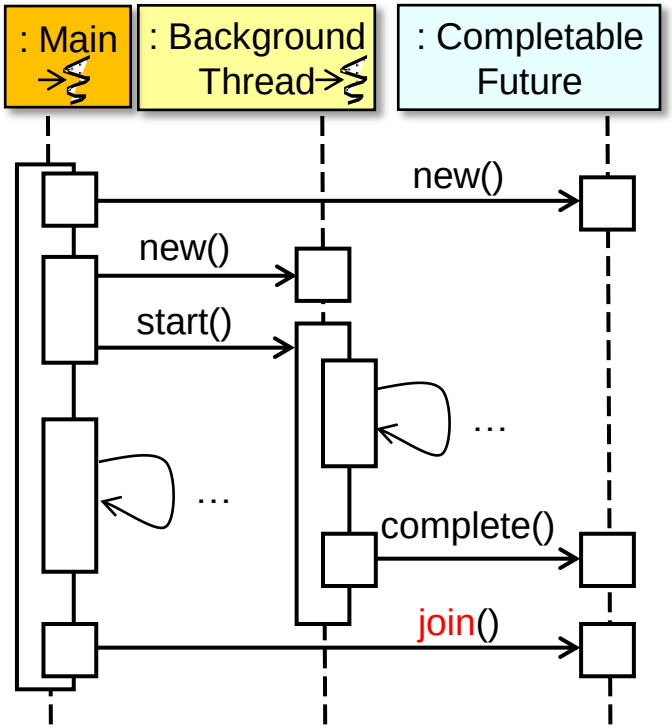


Applying Basic Completable Future Features

- Multiplying big fractions with a completable future

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```

Convert result to a mixed fraction

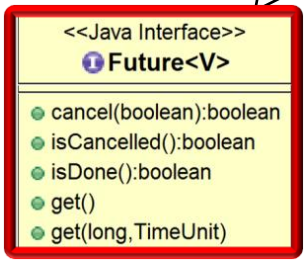


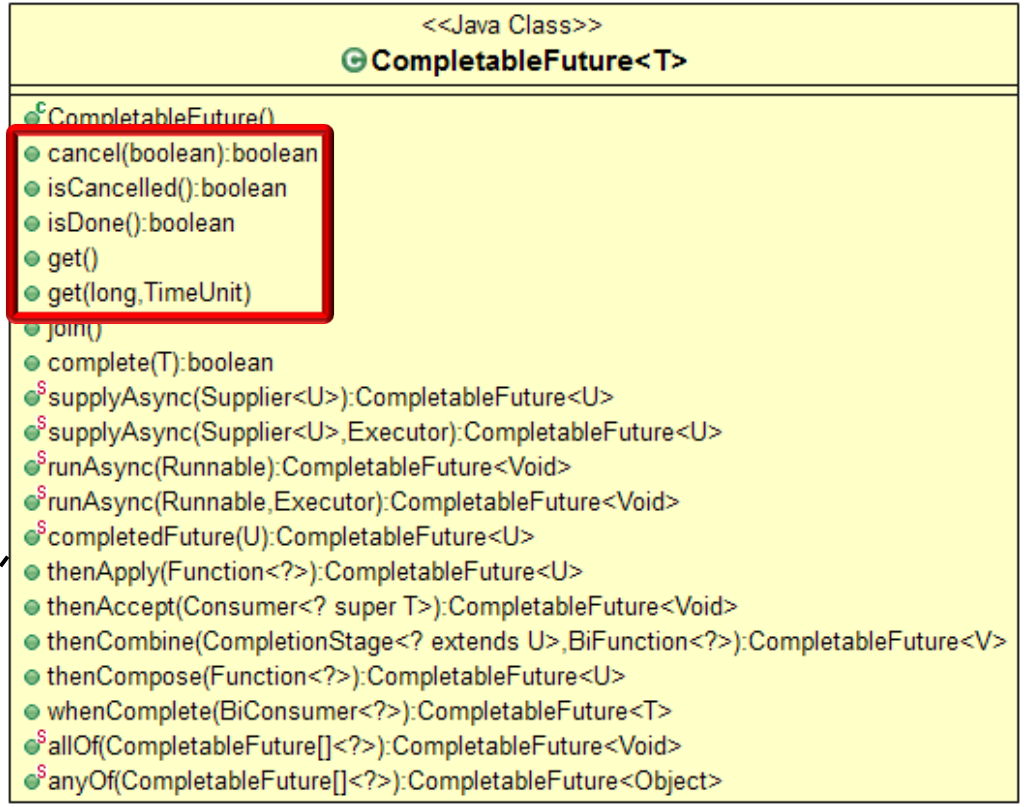
See hwww.mathsisfun.com/mixed-fractions.html

Limitations With Basic Completable Futures Features

Limitations With Basic Completable Futures Features

- Basic completable future features have similar limitations as futures.
  - *Cannot* be chained fluently to handle asynchronous results
  - *Cannot* be triggered reactively
  - *Cannot* be treated efficiently as a *collection* of futures

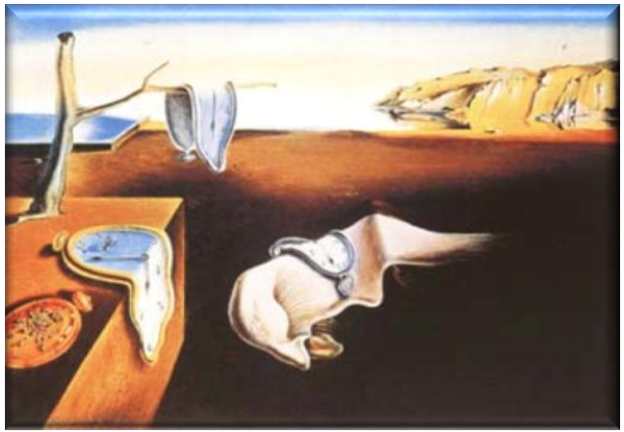




See earlier lesson on "*Evaluating the Pros and Cons of Java Futures*"

Limitations With Basic Completable Futures Features

- Join() blocks until the future is completed.



```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.join().toMixedString());
```

This blocking call underutilizes cores and increases overhead.

Limitations With Basic Completable Futures Features

- Using timed get() is also problematic.

```
CompletableFuture<BigFraction> future
  = new CompletableFuture<>();

new Thread (() -> {
  BigFraction bf1 =
    new BigFraction("62675744/15668936");
  BigFraction bf2 =
    new BigFraction("609136/913704");

  future.complete(bf1.multiply(bf2));
}).start();

...
System.out.println(future.get(1, SECONDS).toMixedString());
```

Using a timeout to bound the blocking duration is inefficient and error-prone.

See crondev.blog/2017/01/23/timeouts-with-java-8-completablefuture-youre-probably-doing-it-wrong

Limitations With Basic Completable Futures Features

- We therefore need to leverage the advanced features of completable futures.

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage<T>, Future<T>

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.

When two or more threads attempt to complete, completeExceptionally, or cancel a CompletableFuture, only one of them succeeds.

In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

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

The End

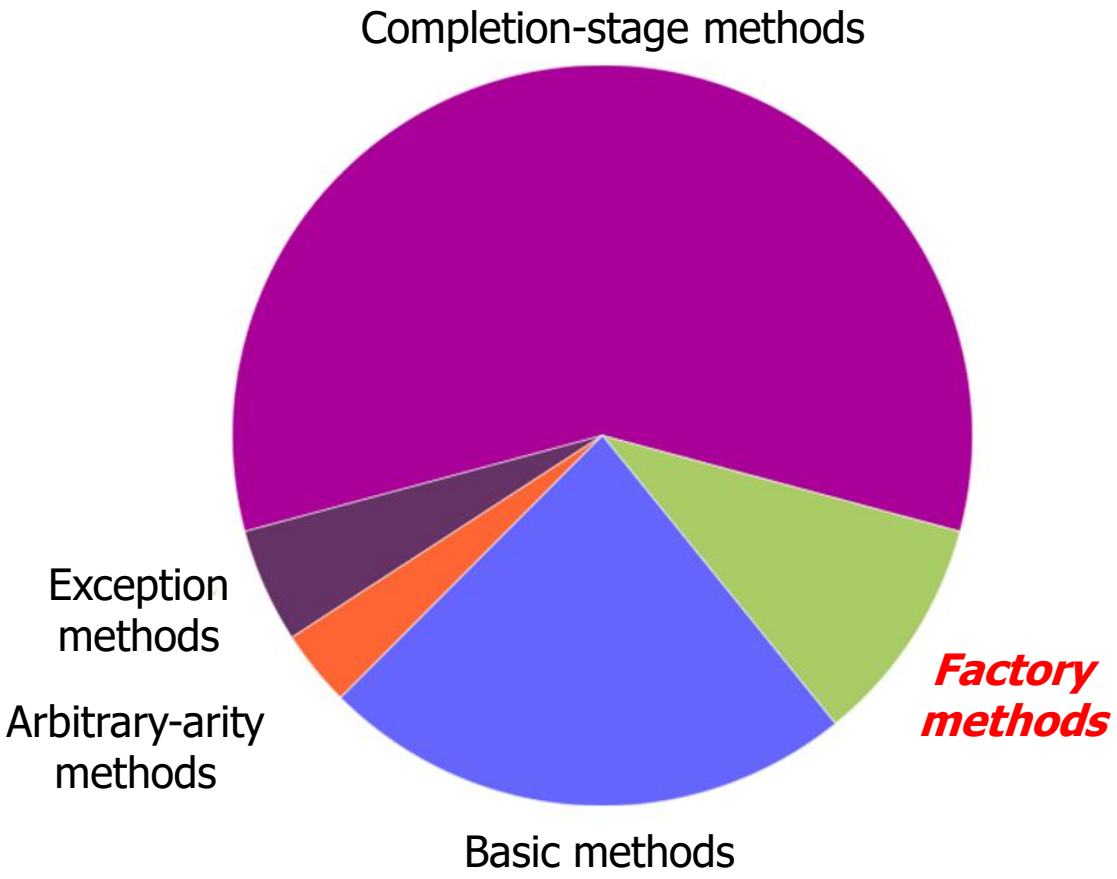
Advanced Java Completable Future Features

Introducing Factory Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations



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

Factory Methods
Initiate Asynchronous Computations

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.

<<Java Class>>

CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

See en.wikipedia.org/wiki/Factory_method_pattern

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.

<<Java Class>>
CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.

Methods	Params	Returns	Behavior
supplyAsync	Supplier	CompletableFuture with result of Supplier	Asynchronously run supplier in common fork-join pool
supplyAsync	Supplier, Executor	CompletableFuture with result of Supplier	Asynchronously run supplier in given executor pool

See docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.
      - Can be passed params and returns a value

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

CompletableFuture<BigFraction> future
  = CompletableFuture
      .supplyAsync(() -> {
        BigFraction bf1 =
          new BigFraction(f1);
        BigFraction bf2 =
          new BigFraction(f2);

        return bf1.multiply(bf2);
      });
```

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

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.
      - Can be passed params and returns a value

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

CompletableFuture<BigFraction> future
  = CompletableFuture
      .supplyAsync(() -> {
        BigFraction bf1 =
          new BigFraction(f1);
        BigFraction bf2 =
          new BigFraction(f2);

        return bf1.multiply(bf2);
      });
```

Params are passed as "effectively final" objects to the supplier lambda.

See javarevisited.blogspot.com/2015/03/what-is-effectively-final-variable-of.html

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.
    - runAsync() enables one-way calls via a runnable.

Methods	Params	Returns	Behavior
runAsync	Runnable	CompletableFuture with result of Void	Asynchronously run runnable in common fork-join pool
runAsync	Runnable, Executor	CompletableFuture with result of Void	Asynchronously run runnable in given executor pool

See docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.
  - runAsync() enables one-way calls via a runnable.
    - Can be passed params but returns no values

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

CompletableFuture<Void> future
  = CompletableFuture
      .runAsync(() -> {
        BigFraction bf1 =
          new BigFraction(f1);
        BigFraction bf2 =
          new BigFraction(f2);

        System.out.println
          (bf1.multiply(bf2)
              .toMixedString());
      });
```

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

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.
    - runAsync() enables one-way calls via a runnable.
      - Can be passed params but returns no values

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

CompletableFuture<Void> future
  = CompletableFuture
      .runAsync(() -> {
        BigFraction bf1 =
          new BigFraction(f1);
        BigFraction bf2 =
          new BigFraction(f2);

        System.out.println
          (bf1.multiply(bf2)
              .toMixedString());
      });
```

"Void" is not a value!

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.
  - runAsync() enables one-way calls via a runnable.
    - Can be passed params but returns no values

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

CompletableFuture<Void> future
  = CompletableFuture
      .runAsync(() -> {
        BigFraction bf1 =
          new BigFraction(f1);
        BigFraction bf2 =
          new BigFraction(f2);

        System.out.println
          (bf1.multiply(bf2)
              .toMixedString());
      });
```

Any output must therefore come from "side effects."

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
    - supplyAsync() allows two-way calls via a supplier.
    - runAsync() enables one-way calls via a runnable.

supplyAsync() is more commonly used than runAsync() in practice.

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
  - Asynchronous functionality runs in a thread pool.

<<Java Class>>
CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
  - Asynchronous functionality runs in a thread pool.

<<Java Class>>
CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

By default, the common fork-join pool is used.

See dzone.com/articles/common-fork-join-pool-and-streams

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
  - Asynchronous functionality runs in a thread pool.

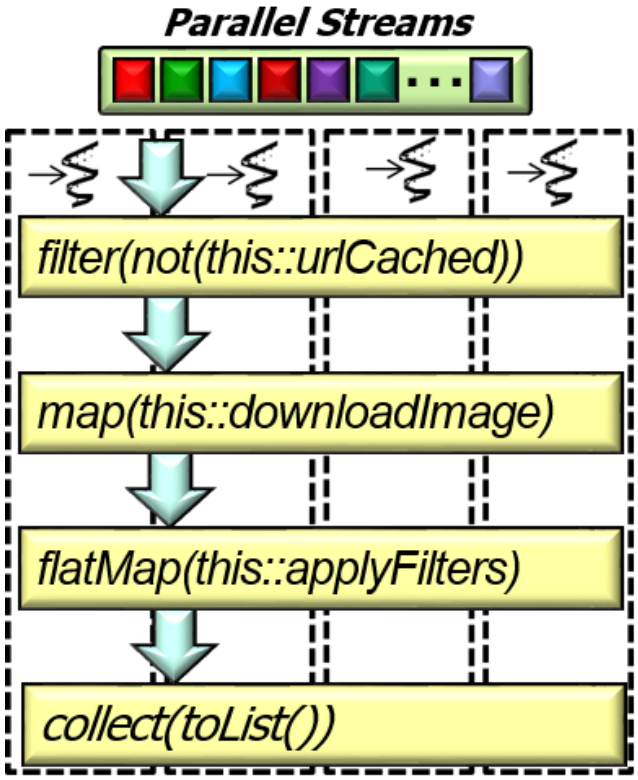
<<Java Class>>
CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

However, a pre- or user-defined thread pool can also be given.

Factory Methods Initiate Asynchronous Computations

- Four factory methods initiate asynchronous computations.
  - These computations may or may not return a value.
  - Asynchronous functionality runs in a thread pool.
    - In contrast, Java parallel streams are designed for use with the common fork-join pool.



See lesson on "*Java Parallel Stream Internals: Parallel Processing via the Common Fork-Join Pool*"

Advanced Java Completable Future
Features: Introducing Factory Methods

The End

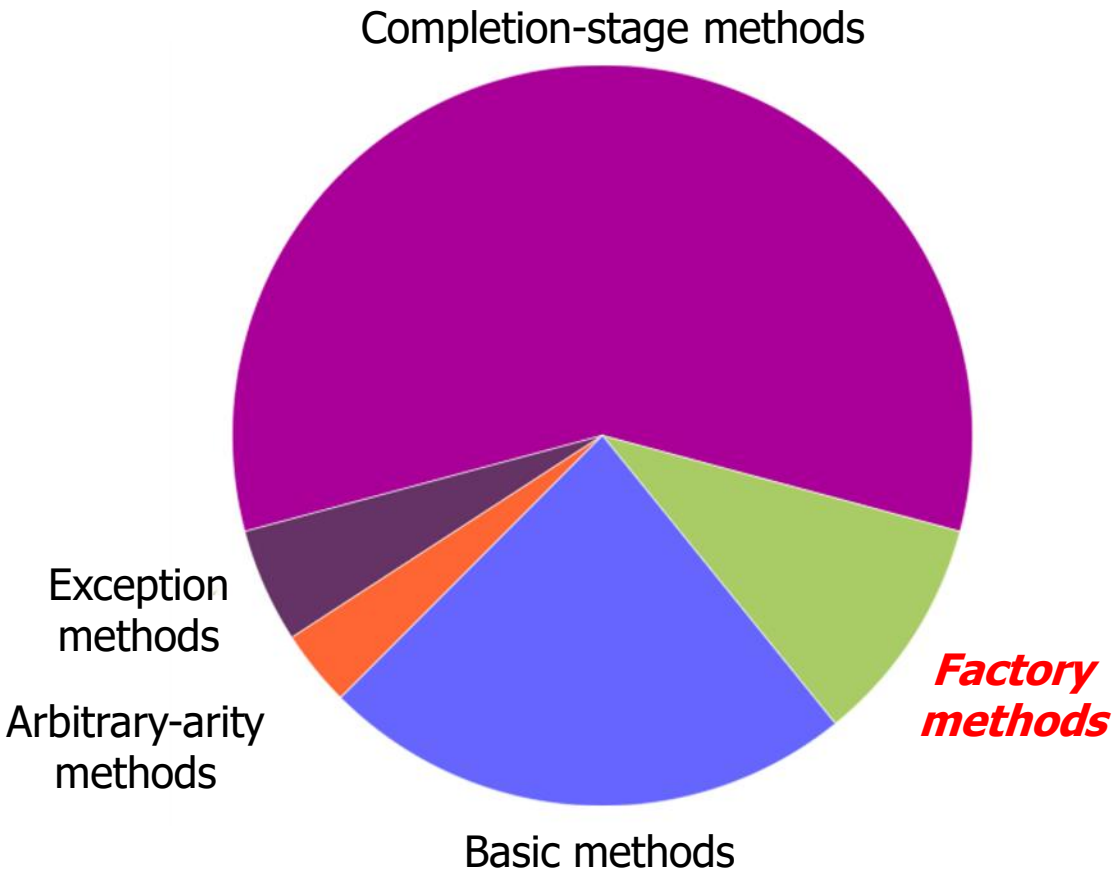
Advanced Java Completable Future Features

Applying Factory Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations
    - Applying factory methods

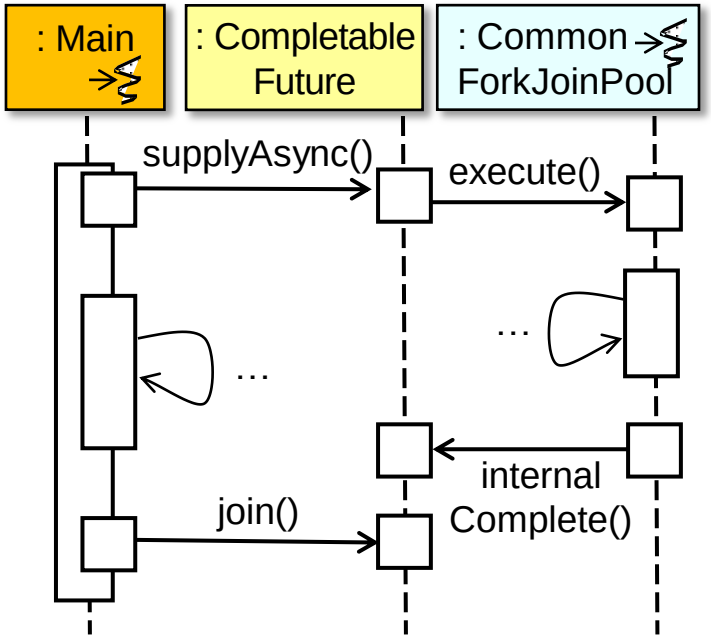


Applying Completable Future Factory Methods

Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```

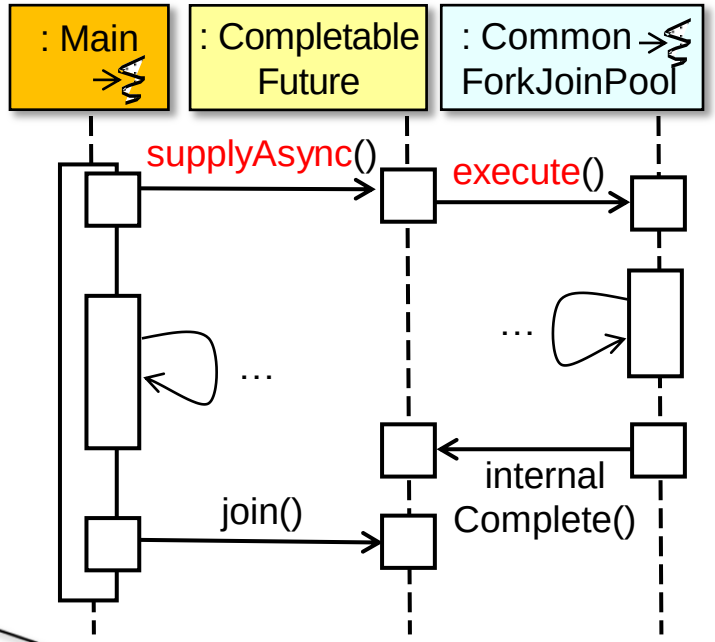


See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
      BigFraction bf1 =
        new BigFraction(f1);
      BigFraction bf2 =
        new BigFraction(f2);

      return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```



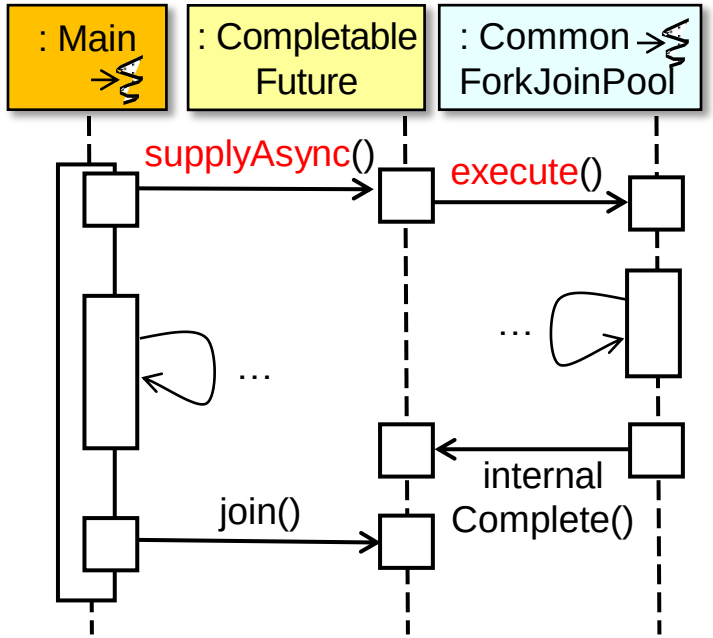
Arrange to execute the supplier lambda in common fork-join pool

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

Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
      BigFraction bf1 =
        new BigFraction(f1);
      BigFraction bf2 =
        new BigFraction(f2);

      return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```

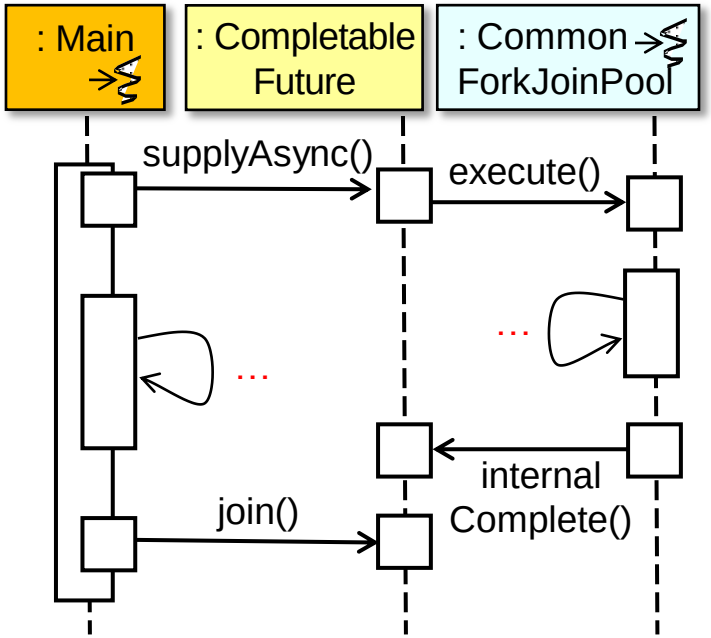


Define a supplier lambda that multiplies two BigFractions

Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
      BigFraction bf1 =
        new BigFraction(f1);
      BigFraction bf2 =
        new BigFraction(f2);

      return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```

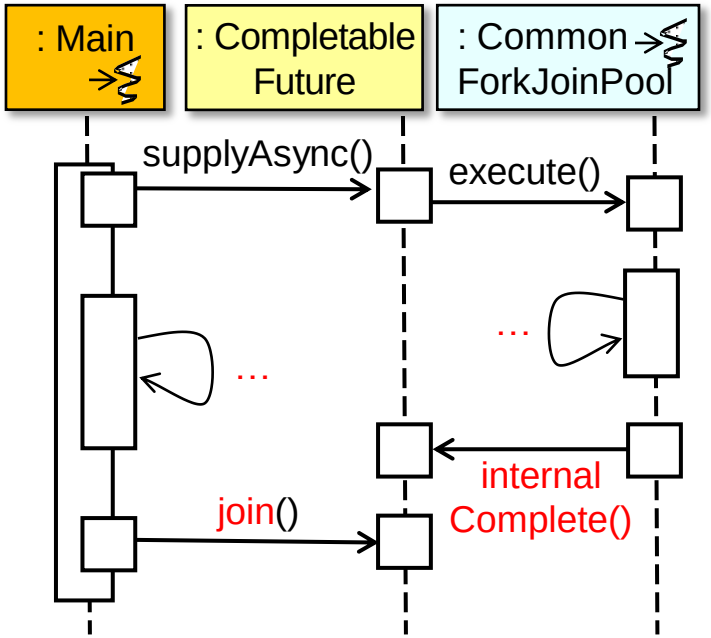


These computations run concurrently.

Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```



join() blocks until result is complete.

Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```

Calling CompletableFuture.supplyAsync() avoids the use of threads in this example!

Applying Completable Future Factory Methods

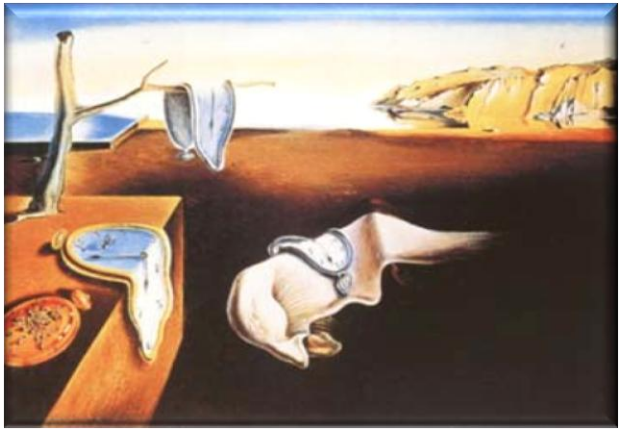
- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```

There's no need to explicitly complete the future since supplyAsync() returns one.

Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions



```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
  CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);
    });
...
System.out.println(future.join().toMixedString());
```

However, we still must fix the problem with calling join() explicitly.

Advanced Java Completable Future Features: Applying Factory Methods

The End

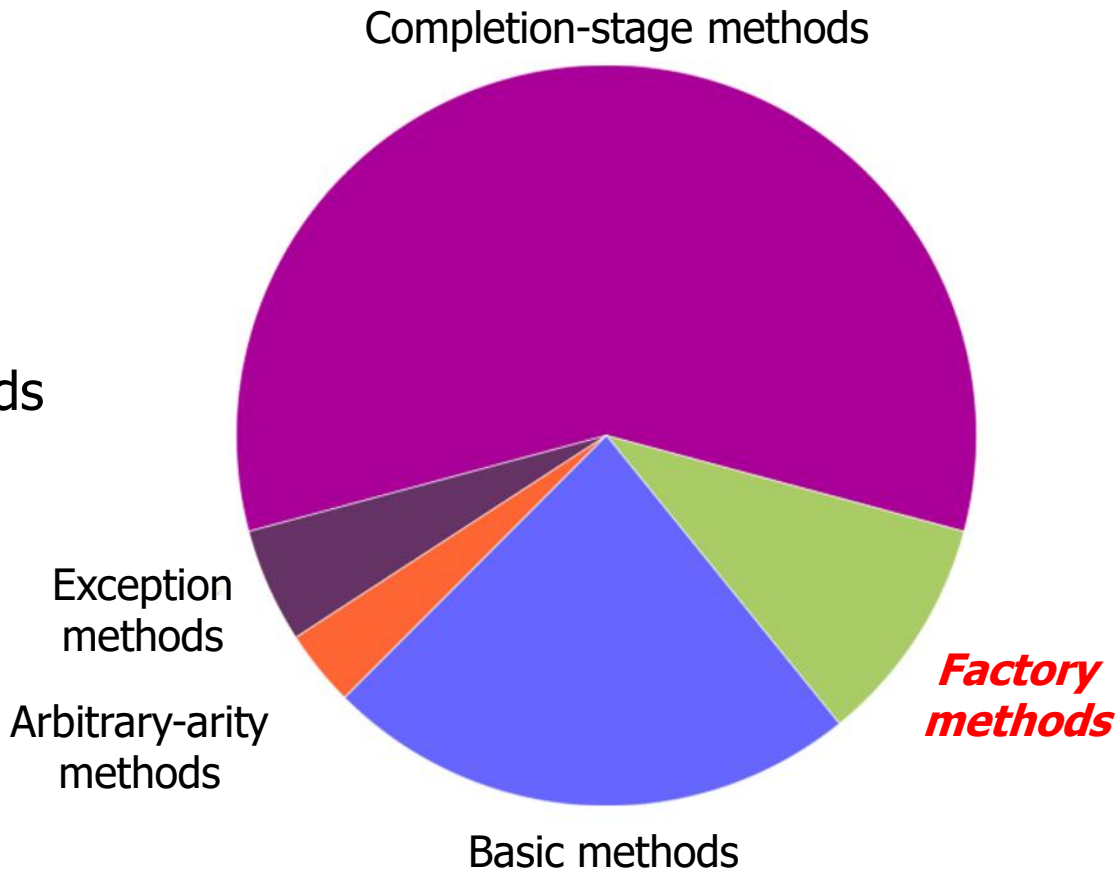
Advanced Java Completable Future Features

Factory Method Internals

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations
    - Applying factory methods
    - Internals of factory methods



Internals of Completable Future Factory Methods

Internals of Completable Future Factory Methods

- The supplyAsync() method runs the supplier lambda in a thread residing in the common fork-join pool.

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

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

Internals of Completable Future Factory Methods

- The supplyAsync() method runs the supplier lambda in a thread residing in the common fork-join pool.

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

supplyAsync() does not create a new thread!

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

Internals of Completable Future Factory Methods

- The supplyAsync() method runs the supplier lambda in a thread residing in the common fork-join pool.

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

Instead, it returns a future that's completed by a worker thread running in common fork-join pool.

Internals of Completable Future Factory Methods

- The supplyAsync() method runs the supplier lambda in a thread residing in the common fork-join pool.

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

supplyAsync()'s parameter is a supplier lambda that multiplies two BigFractions.

Internals of Completable Future Factory Methods

- The supplyAsync() method runs the supplier lambda in a thread residing in the common fork-join pool.

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

Although Supplier.get() takes no params, effectively final values can be passed to this supplier lambda.

See javarevisited.blogspot.com/2015/03/what-is-effectively-final-variable-of.html

Internals of Completable Future Factory Methods

- The supplyAsync() method runs the supplier lambda in a thread residing in the common fork-join pool.

```
String f1("62675744/15668936"); String f2("609136/913704");

CompletableFuture<BigFraction> future = CompletableFuture
    .supplyAsync(() -> {
       BigFraction bf1 =
         new BigFraction(f1);
       BigFraction bf2 =
         new BigFraction(f2);

       return bf1.multiply(bf2);});

System.out.println(future.join().toMixedString());
```

The worker thread calls the Supplier.get() method to obtain this supplier lambda and perform the computation.

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

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

Here's how supplyAsync() code uses the supplier passed to it.

See classes/java/util/concurrent/CompletableFuture.java

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

```
() -> { ... return
  bf1.multiply(bf2);
}
```

The supplier parameter is bound to the lambda passed to supplyAsync().

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

The supplier is encapsulated in an AsyncSupply message.

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {
  ...
  CompletableFuture<U> f =
    new CompletableFuture<U>();

  execAsync(ForkJoinPool.commonPool(),
            new AsyncSupply<U>(supplier, f));

  return f;
}
...
```

This message is enqueued for async execution in common fork-join pool.

This design is one example of "message passing" à la reactive programming!

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
...
static final class AsyncSupply<U> extends Async {
    final Supplier<U> fn;

    AsyncSupply(Supplier<U> fn, ...) { this.fn = fn; ... }

    public final boolean exec() {
        ...
        U u = fn.get();
        ...
    }
}
```

Async extends ForkJoinTask and Runnable, so it can be executed in a thread pool.

See classes/java/util/concurrent/CompletableFuture.java

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
...
static final class AsyncSupply<U> extends Async {
    final Supplier<U> fn;

    AsyncSupply(Supplier<U> fn, ...) { this.fn = fn; ... }

    public final boolean exec() {
        ...
        U u = fn.get();
        ...
    }
}
```

```
() -> { ... return
  bf1.multiply(bf2);
}
```

AsyncSupply stores the original supplier lambda passed into supplyAsync().

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
...
static final class AsyncSupply<U> extends Async {
    final Supplier<U> fn;

    AsyncSupply(Supplier<U> fn, ...) { this.fn = fn; ... }

    public final boolean exec() {
        ...
        U u = fn.get();
        ...
    }
}
```

```
() -> { ... return
  bf1.multiply(bf2);
}
```

A worker thread in the pool then runs the supplier lambda asynchronously.

Internals of Completable Future Factory Methods

- The supplyAsync() method arranges to execute a supplier lambda in a worker thread residing in the common fork-join pool.

```
...
static final class AsyncSupply<U> extends Async {
    final Supplier<U> fn;

    AsyncSupply(Supplier<U> fn, ...) { this.fn = fn; ... }

    public final boolean exec() {
        ...
        U u = fn.get();
        ...
    }
}
```

This get() method could use ForkJoinPool ManagedBlocker mechanism to auto-scale the pool size for blocking operations.

See earlier lesson on "*The Java Fork-Join Pool: The ManagedBlocker Interface*"

The End

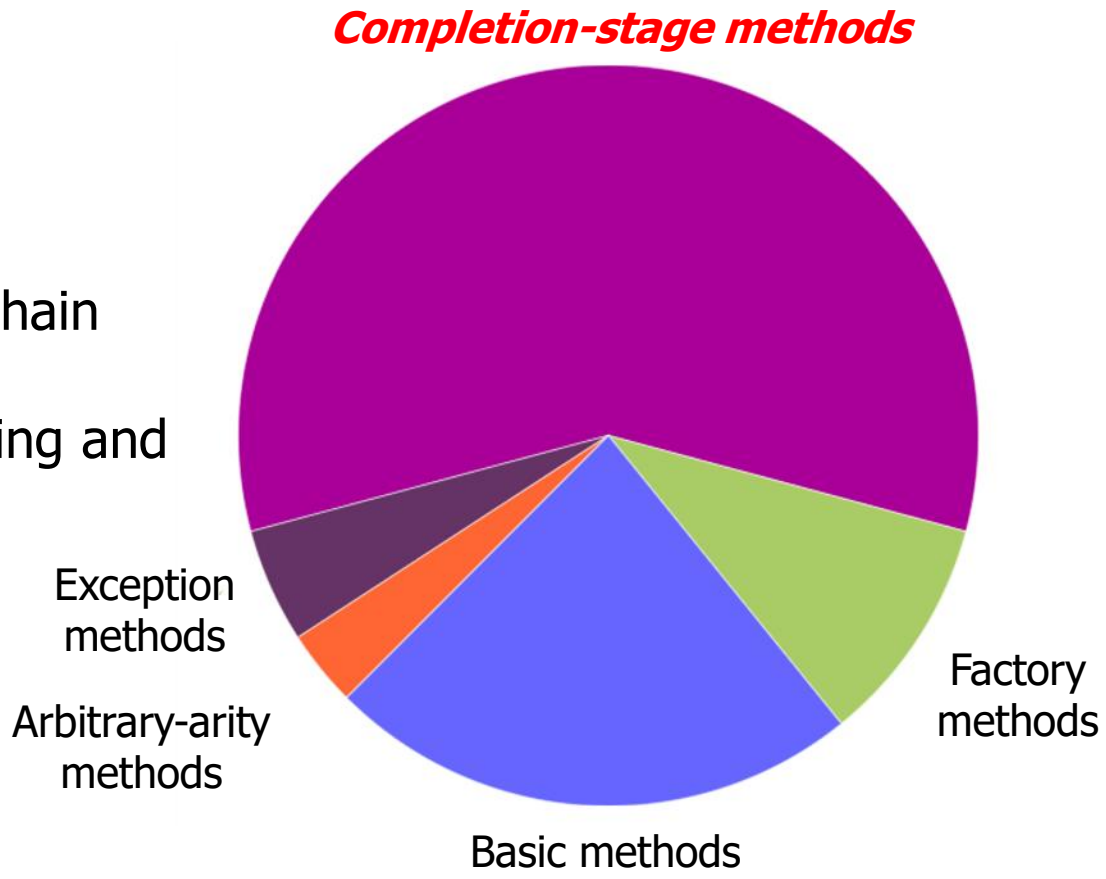
Advanced Java Completable Future Features

Introducing Completion-Stage Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

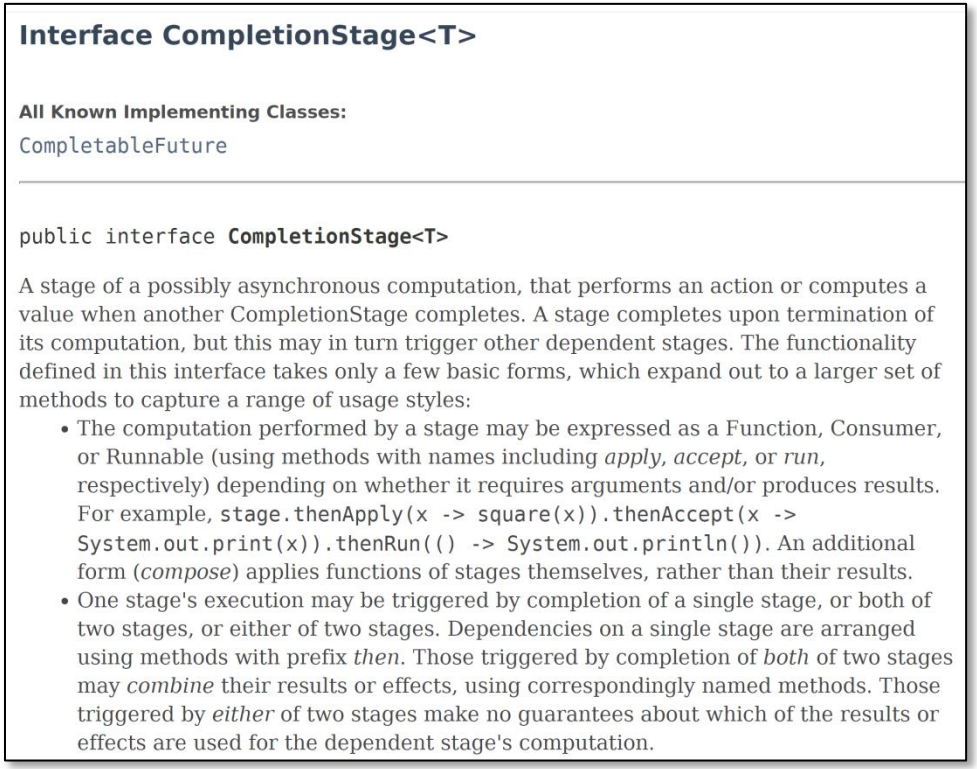
- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.



Completion-Stage Methods Chain Actions Together

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.



Interface CompletionStage<T>

All Known Implementing Classes:

CompletableFuture

`public interface CompletionStage<T>`

A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes. A stage completes upon termination of its computation, but this may in turn trigger other dependent stages. The functionality defined in this interface takes only a few basic forms, which expand out to a larger set of methods to capture a range of usage styles:

- The computation performed by a stage may be expressed as a Function, Consumer, or Runnable (using methods with names including *apply*, *accept*, or *run*, respectively) depending on whether it requires arguments and/or produces results. For example, `stage.thenApply(x -> square(x)).thenAccept(x -> System.out.print(x)).thenRun(() -> System.out.println())`. An additional form (*compose*) applies functions of stages themselves, rather than their results.
- One stage's execution may be triggered by completion of a single stage, or both of two stages, or either of two stages. Dependencies on a single stage are arranged using methods with prefix *then*. Those triggered by completion of *both* of two stages may *combine* their results or effects, using correspondingly named methods. Those triggered by *either* of two stages make no guarantees about which of the results or effects are used for the dependent stage's computation.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("8461225360066988 2"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   ...
```

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

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.

Create an unreduced big fraction variable

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("8461225360066988 2"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   ...
```

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("846122553600669882"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
   BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   ...
```

Create a supplier lambda variable that will reduce the big fraction

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("846122553600669882"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   ...
```

This factory method will asynchronously reduce the big fraction supplier lambda.

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("846122553600669882"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   ...
```

thenApply()'s action is triggered when future from supplyAsync() completes.

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.
  - Methods can be chained together "fluently."

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("846122553600669882"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

thenAccept()'s action is triggered when future from thenApply() completes.

See en.wikipedia.org/wiki/Fluent_interface

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.
  - Methods can be chained together "fluently."
    - Each method registers a lambda action to apply.


```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("846122553600669882"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.
  - Methods can be chained together "fluently."
    - Each method registers a lambda action to apply.
    - A lambda action is called only after previous stage completes successfully.

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("846122553600669882"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

This is what is meant by "chaining."

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.
  - A dependent action runs on a completed asynchronous call result.
  - Methods can be chained together "fluently."
    - Each method registers a lambda action to apply.
    - A lambda action is called only after previous stage completes successfully.

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger
                ("846122553600669882"),
            new BigInteger
                ("18802723413348219 6"),
            false); // Don't reduce!

Supplier<BigFraction> reduce = () ->
  BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

Action is "deferred" until previous stage completes and fork-join thread is available.

Completion-Stage Methods Chain Actions Together

- Use completion stages to avoid blocking a thread until the result *must* be obtained.

Completion-Stage Methods Chain Actions Together

- Use completion stages to avoid blocking a thread until the result *must* be obtained.
  - Try not to call join() or get() unless absolutely necessary.

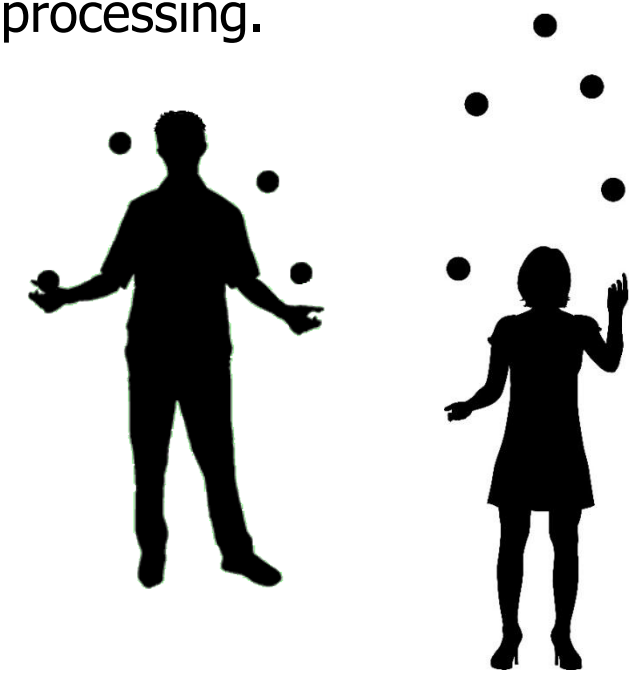
Servers may avoid blocking completely, whereas clients may need join() sparingly.

Completion-Stage Methods Chain Actions Together

- Use completion stages to avoid blocking a thread until the result *must* be obtained.
  - Try not to call join() or get() unless absolutely necessary.
    - The goal is to improve responsiveness.

Completion-Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for asynchronous result processing.



<<Java Class>>

CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Juggling is a good analogy for completion stages!

The End

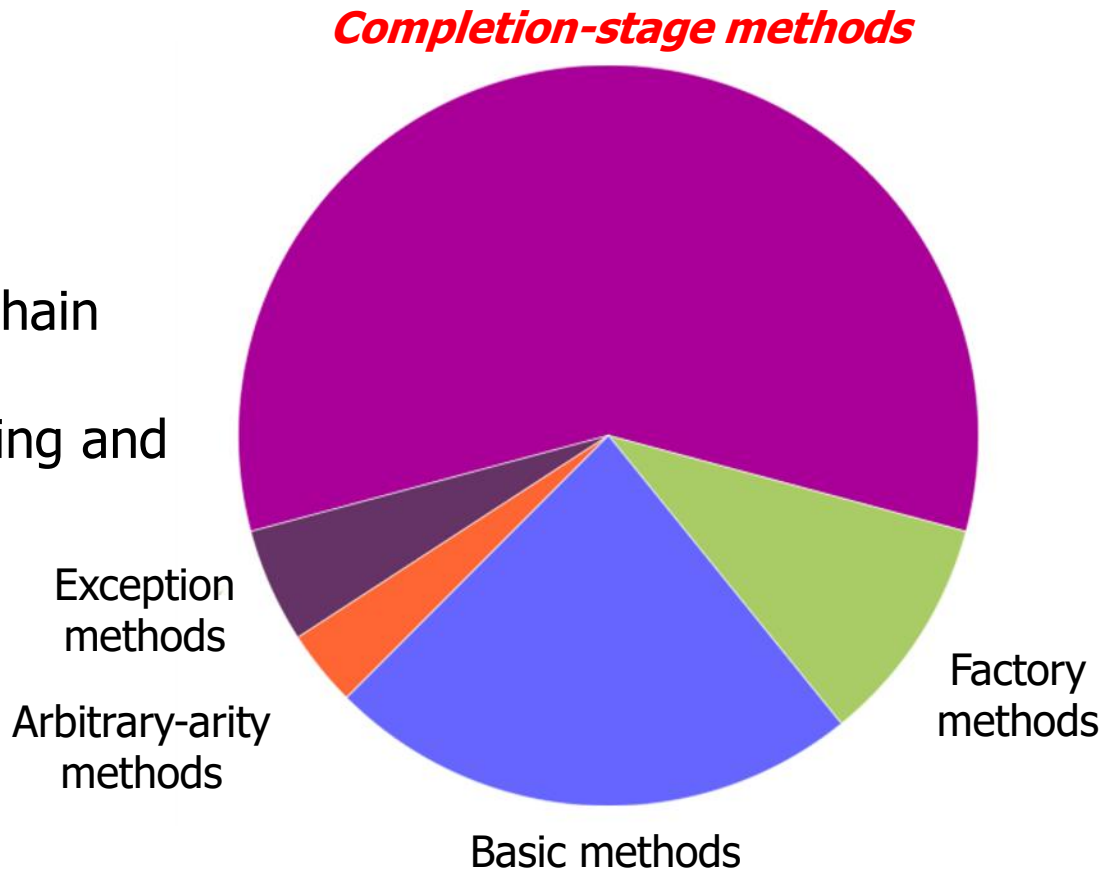
Advanced Java Completable Future Features

Grouping Completion-Stage Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

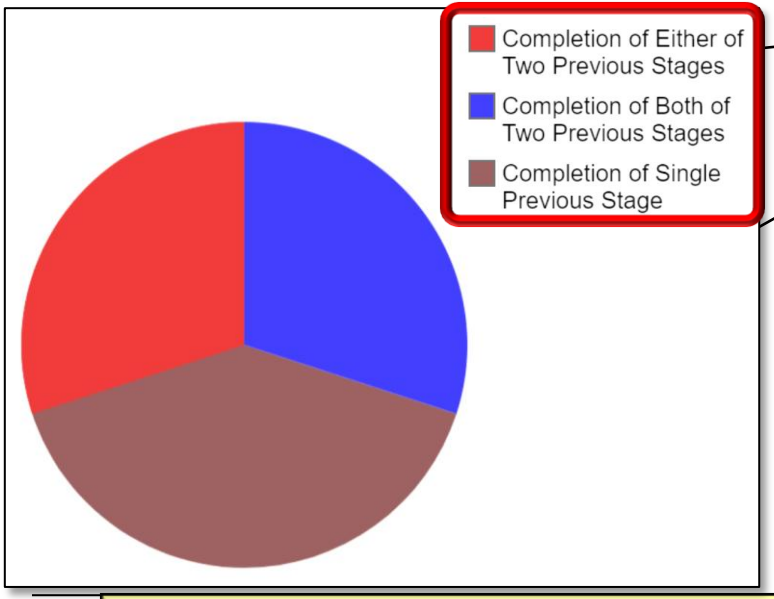
- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
    - Method grouping

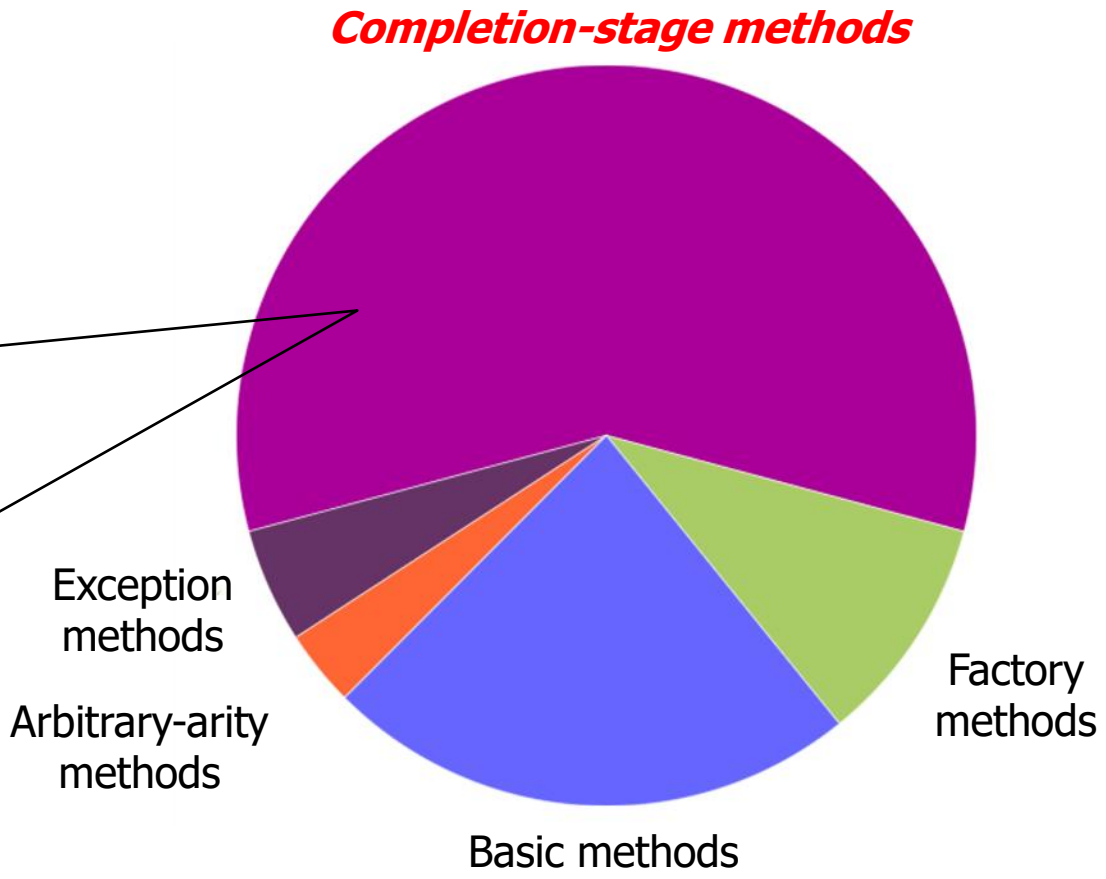


Grouping Completable Future Completion-Stage Methods

Grouping Completable Future Completion-Stage Methods

- Completion-stage methods are grouped based on how a stage is triggered by one or more previous stages.





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

Grouping Completable Future Completion-Stage Methods

- Completion-stage methods are grouped based on how a stage is triggered by one or more previous stages.
  - Completion of a single previous stage

These methods run in the invoking thread or the same thread as previous stage.

Methods	Params	Returns	Behavior
thenApply (Async)	Function	Completable Future with Function result	Apply function to result of the previous stage
then Compose (Async)	Function	Completable Future with Function result directly, *not* a nested future	Apply function to result of the previous stage
then Accept (Async)	Consumer	Completable Future<Void>	Consumer handles result of previous stage
thenRun (Async)	Runnable	Completable Future<Void>	Run action without returning value

The thread that executes these methods depends on various runtime factors.

Grouping Completable Future Completion-Stage Methods

- Completion-stage methods are grouped based on how a stage is triggered by one or more previous stages.
  - Completion of a single previous stage

**Async() variants run in common fork-join pool.*

Methods	Params	Returns	Behavior
thenApply (Async)	Function	Completable Future with Function result	Apply function to result of the previous stage
then Compose (Async)	Function	Completable Future with Function result directly, *not* a nested future	Apply function to result of the previous stage
then Accept (Async)	Consumer	Completable Future<Void>	Consumer handles result of previous stage
thenRun (Async)	Runnable	Completable Future<Void>	Run action without returning value

See blog.krecan.net/2013/12/25/completablefutures-why-to-use-async-methods

Grouping Completable Future Completion-Stage Methods

- Completion-stage methods are grouped based on how a stage is triggered by one or more previous stages.
  - Completion of a single previous stage
  - Completion of both of two previous stages
    - An "and"

Methods	Params	Returns	Behavior
then Combine (Async)	Bi Function	Completable Future with Bi Function result	Apply BiFunction to results of both previous stages
then Accept Both (Async)	Bi Consumer	Completable Future<Void>	BiConsumer handles results of both previous stages
runAfter Both (Async)	Runnable	Completable Future<Void>	Run action when both previous stages complete

Grouping Completable Future Completion-Stage Methods

- Completion-stage methods are grouped based on how a stage is triggered by one or more previous stages.
  - Completion of a single previous stage
  - Completion of both of two previous stages
  - Completion of either of two previous stages
    - An "or"

Methods	Params	Returns	Behavior
applyToEither (Async)	Function	CompletableFuture with Function result	Apply Function to results of either previous stage
acceptEither (Async)	Consumer	CompletableFuture<Void>	Consumer handles results of either previous stage
runAfterEither (Async)	Runnable	CompletableFuture<Void>	Run action when either previous stage completes

Advanced Java Completable Future
Features: Grouping Completion-Stage Methods

The End

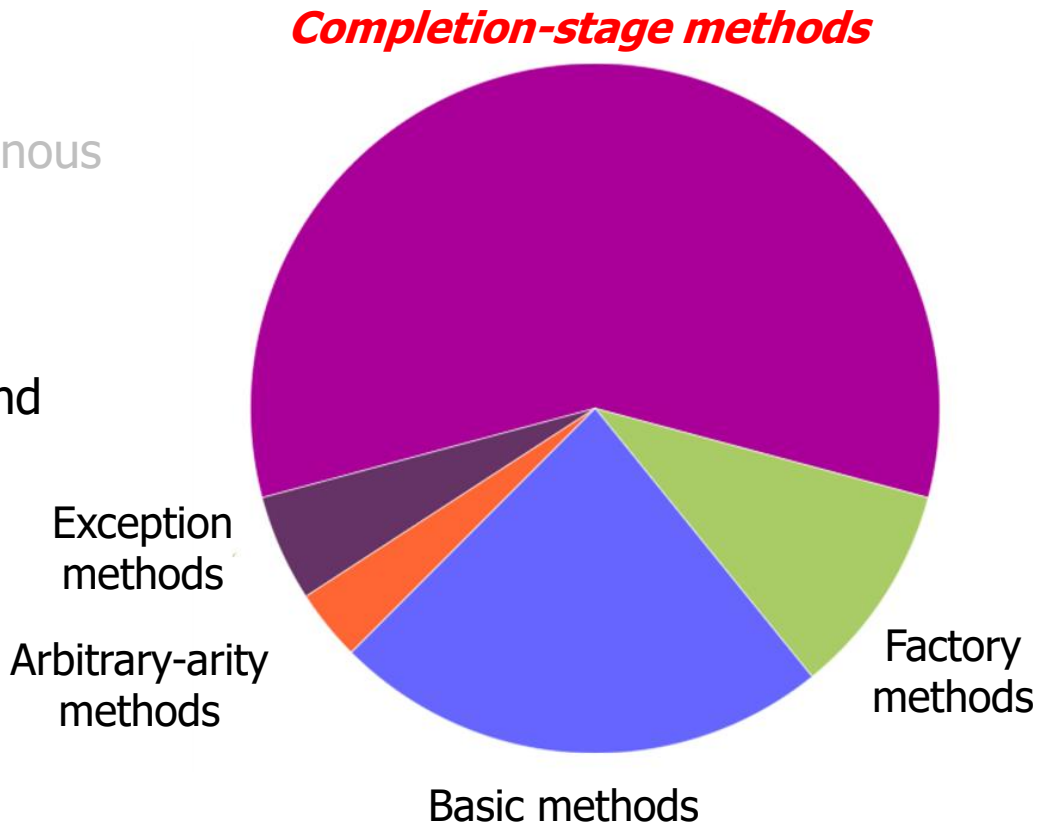
Advanced Java Completable Future Features

Single-Stage Completion Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
    - Method grouping
    - Single-stage methods



Methods Triggered by Completion of a Single Stage

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()

```
CompletableFuture<U> thenApply
   (Function<? super T,
             ? extends U> fn)
{ ... }
```

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
    - Applies a function action to the previous stage's result

```
CompletableFuture<U> thenApply
   (Function<? super T,
             ? extends U> fn)
{ ... }
```

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
    - Applies a function action to the previous stage's result
    - Returns a future containing the result of the action

```
CompletableFuture<U> thenApply
   (Function<? super T,
             ? extends U> fn)
{ ... }
```

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
    - Applies a function action to the previous stage's result
    - Returns a future containing the result of the action
    - Used for a quick *synchronous* action that returns a value rather than a future

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger("..."),
            new BigInteger("..."),
            false); // Don't reduce!

Supplier<BigFraction> reduce = ()
  -> BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   ...
```

toMixedString() returns a string value.

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()

```
CompletableFuture<U> thenCompose
   (Function<? super T,
             ? extends
             CompletionStage<U>> fn)
{ ... }
```

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result

```
CompletableFuture<U> thenCompose
   (Function<? super T,
             ? extends
             CompletionStage<U>> fn)
{ ... }
```

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
      - *Not* a nested future

```
CompletableFuture<U> thenCompose
   (Function<? super T,
             ? extends
             CompletionStage<U>> fn)
{ ... }
```

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
      - *Not* a nested future

```
CompletableFuture<U> thenCompose
   (Function<? super T,
             ? extends
             CompletionStage<U>> fn)
{ ... }
```

thenCompose() is similar to flatMap() on a Stream or Optional.

See dzone.com/articles/understanding-flatmap

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future

```
Function<BF,
         CompletableFuture<BF>>
  reduceAndMultiplyFractions =
    unreduced -> CompletableFuture
    .supplyAsync
      (() -> BF.reduce(unreduced))


    .thenCompose
      (reduced -> CompletableFuture
       .supplyAsync(() ->
        reduced.multiply(...)));
...
```

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future

```
Function<BF,
         CompletableFuture<BF>>
  reduceAndMultiplyFractions =
    unreduced -> CompletableFuture
    .supplyAsync
      (() -> BF.reduce(unreduced))
```

This function reduces and multiplies big fractions.

```
    .thenCompose
      (reduced -> CompletableFuture
       .supplyAsync(() ->
        reduced.multiply(...)));
...
```

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future

```
Function<BF,
         CompletableFuture<BF>>
  reduceAndMultiplyFractions =
    unreduced -> CompletableFuture
    .supplyAsync
      (() -> BF.reduce(unreduced))
    .thenCompose
      (reduced -> CompletableFuture
       .supplyAsync(() ->
        reduced.multiply(...)));
...
```

Reduce big fraction asynchronously and return a completable future

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future

```
Function<BF,
         CompletableFuture<BF>>
  reduceAndMultiplyFractions =
    unreduced -> CompletableFuture
    .supplyAsync
      (() -> BF.reduce(unreduced))

    .thenCompose
      (reduced -> CompletableFuture
       .supplyAsync(() ->
        reduced.multiply(...)));
...
```

supplyAsync() returns a future, but thenCompose() "flattens" this future.

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future
    - Avoids unwieldy nesting of futures à la thenApply()

Nesting is unwieldy!

```
Function<BF, CompletableFuture<
              CompletableFuture<BF>>>
reduceAndMultiplyFractions =
  unreduced -> CompletableFuture
  .supplyAsync
    (() -> BF.reduce(unreduced))


  .thenApply
    (reduced -> CompletableFuture
     .supplyAsync(() ->
      reduced.multiply(...)));
...
```

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future
    - Avoids unwieldy nesting of futures à la thenApply()

Flattening is more concise!

```
Function<BF,
         CompletableFuture<BF>>
reduceAndMultiplyFractions =
  unreduced -> CompletableFuture
  .supplyAsync
    (() -> BF.reduce(unreduced))


  .thenApplyAsync(reduced
     -> reduced.multiply(...)));
...
...
```

thenApplyAsync() can often replace thenCompose(supplyAsync()) nestings.

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future
    - Avoids unwieldy nesting of futures à la thenApply()

```
CompletableFuture<Integer> countF =
   .CompletableFuture
      .supplyAsync
         (() ->
          longRunnerReturnsCF())

      .thenCompose
        (Function.identity())
...
```

supplyAsync() will return a CompletableFuture to a CompletableFuture here!

Can be used to avoid calling join() when flattening nested completable futures

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future
    - Avoids unwieldy nesting of futures à la thenApply()

```
CompletableFuture<Integer> countF =
   .CompletableFuture
      .supplyAsync
         (() ->
          longRunnerReturnsCF())

      .thenCompose
        (Function.identity())
...
```

This idiom flattens the return value to "just" one CompletableFuture!

Can be used to avoid calling join() when flattening nested completable futures

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for a longer *asynchronous* action that returns a future
    - Avoids unwieldy nesting of futures à la thenApply()

```
CompletableFuture<Integer> countF =
   .CompletableFuture
      .supplyAsync
         (() ->
          longRunnerReturnsCF())

      .thenComposeAsync
        (this::longBlockerReturnsCF)
...
```

Runs longBlockerReturnsCF() in a thread in the fork-join pool

thenComposeAsync() can be used to avoid calling supplyAsync() again in a chain.

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()

```
CompletableFuture<Void>
  thenAccept
    (Consumer<? super T> action)
{ ... }
```

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result

```
CompletableFuture<Void>
  thenAccept
    (Consumer<? super T> action)
{ ... }
```

See docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result

```
CompletableFuture<Void>
   thenAccept
      (Consumer<? super T> action)
{  ...  }
```

This action behaves as a "callback" with a side effect.

See en.wikipedia.org/wiki/Callback_(computer_programming)

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void

```
CompletableFuture<Void>
  thenAccept
    (Consumer<? super T> action)
{ ... }
```

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger("..."),
            new BigInteger("..."),
            false); // Don't reduce!

Supplier<BigFraction> reduce = ()
   -> BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

thenApply() returns a string future that thenAccept() prints when it completes.

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

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

```
BigFraction unreduced = BigFraction
   .valueOf(new BigInteger("..."),
            new BigInteger("..."),
            false); // Don't reduce!

Supplier<BigFraction> reduce = ()
  -> BigFraction.reduce(unreduced);

CompletableFuture
   .supplyAsync(reduce)
   .thenApply(BigFraction
                ::toMixedString)
   .thenAccept(System.out::println);
```

println() is a callback that has a side effect (printing the mixed string).

Methods Triggered by Completion of a Single Stage

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void
    - Often used at the end of a chain of completion stages
    - May lead to "callback hell!"

See dzone.com/articles/callback-hell

The End

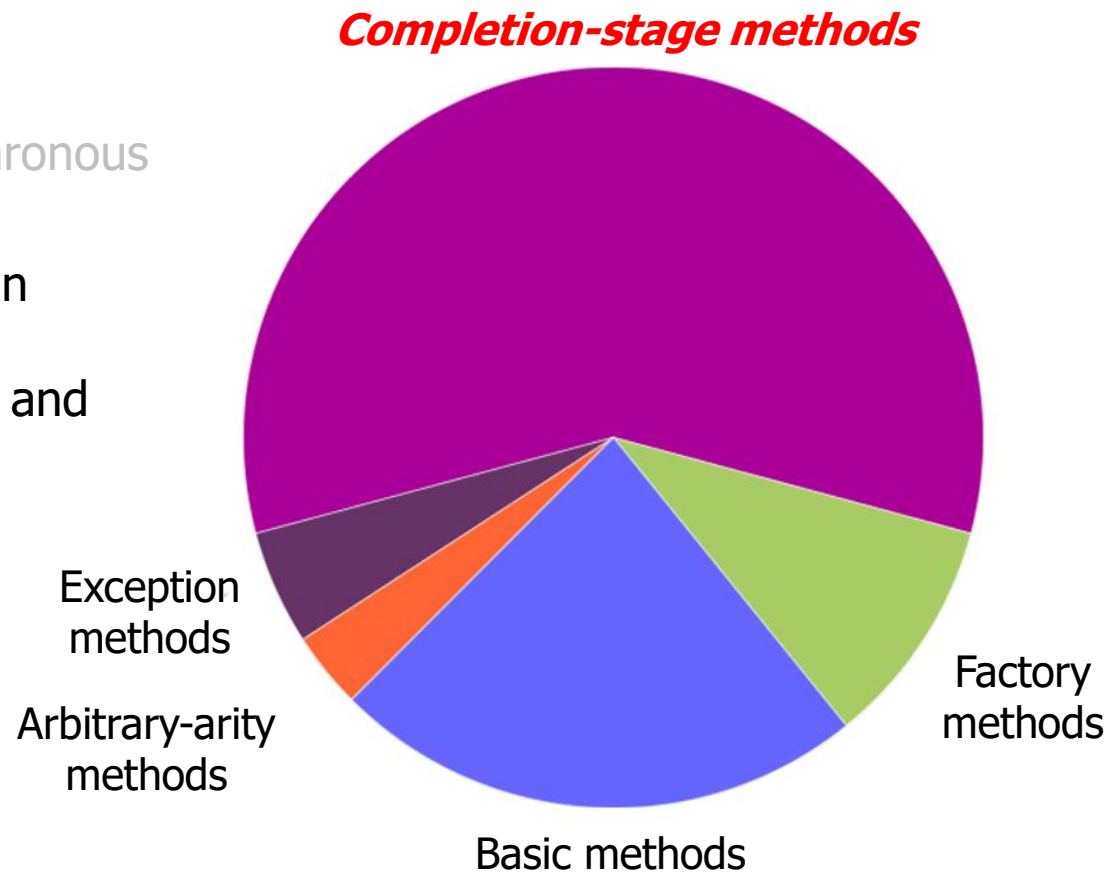
Advanced Java Completable Future Features: Two-Stage Completion Methods

Part I

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

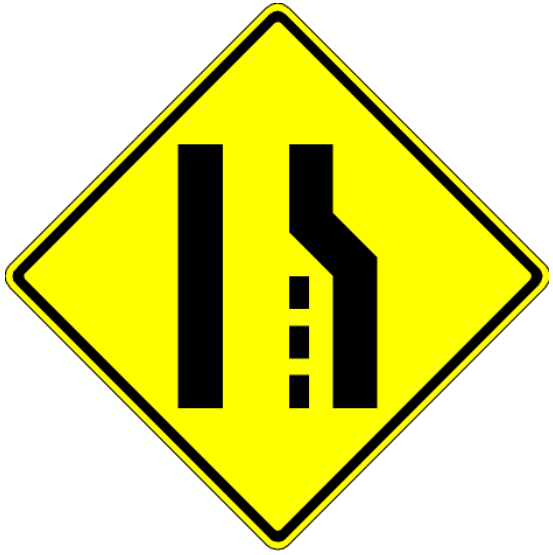
- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
    - Method grouping
    - Single-stage methods
    - Two-stage methods (and)



Methods Triggered by Completion of Both of Two Stages

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()

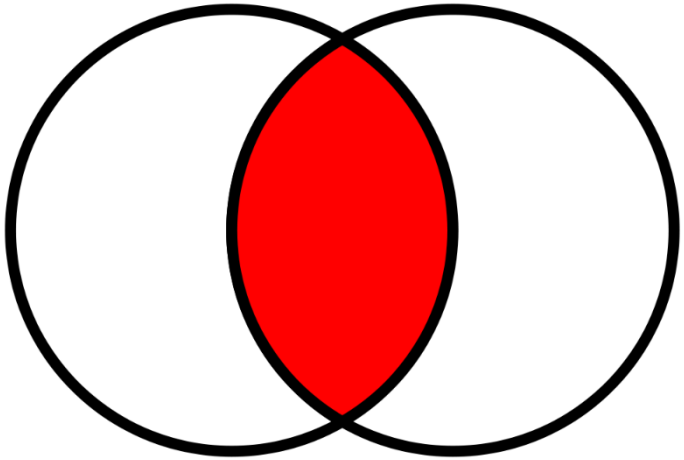


```
CompletableFuture<U> thenCombine
   (CompletionStage<? Extends U>
                          other,
    BiFunction<? super T,
               ? super U,
               ? extends V> fn)
{ ... }
```

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

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results



```
CompletableFuture<U> thenCombine
  (CompletionStage<? Extends U>
                     other,
   BiFunction<? super T,
              ? super U,
              ? extends V> fn)
{ ... }
```

See en.wikipedia.org/wiki/Logical_conjunction

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results
      - Two futures are used here:
        - The future used to invoke thenCombine()
        - The "other" future passed to thenCombine()

```
CompletableFuture<U> thenCombine
  (CompletionStage<? Extends U>
                   other,
   BiFunction<? super T,
              ? super U,
              ? extends V> fn)
{ ... }
```

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results
    - Returns a future containing the result of the action

```
CompletableFuture<U> thenCombine
  (CompletionStage<? Extends U>
                     other,
   BiFunction<? super T,
              ? super U,
              ? extends V> fn)
{ ... }
```

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results
    - Returns a future containing the result of the action

```
CompletableFuture<U> thenCombine
  (CompletionStage<? Extends U>
                     other,
   BiFunction<? super T,
              ? super U,
              ? extends V> fn)
{ ... }
```

thenCombine() essentially performs a "reduction."

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results
    - Returns a future containing the result of the action
    - Used to "join" two paths of asynchronous execution

```
CompletableFuture<BF> compF1 =
    CompletableFuture
    .supplyAsync(() ->
       /* multiply two BFs. */);

CompletableFuture<BF> compF2 =
    CompletableFuture
    .supplyAsync(() ->
       /* divide two BFs. */);

compF1
   .thenCombine(compF2,
                BigFraction::add)

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

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

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results
    - Returns a future containing the result of the action
    - Used to "join" two paths of asynchronous execution

Asynchronously multiple and divide two big fractions

```
CompletableFuture<BF> compF1 =
    CompletableFuture
    .supplyAsync(() ->
       /* multiply two BFs. */);

CompletableFuture<BF> compF2 =
    CompletableFuture
    .supplyAsync(() ->
       /* divide two BFs. */);

compF1
   .thenCombine(compF2,
                BigFraction::add)

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

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results
    - Returns a future containing the result of the action
    - Used to "join" two paths of asynchronous execution

thenCombine()'s action is triggered when its two associated futures complete.

```
CompletableFuture<BF> compF1 =
    CompletableFuture
    .supplyAsync(() ->
       /* multiply two BFs. */);

CompletableFuture<BF> compF2 =
    CompletableFuture
    .supplyAsync(() ->
       /* divide two BFs. */);

compF1
   .thenCombine(compF2,
                BigFraction::add)

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

Methods Triggered by Completion of Both of Two Stages

- Methods triggered by completion of both of two previous stages
  - thenCombine()
    - Applies a bifunction action to two previous stages' results
    - Returns a future containing the result of the action
    - Used to "join" two paths of asynchronous execution

```
CompletableFuture<BF> compF1 =
    CompletableFuture
    .supplyAsync(() ->
       /* multiply two BFs. */);

CompletableFuture<BF> compF2 =
    CompletableFuture
    .supplyAsync(() ->
       /* divide two BFs. */);

compF1
   .thenCombine(compF2,
                BigFraction::add)

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

Print out the results

End of Advanced Java Completable Future Features: Two-Stage Completion Methods, Part I

The End

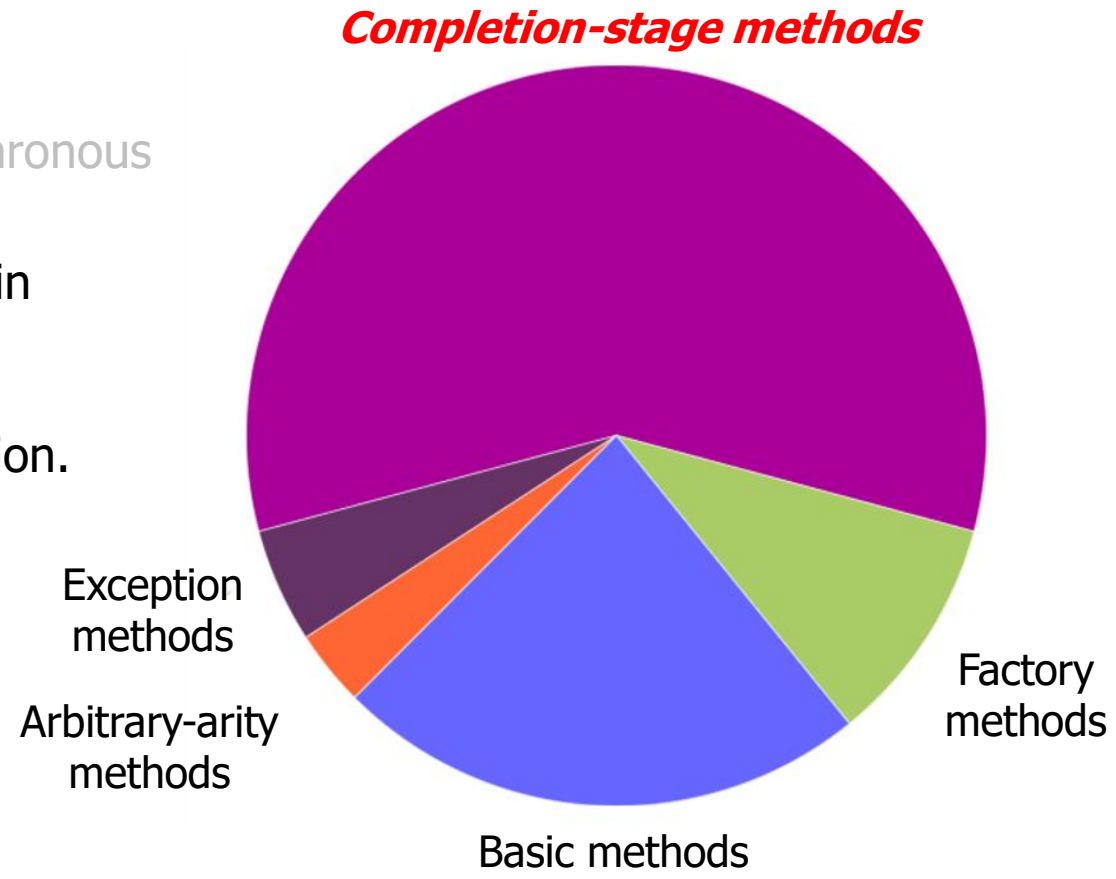
Advanced Java Completable Future Features: Two-Stage Completion Methods

Part II

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
    - Method grouping
    - Single-stage methods
    - Two-stage methods (and)
    - Two-stage methods (or)

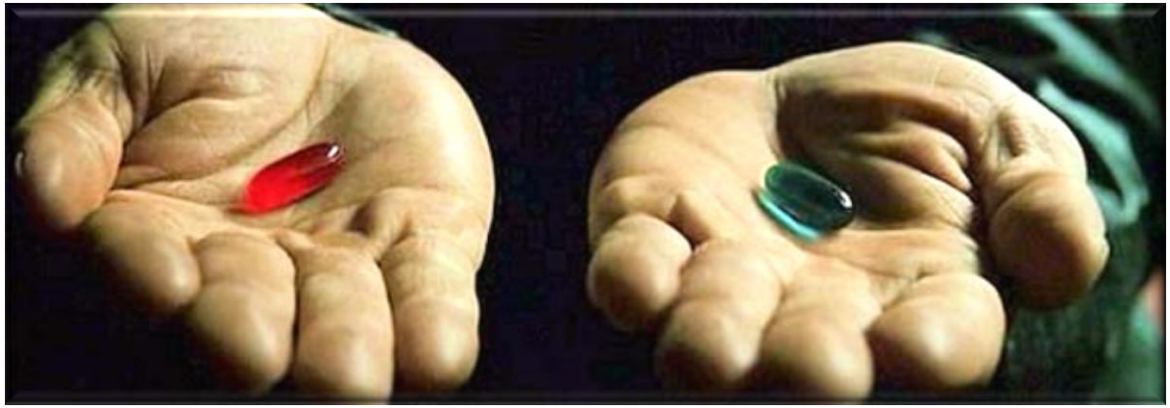


Methods Triggered by Completion of Two Stages

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()

```
CompletableFuture<Void> acceptEither
  (CompletionStage<? Extends T>
                        other,
   Consumer<? super T> action)
{ ... }
```

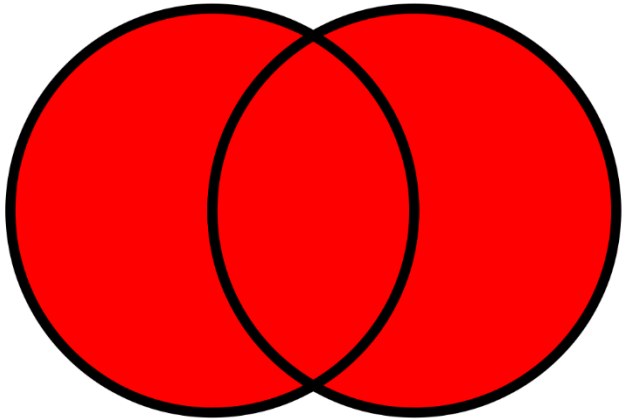


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#acceptEither

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stages' results

```
CompletableFuture<Void> acceptEither
  (CompletionStage<? Extends T>
                          other,
   Consumer<? super T> action)
{ ... }
```



See en.wikipedia.org/wiki/Logical_disjunction

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stages' results
      - Two futures are used here:
        - The future used to invoke acceptEither()
        - The "other" future passed to acceptEither()

```
CompletableFuture<Void> acceptEither
 (CompletionStage<? Extends T>
                        other,
  Consumer<? super T> action)
{ ... }
```

See en.wikipedia.org/wiki/Logical_disjunction

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stages' results
    - Returns a future to Void

```
CompletableFuture<Void> acceptEither
  (CompletionStage<? Extends T>
                          other,
   Consumer<? super T> action)
{ ... }
```

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stages' results
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

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

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

Create two completable futures that will contain the results of sorting the list using two different algorithms in two different threads

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stages' results
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

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

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

quickSortF.acceptEither
  (mergeSortF, results -> results
    .forEach(fraction ->
             System.out.println
              (fraction
               .toMixedString())));
```

This method is invoked when either quickSortF or mergeSortF complete.

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stages' results
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

```
CompletableFuture<List<BigFraction>>
  quickSortF = Comple ableFu e
    .supplyAsync(() ->

CompletableFutur >
  mergeSortF = C
    .supplyAsync
                  me eSo rt(list) ;

quickSortF.acceptEither
  (mergeSortF, results -> results
    .forEach(fraction ->
               System.out.println
                 (fraction
                  .toMixedString())));
```

acceptEither() does *not* cancel the second future after the first one completes.

Methods Triggered by Completion of Either of Two Stages

- Methods triggered by completion of either of two previous stages
  - acceptEither()
    - Applies a consumer action that handles either of the previous stages' results
    - Returns a future to Void
    - Often used at the end of a chain of completion stages

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

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

quickSortF.acceptEither
  (mergeSortF, results -> results
    .forEach(fraction ->
               System.out.println
                (fraction
                 .toMixedString())));
```

Printout sorted results from the sorting routine finishing first

Advanced Java Completable Future
Features: Two Stage Completion Methods, Part II

The End

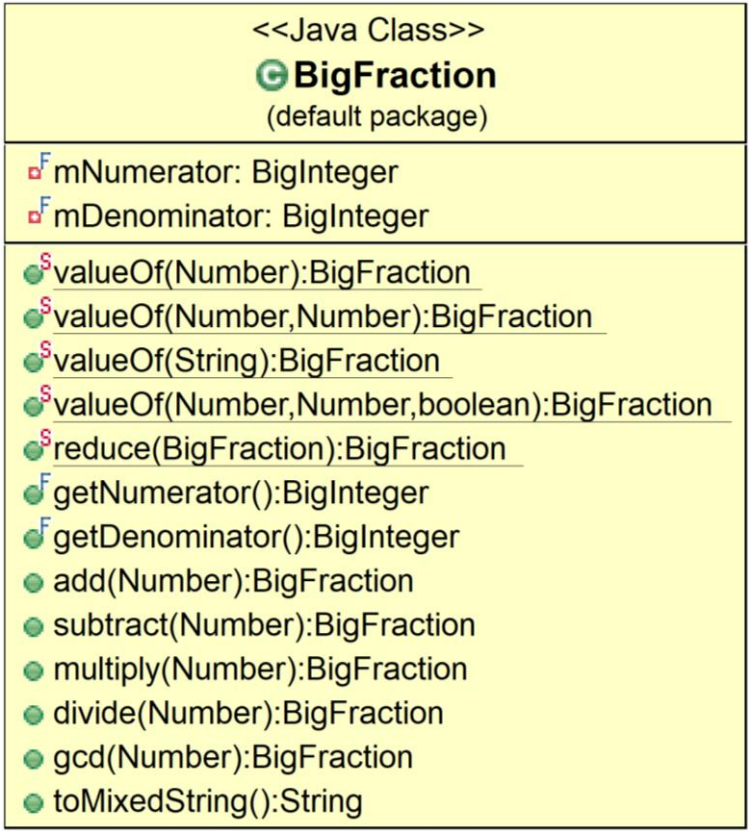
Advanced Java Completable Future Features

Applying Completion-Stage Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
    - Method grouping
    - Single-stage methods
    - Two-stage methods (and)
    - Two-stage methods (or)
    - Apply these methods



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

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 completable futures.

```
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 completable futures.

```
static void testFractionMultiplications1() {
   ...
   Stream.generate(() -> makeBigFraction(new Random(), false))

         .limit(sMAX_FRACTIONS)

         .map(reduceAndMultiplyFraction)

         .collect(FuturesCollector.toFuture())

         .thenAccept(ex8::sortAndPrintList);
}
```

Generate a bounded number of large, random, and unreduced fractions

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#generate

Applying Completable Future Completion-Stage Methods

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

```
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 and 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 completable futures.

```
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);
}
```

A random number generator and a flag indicating whether to reduce the BigFraction

Applying Completable Future Completion-Stage Methods

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

```
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 completable futures.

```
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 denominator by dividing numerator by a random number between 1 and 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 completable futures.

```
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 with the numerator and 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 completable futures.

```
static void testFractionMultiplications1() {
   ...
   Stream.generate(() -> makeBigFraction(new Random(), false))

         .limit(sMAX_FRACTIONS)

         .map(reduceAndMultiplyFraction)

         .collect(FuturesCollector.toFuture())

         .thenAccept(ex8::sortAndPrintList);
}
```

Reduce and 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 completable futures.

```
static void testFractionMultiplications1() {
  Function<BigFraction, CompletableFuture<BigFraction>>
    reduceAndMultiplyFraction = unreducedFrac ->
      CompletableFuture


        .supplyAsync(() -> BigFraction.reduce(unreducedFrac))

        .thenCompose(reducedFrac -> CompletableFuture
                     .supplyAsync(() -> reducedFrac
                                  .multiply(sBigFraction)));

  ...
```

Lambda function that asynchronously reduces and multiplies big 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 completable futures.

```
static void testFractionMultiplications1() {
  Function<BigFraction, CompletableFuture<BigFraction>>
    reduceAndMultiplyFraction = unreducedFrac ->
      CompletableFuture

        .supplyAsync(() -> BigFraction.reduce(unreducedFrac))

        .thenCompose(reducedFrac -> CompletableFuture
                     .supplyAsync(() -> reducedFrac
                                  .multiply(sBigFraction)));

  ...
```

Asynchronously reduce a 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 completable futures.

```
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 Completable Future Completion-Stage Methods

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

```
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 Completable Future Completion-Stage Methods

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

```
static void testFractionMultiplications2() {
  Function<BigFraction, CompletableFuture<BigFraction>>
    reduceAndMultiplyFraction = unreducedFrac ->
      CompletableFuture


        .supplyAsync(() -> BigFraction.reduce(unreducedFrac))

        .thenApplyAsync(reducedFrac ->
                        reducedFrac.multiply(sBigFraction)));
  ...
```

thenApplyAsync() is an alternative means to avoid calling supplyAsync() again.

Applying Completable Future Completion-Stage Methods

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

```
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 asynchronous operations on big 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 completable futures.

```
static void testFractionMultiplications1() {
   ...
   Stream.generate(() -> makeBigFraction(new Random(), false))

         .limit(sMAX_FRACTIONS)

         .map(reduceAndMultiplyFraction)

         .collect(FuturesCollector.toFuture())

         .thenAccept(ex8::sortAndPrintList);
}
```

Return a single future to a list of big fractions being reduced and multiplied asynchronously

See lesson on "*Advanced Java Completable Future Features: Implementing FuturesCollector*"

Applying Completable Future Completion-Stage Methods

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

```
static void testFractionMultiplications1() {
   ...
   Stream.generate(() -> makeBigFraction(new Random(), false))

         .limit(sMAX_FRACTIONS)

         .map(reduceAndMultiplyFraction)

         .collect(FuturesCollector.toFuture())

         .thenAccept(ex8::sortAndPrintList);
}
```

Sort and print results when all asynchronous 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 completable futures.

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

Sort and print a list of reduced and multiplied big 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 completable futures.

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


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

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

Asynchronously apply quick sort and merge sort!

```
  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 completable futures.

```
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 completable futures.

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


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

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

If future is already completed, the action runs in the thread that registered the action.

```
  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 completable futures.

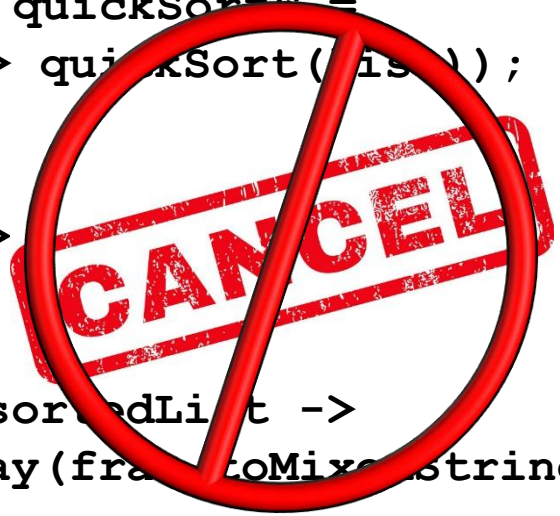
```
static void sortAndPrintList(List<BigFraction> list) {


  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 Completable Future Completion-Stage Methods

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

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


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

  CompletableFuture<List<BigFraction>>
    CompletableFuture.supplyAsync(() ->


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



acceptEither() does *not* cancel the second future after the first one completes.

The End

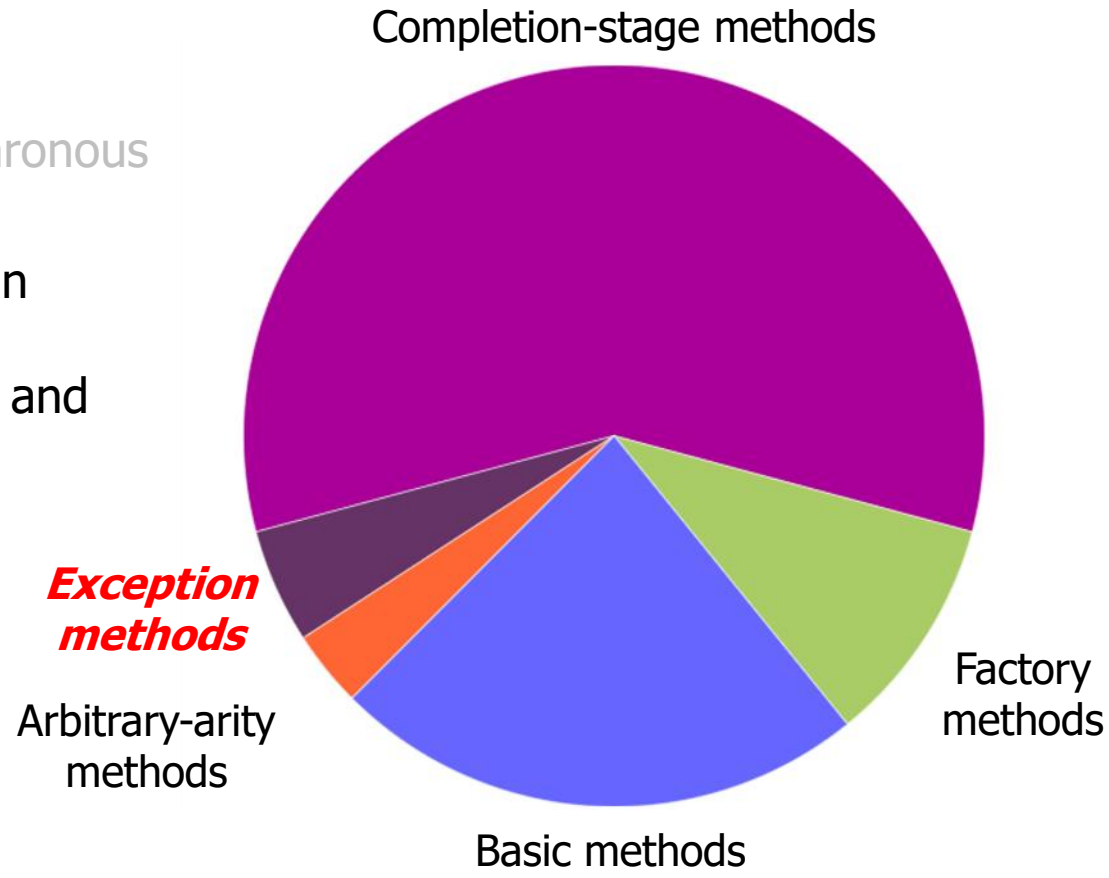
Advanced Java Completable Future Features

Handling Runtime Exceptions

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

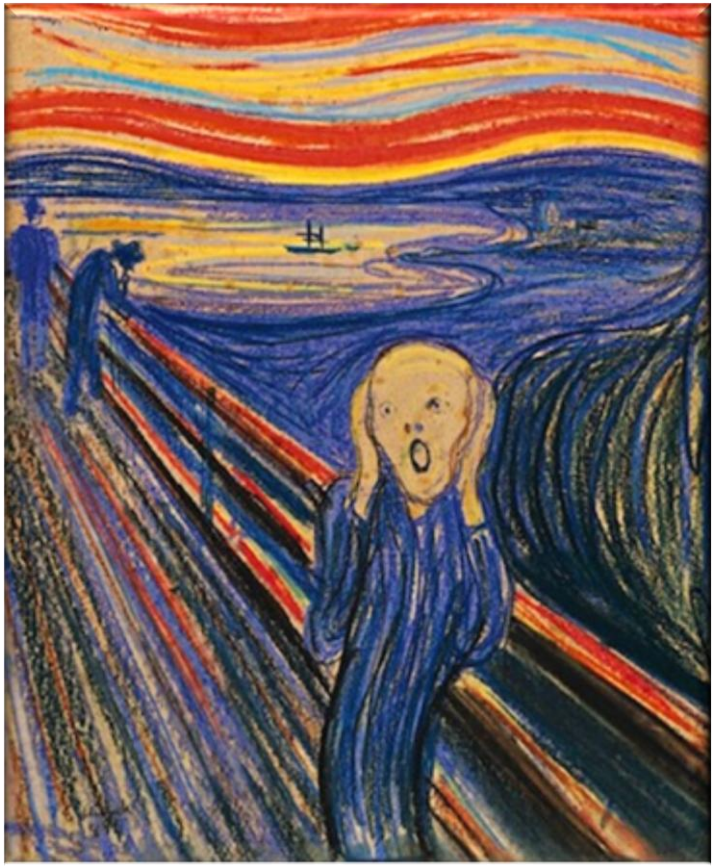
- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
    - Method grouping
    - Single-stage methods
    - Two-stage methods (and)
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



Handling Runtime Exceptions in Completion Stages

Handling Runtime Exceptions in Completion Stages

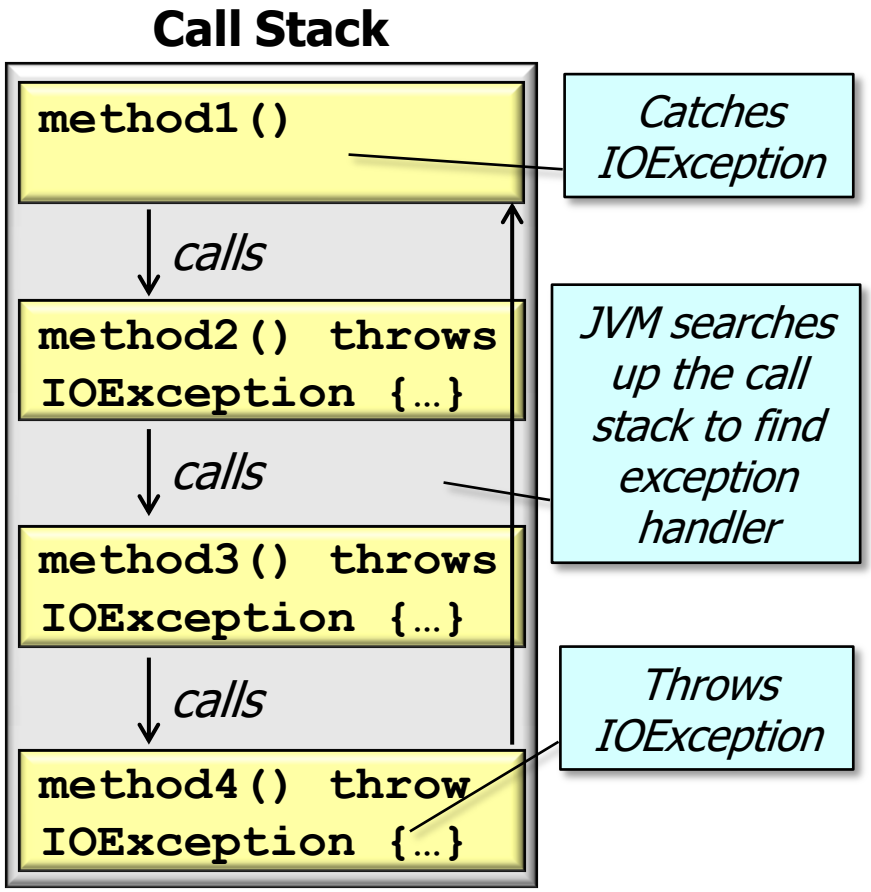
- Exception handling is more complex for asynchronous computations than for synchronous computations.



See blog.lightstreamer.com/2014/07/exception-handling-in-asynchronous-java.html

Handling Runtime Exceptions in Completion Stages

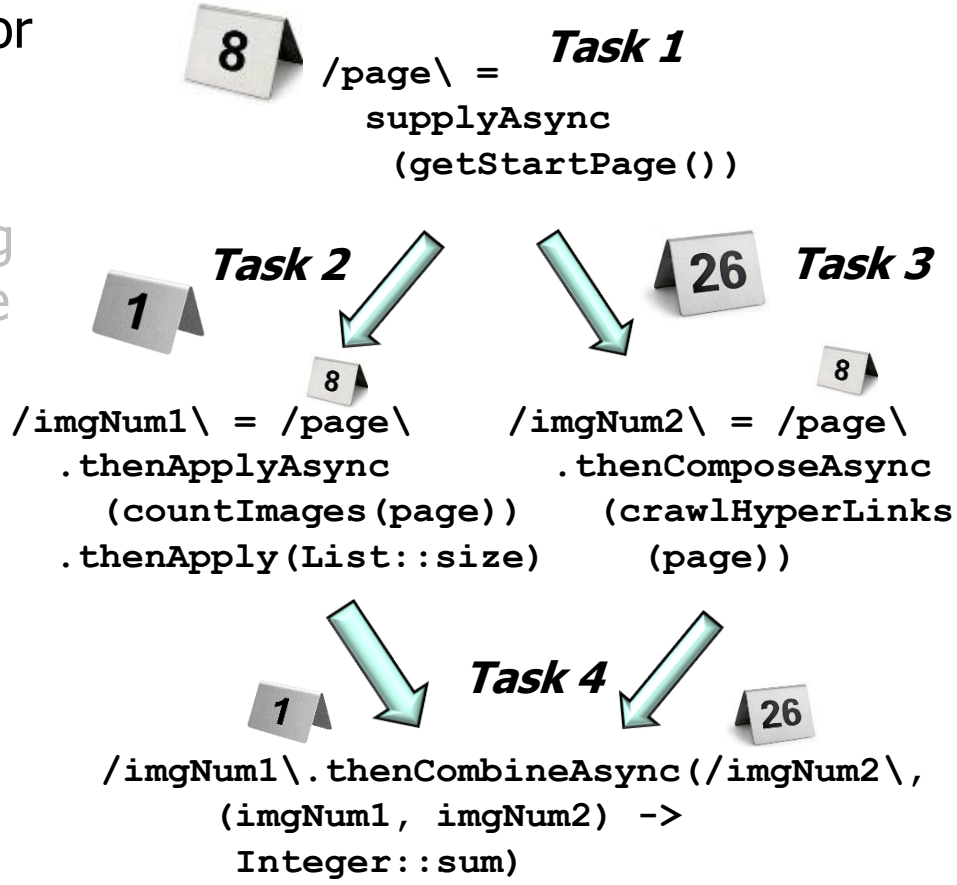
- Exception handling is more complex for asynchronous computations than for synchronous computations.
  - The conventional exception handling model propagates exceptions up the runtime call stack synchronously.



See en.wikipedia.org/wiki/Exception_handling

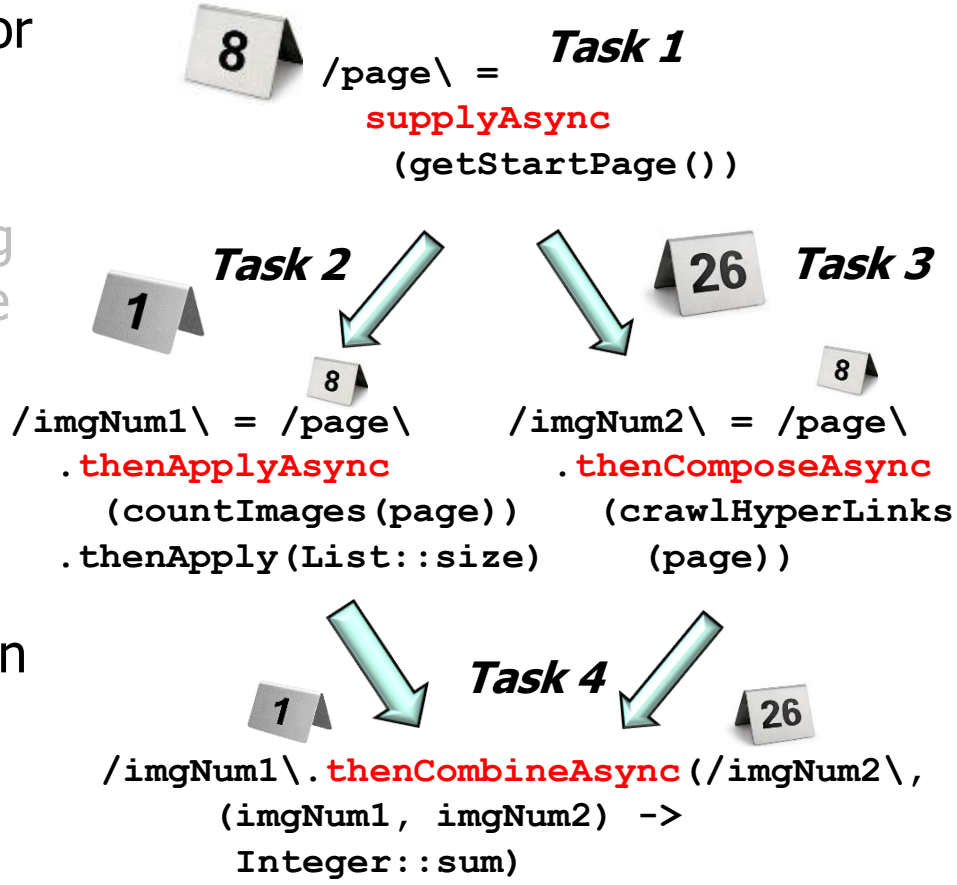
Handling Runtime Exceptions in Completion Stages

- Exception handling is more complex for asynchronous computations than for synchronous computations.
  - The conventional exception handling model propagates exceptions up the runtime call stack synchronously.
  - However, completable futures that run asynchronously don't conform to a conventional call stack model.



Handling Runtime Exceptions in Completion Stages

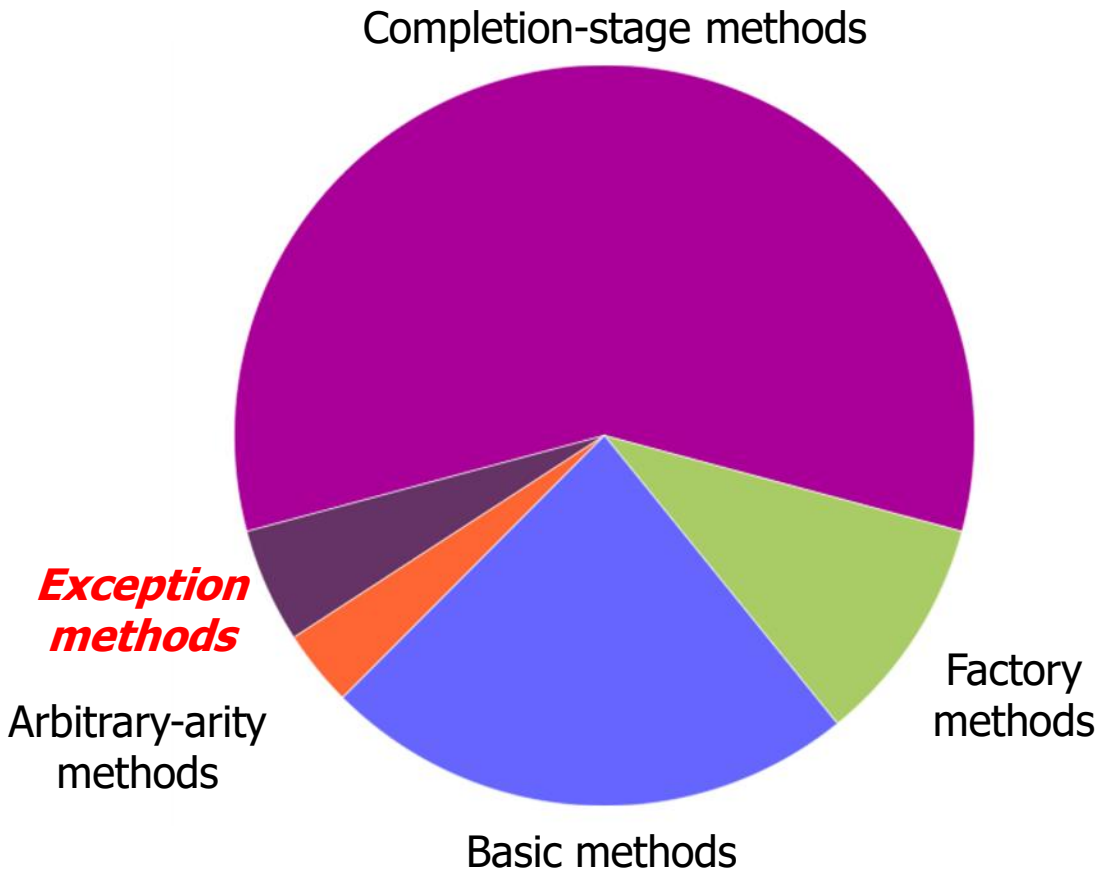
- Exception handling is more complex for asynchronous computations than for synchronous computations.
  - The conventional exception handling model propagates exceptions up the runtime call stack synchronously.
  - However, completable futures that run asynchronously don't conform to a conventional call stack model.
    - Completion-stage methods can run in different worker threads!



See suryanarayanjena.wordpress.com/async-methods-in-completablefuture

Handling Runtime Exceptions in Completion Stages

- Completion-stage methods handle runtime exceptions that occur asynchronously.



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 that occur asynchronously.

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 and 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

- Completion-stage methods handle runtime exceptions that occur asynchronously.

Methods	Params	Returns	Behavior
when Complete (Async)	Bi Consumer	Completable Future with result of earlier stage or throws exception	Handle outcome of a stage, whether a result value or an exception
handle (Async)	Bi Function	Completable Future with result of BiFunction	Handle outcome of a stage and return new value
exceptionally (Async)	Function	Completable Future<T>	When exception occurs, replace exception with result value

Added in Java 12

See www.logicbig.com/tutorials/core-java-tutorial/java-12-changes/completion-stage-new-methods.html

Handling Runtime Exceptions in Completion Stages

- This example shows three ways to handle exceptions with completable futures.

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

   ...
```

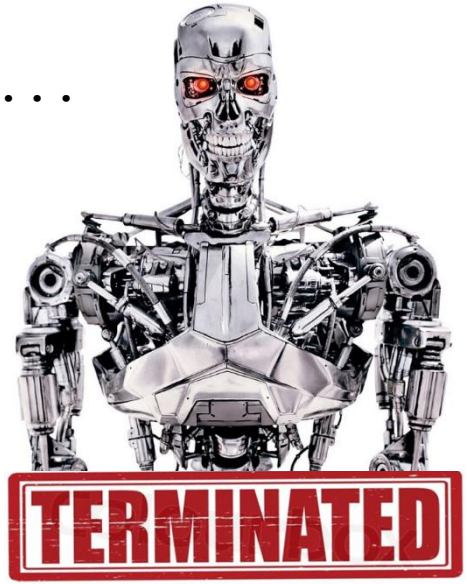
An exception will occur if denominator param is 0!

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

Handling Runtime Exceptions in Completion Stages

- This example shows three ways to handle exceptions with 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

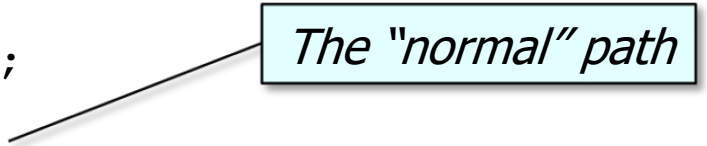
```
CompletableFuture
   .supplyAsync(() ->
                BigFraction.valueOf(100, denominator))

   .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()));
```



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 (and 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

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

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

   .exceptionally(ex -> BigFraction.ZERO)


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

An exception occurs if denominator is 0!

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)
```

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

```
   .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)
```

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

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

exceptionally() is akin to catch() in a Java try/catch block (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 an 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 and 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 an 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 an 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 an 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 an exceptional or normal action

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

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

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

whenComplete() is like Java Streams.peek(), i.e., it has a side-effect, doesn't change the return value, & doesn't swallow the exception

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#peek

The End

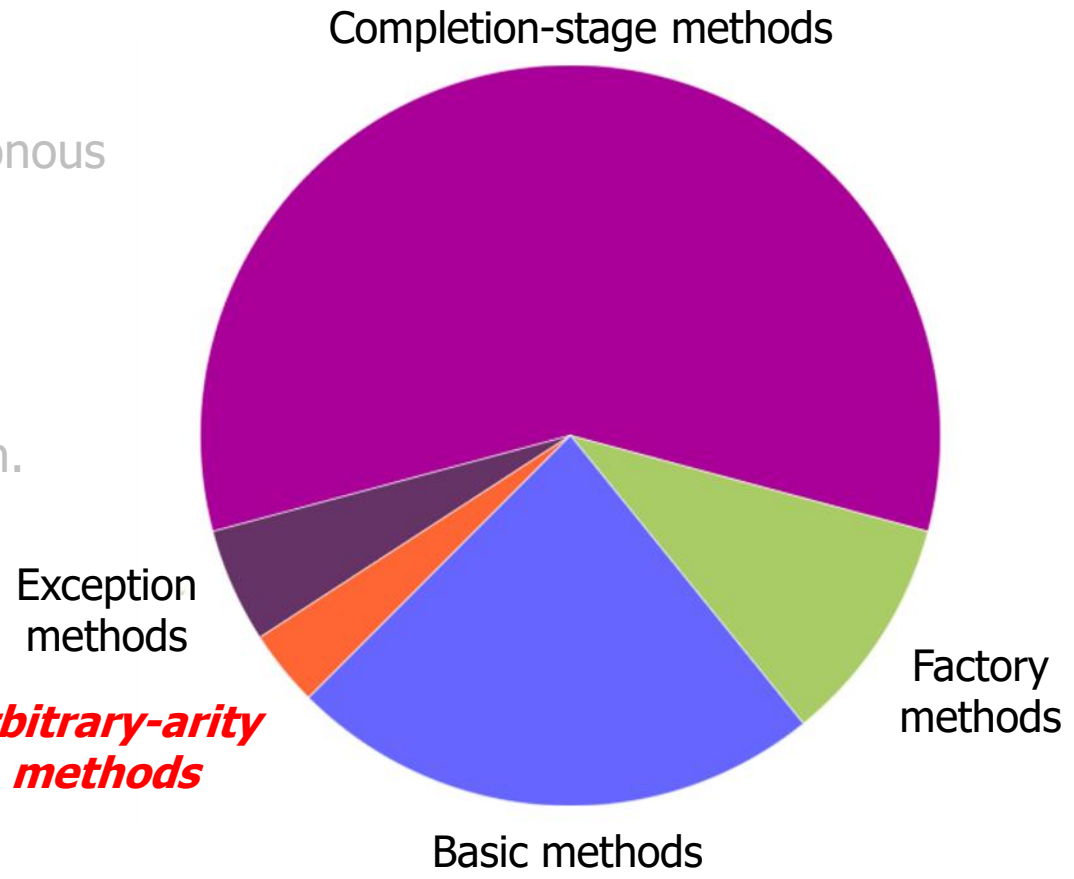
Advanced Java Completable Future Features

Arbitrary-Arity Methods

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
  - Arbitrary-arity methods process futures in bulk.



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

Arbitrary-Arity Methods Process Futures in Bulk

Arbitrary-Arity Methods Process Futures in Bulk

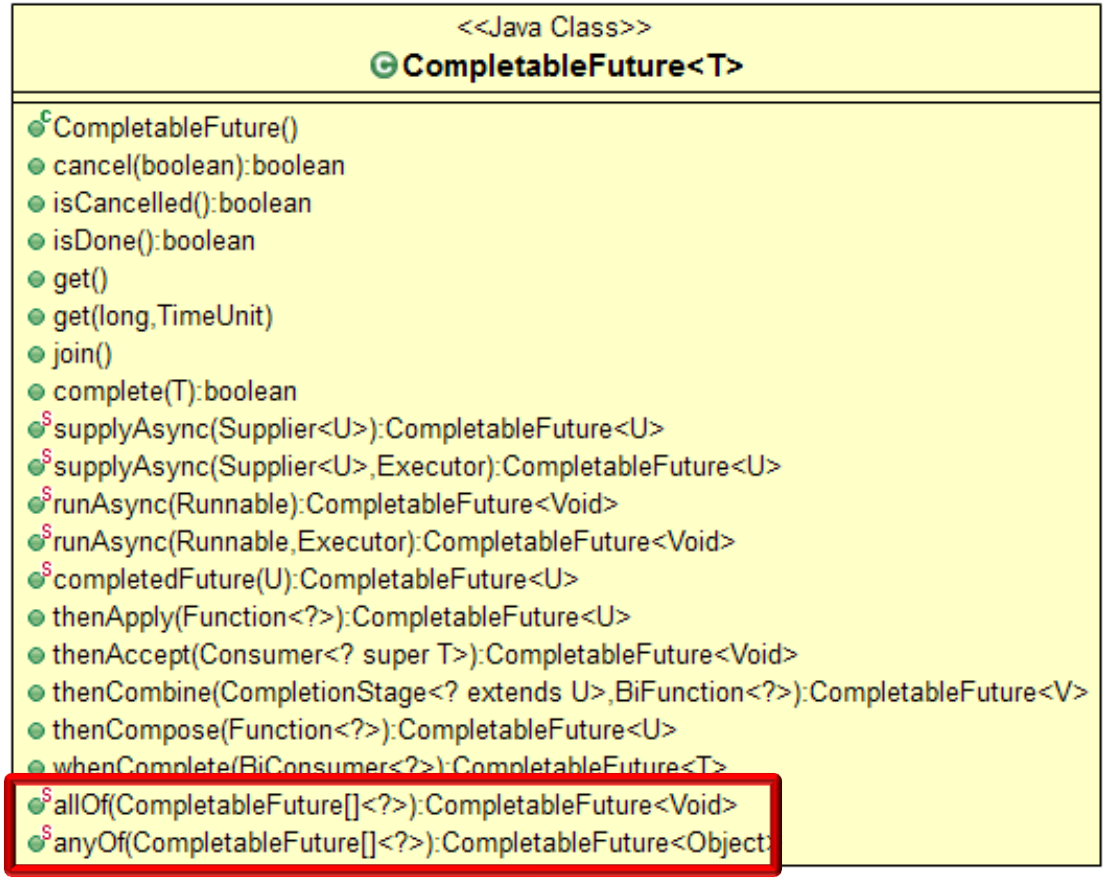
- Arbitrary-arity methods return futures that are triggered after completion of any/all futures.

Methods	Params	Returns	Behavior
allOf	Varargs	CompletableFuture<Void>	Return a future that completes when all futures in params complete
anyOf	Varargs	CompletableFuture<Void>	Return a future that completes when any future in params complete

See en.wikipedia.org/wiki/Arity

Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods return futures that are triggered after completion of any/all futures.
  - The returned future can be used to wait for any or all of *N* completable futures in an array to complete.



Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods return futures that are triggered after completion of any/all futures.
  - The returned future can be used to wait for any or all of *N* completable futures in an array to complete.
    - This "wait" usually doesn't block, but instead uses completion stage methods.

<<Java Class>>

CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods return futures that are triggered after completion of any/all futures.
  - The returned future can be used to wait for any or all of *N* completable futures in an array to complete.
  - We focus on allOf(), which is like thenCombine() on steroids!

<<Java Class>>

CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- **allOf(CompletableFuture[]<?>):CompletableFuture<Void>**
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

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

Arbitrary-Arity Methods Process Futures in Bulk

- These arbitrary-arity methods are hard to program without using wrappers

<<Java Class>>

CompletableFuture<T>

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long,TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>
- supplyAsync(Supplier<U>,Executor):CompletableFuture<U>
- runAsync(Runnable):CompletableFuture<Void>
- runAsync(Runnable,Executor):CompletableFuture<Void>
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

See next lesson on "Advanced Java CompletableFuture Features: Arbitrary-Arity Methods"

The End

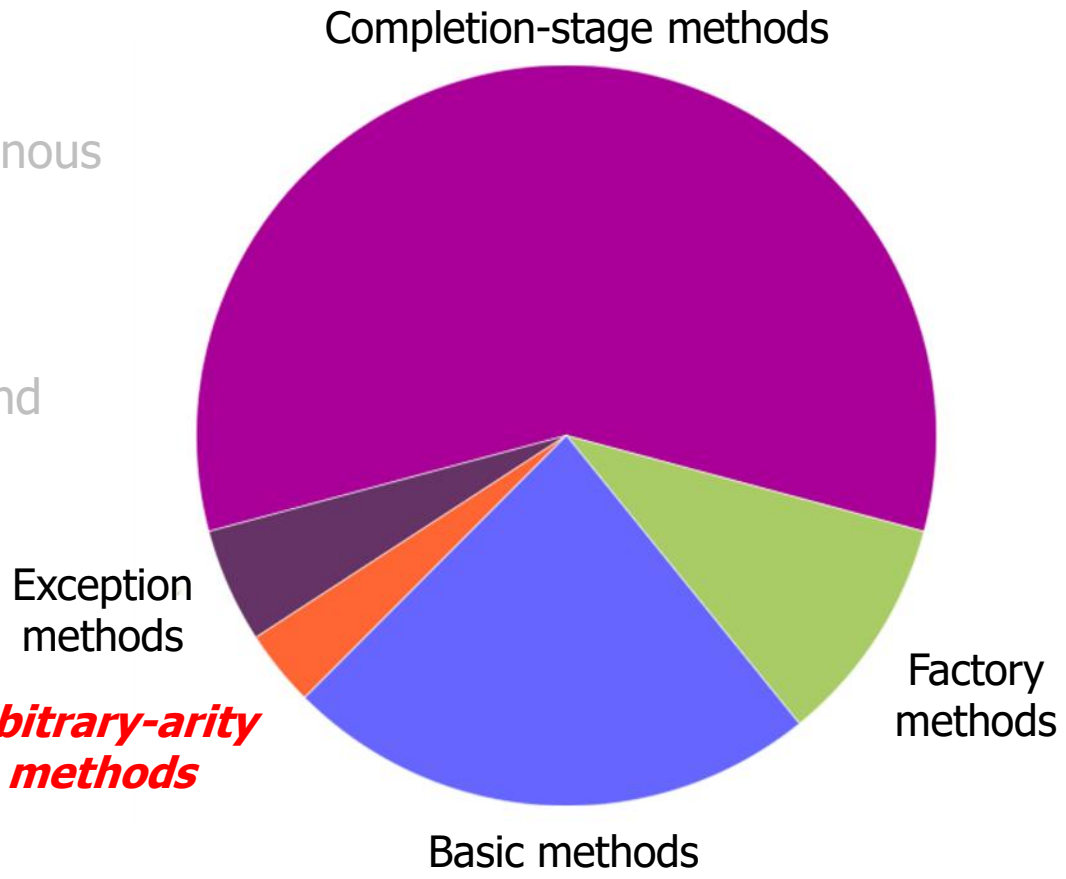
Advanced Java Completable Future Features

Implementing FuturesCollector

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand advanced features of completable futures
  - Factory methods initiate asynchronous computations.
  - Completion-stage methods chain together actions to perform asynchronous result processing and composition.
  - Arbitrary-arity methods process futures in bulk.
    - Provide a wrapper for the allOf() method



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

Implementing the FuturesCollector Class

Implementing the FuturesCollector Class

- FuturesCollector returns a completable future to a list of big fractions that are being reduced and multiplied asynchronously.

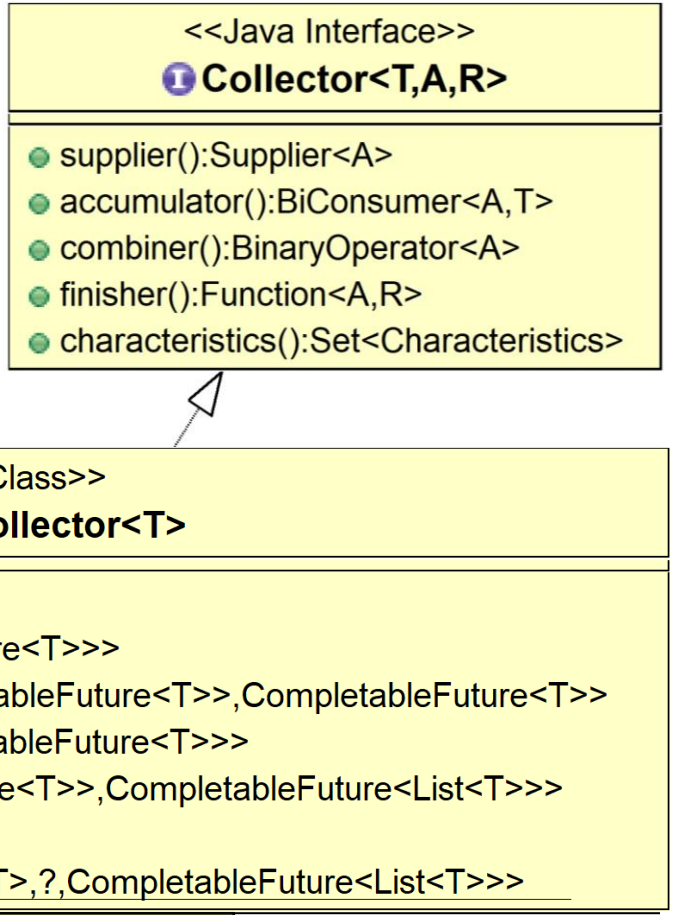
```
static void testFractionMultiplications1() {
   ...
   Stream.generate(() -> makeBigFraction(new Random(), false))

         .limit(sMAX_FRACTIONS)

         .map(reduceAndMultiplyFractions)

         .collect(FuturesCollector.toFuture())

         .thenAccept(this::sortAndPrintList);
}
```

collect() converts a stream of completable futures into a single completable future.

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

Implementing the FuturesCollector Class

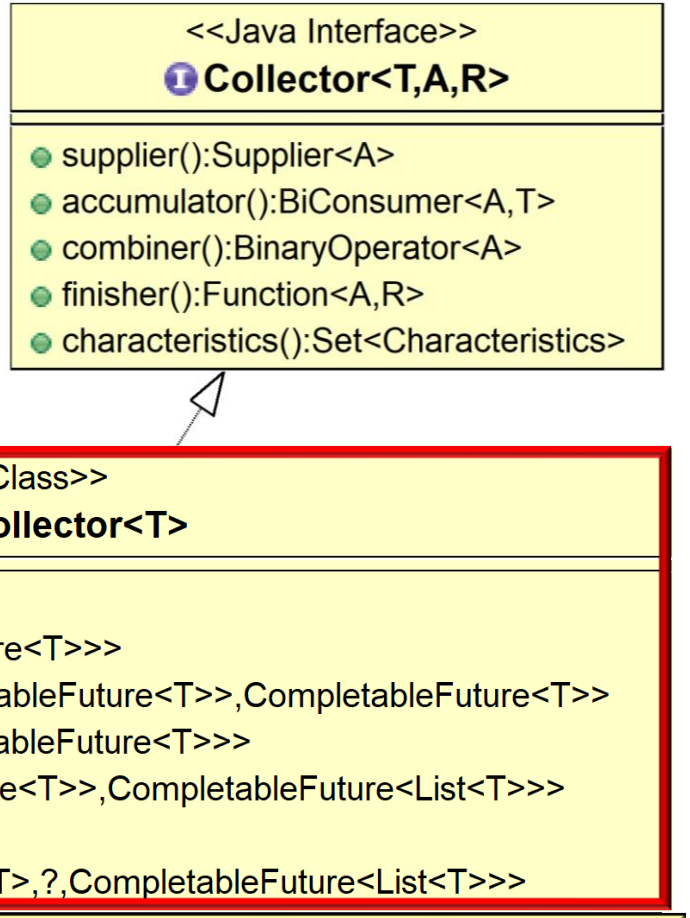
- FuturesCollector provides a wrapper for allOf().



See Java8/ex8/utils/FuturesCollector.java

Implementing the FuturesCollector Class

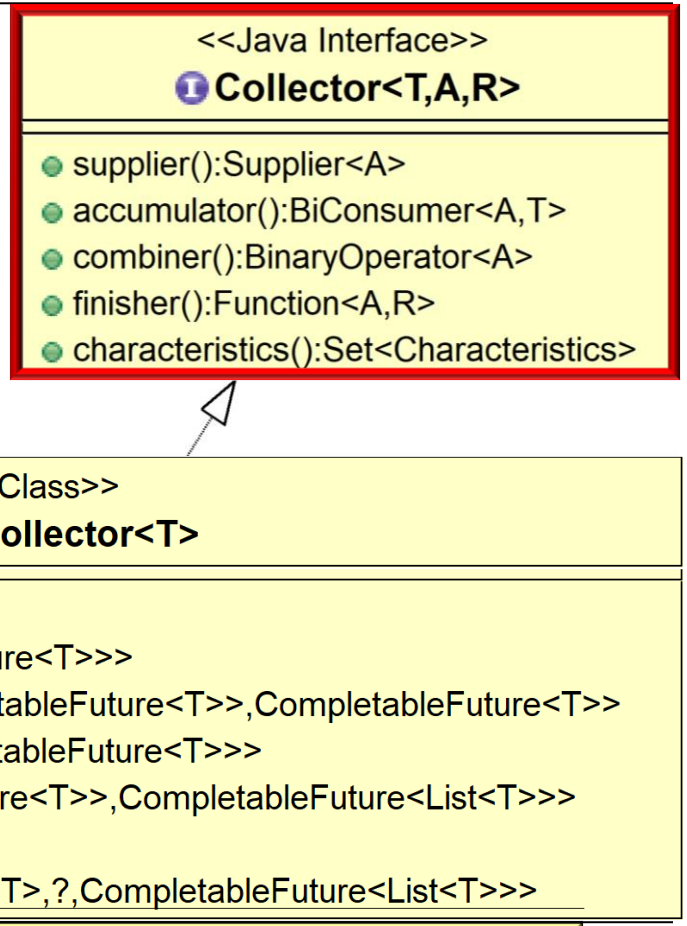
- FuturesCollector provides a wrapper for allOf().
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete



FuturesCollector is a nonconcurrent collector (supports parallel and sequential streams).

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete
  - Implements the Collector interface that accumulates input elements into a mutable result container



See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

<<Java Interface>>
Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

<<Java Class>>
FuturesCollector<T>

- FuturesCollector()
- supplier():Supplier<List<CompletableFuture<T>>>
- accumulator():BiConsumer<List<CompletableFuture<T>>,CompletableFuture<T>>
- combiner():BinaryOperator<List<CompletableFuture<T>>>
- finisher():Function<List<CompletableFuture<T>>,CompletableFuture<List<T>>>
- characteristics():Set
- toFuture():Collector<CompletableFuture<T>,?,CompletableFuture<List<T>>>

FuturesCollector provides a powerful wrapper for some complex code!

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
       implements Collector<CompletableFuture<T>,
                            List<CompletableFuture<T>>,
                            CompletableFuture<List<T>>> {
   ...
```

Implements a custom collector

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

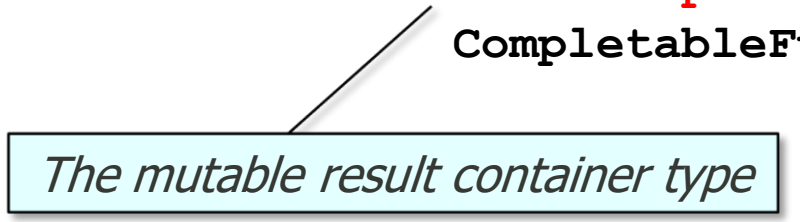
```
public class FuturesCollector<T>
       implements Collector<CompletableFuture<T>,
                            List<CompletableFuture<T>>,
                            CompletableFuture<List<T>>> {
   ...
```

The type of input elements in the stream

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

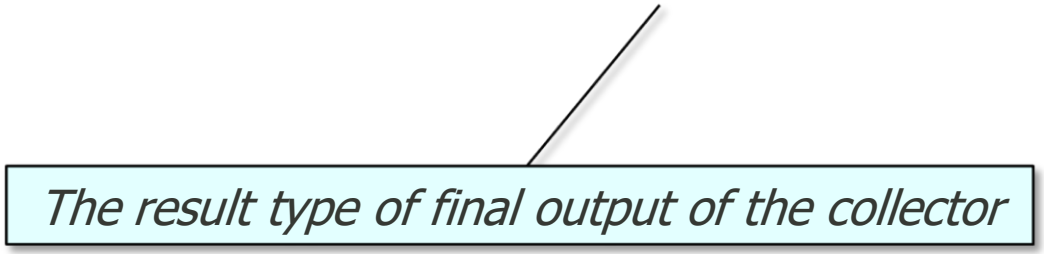
```
public class FuturesCollector<T>
       implements Collector<CompletableFuture<T>,
                            List<CompletableFuture<T>>,
                            CompletableFuture<List<T>>> {
   ...
```



Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
       implements Collector<CompletableFuture<T>,
                            List<CompletableFuture<T>>,
                            CompletableFuture<List<T>>> {
   ...
```



Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
       implements Collector<CompletableFuture<T>,
                            List<CompletableFuture<T>>,
                            CompletableFuture<List<T>>> {
  public Supplier<List<CompletableFuture<T>>> supplier() {
    return ArrayList::new;
  }

  public BiConsumer<List<CompletableFuture<T>>,
                    CompletableFuture<T>> accumulator()
  { return List::add; }
  ...
```

This factory method returns a supplier used by the Java streams collector framework to create a new mutable array list container.

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
       implements Collector<CompletableFuture<T>,
                            List<CompletableFuture<T>>,
                            CompletableFuture<List<T>>> {
  public Supplier<List<CompletableFuture<T>>> supplier() {
    return ArrayList::new;
  }

  public BiConsumer<List<CompletableFuture<T>>,
                    CompletableFuture<T>> accumulator()
  { return List::add; }
  ...
```

This mutable result container stores a list of completable futures of type T.

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
       implements Collector<CompletableFuture<T>,
                            List<CompletableFuture<T>>,
                            CompletableFuture<List<T>>> {
  public Supplier<List<CompletableFuture<T>>> supplier() {
    return ArrayList::new;
  }
```

This factory method returns a bi-consumer used by the Java streams collector framework to add a new completable future into the mutable array list container.

```
  public BiConsumer<List<CompletableFuture<T>>,
                    CompletableFuture<T>> accumulator()
  { return List::add; }
  ...
```

This method is only ever called in a single thread (so no locks are needed).

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public BinaryOperator<List<CompletableFuture<T>>> combiner() {
    return (List<CompletableFuture<T>> one,
            List<CompletableFuture<T>> another) -> {
      one.addAll(another);
      return one;
    };
  }
  ...
```

This factory method returns a binary operator that merges two partial array list results into a single array list (only relevant for parallel streams).

This method is only ever called in a single thread (so no locks are needed).

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<List<T>>> finisher() {
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))
      .thenApply(v -> futures.stream()
                             .map(CompletableFuture::join)
                             .collect(toList()));
  }
  ...
```

This factory method returns a function used by the Java streams collector framework to transform the array list multiple result container to the completable future result type.

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<List<T>>> finisher() {
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                             .map(CompletableFuture::join)
                             .collect(toList()));
  }
  ...
```

Reference to the mutable result contain, which is an ArrayList

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<List<T>>> finisher() {
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                             .map(CompletableFuture::join)
                             .collect(toList()));
  }
  ...
```

Convert the list of futures to an array of futures and pass to allOf() to obtain a future that will complete when all futures complete

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<List<T>>> finisher() {
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))
      .thenApply(v -> futures.stream()
                             .map(CompletableFuture::join)
                             .collect(toList()));
  }
  ...
```

When all futures have completed, get a single future to a list of joined elements of type T

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<List<T>>> finisher() {
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                       .map(CompletableFuture::join)
                       .collect(toList()));
  }
  ...
```

Convert the array list of futures into a stream of futures

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Function<List<CompletableFuture<T>>,
                  CompletableFuture<List<T>>> finisher() {
    return futures -> CompletableFuture
      .allOf(futures.toArray(new CompletableFuture[0]))

      .thenApply(v -> futures.stream()
                             .map(CompletableFuture::join)
                             .collect(toList()));
  }
  ...
```

This call to join() will never block!

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
   ...
   public Function<List<CompletableFuture<T>>,
                                CompletableFuture<List<T>>> finisher() {
     return futures -> CompletableFuture
       .allOf(futures.toArray(new CompletableFuture[0]))

       .thenApply(v -> futures.stream()
                                    .map(CompletableFuture::join)
                                    .collect(toList()));
   }
   ...
```

Return a future to a list of elements of T

Implementing the FuturesCollector Class

- FuturesCollector is used to return a completable future to a list of big fractions that are being reduced and multiplied asynchronously.

```
static void testFractionMultiplications1() {
   ...
   Stream.generate(() -> makeBigFraction(new Random(), false))

         .limit(sMAX_FRACTIONS)

         .map(reduceAndMultiplyFraction)

         .collect(FuturesCollector.toFuture())

         .thenAccept(this::sortAndPrintList);
}
```

thenAccept() is called only when the future returned from collect() completes.

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

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Set characteristics() {
    return Collections.singleton(Characteristics.UNORDERED);
  }



  public static <T> Collector<CompletableFuture<T>, ?,
                              CompletableFuture<List<T>>>
  toFuture() {
    return new FuturesCollector<>();
  }
}
```

Returns a set indicating the FuturesCollector characteristics

FuturesCollector is thus a *nonconcurrent* collector.

Implementing the FuturesCollector Class

- FuturesCollector provides a wrapper for allOf().

```
public class FuturesCollector<T>
  ...
  public Set characteristics() {
    return Collections.singleton(Characteristics.UNORDERED);
  }

  public static <T> Collector<CompletableFuture<T>, ?,
                              CompletableFuture<List<T>>>
  toFuture() {
    return new FuturesCollector<>();
  }
}
```

This static factory method creates a new FuturesCollector.

The End