Advanced Java CompletableFuture Features: Applying Factory Methods

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

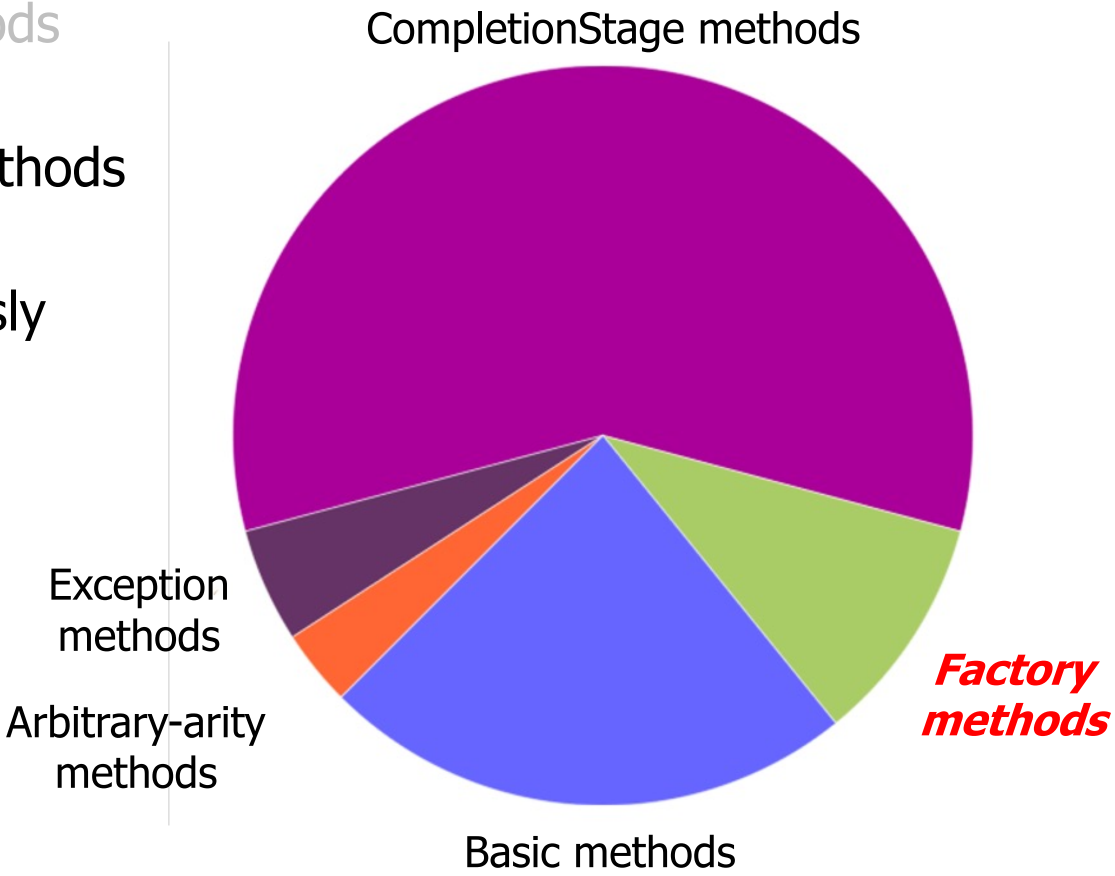
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how factory methods initiate async computations
- Know how to apply factory methods
  - Multiply BigFraction objects concurrently & asynchronously



Learning Objectives in this Part of the Lesson

- Understand how factory methods initiate async computations
- Know how to apply factory methods
  - Multiply BigFraction objects concurrently & asynchronously
  - Evaluate pros & cons of factory methods

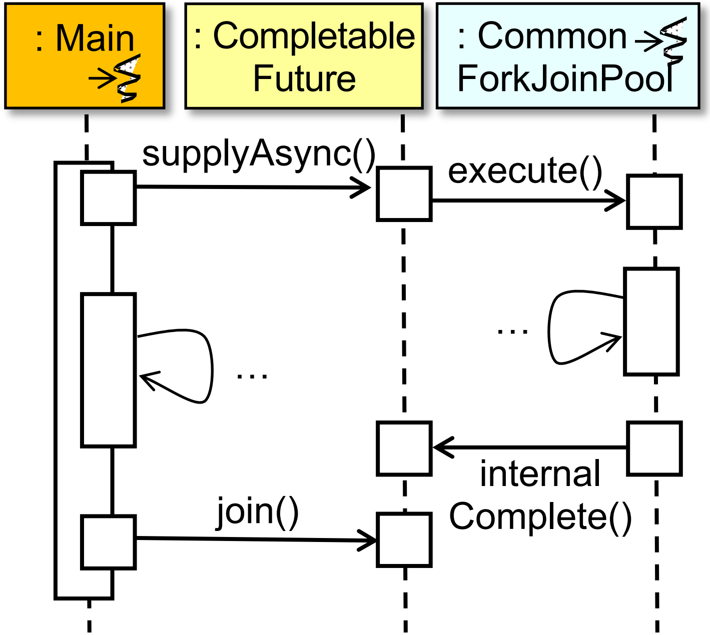
Applying Completable Future Factory Methods

Applying CompletableFuture Factory Methods

- Use supplyAsync() to multiply BigFraction objects

```
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());
```



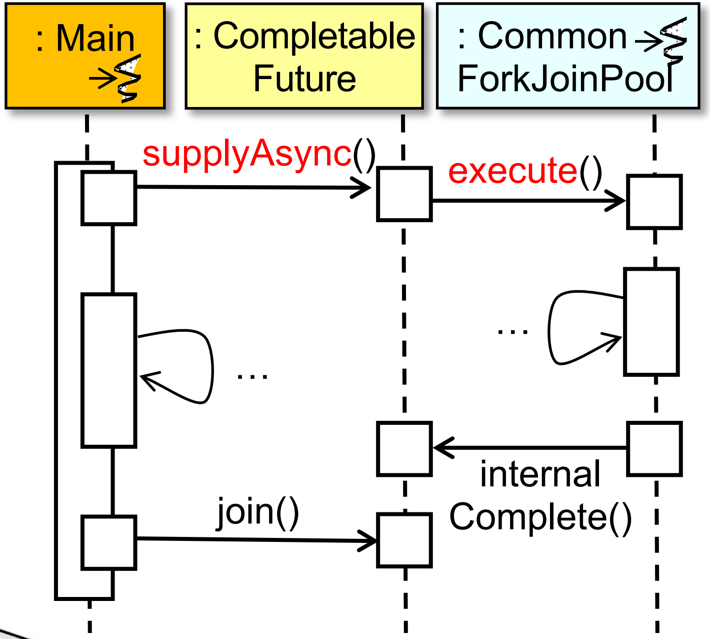
See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

Applying CompletableFuture Factory Methods

- Use supplyAsync() to multiply BigFraction objects

```
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

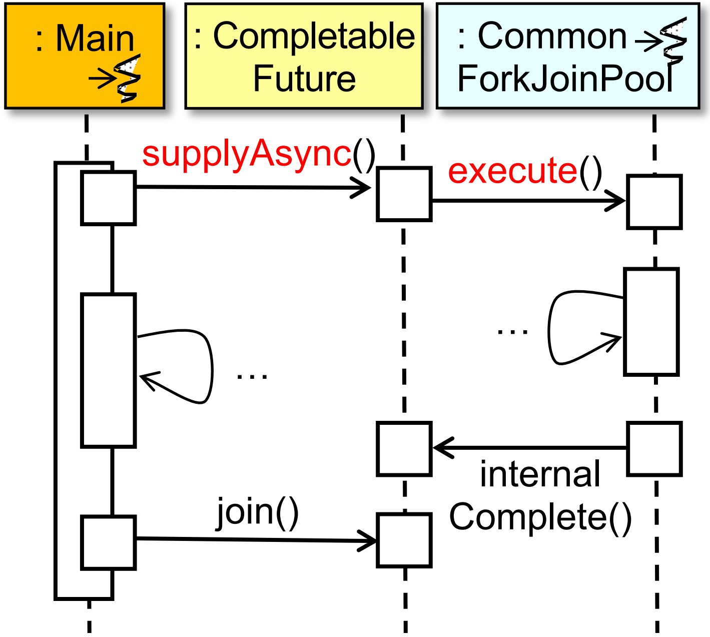
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#supplyAsync

Applying CompletableFuture Factory Methods

- Use supplyAsync() to multiply BigFraction objects

```
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());
```



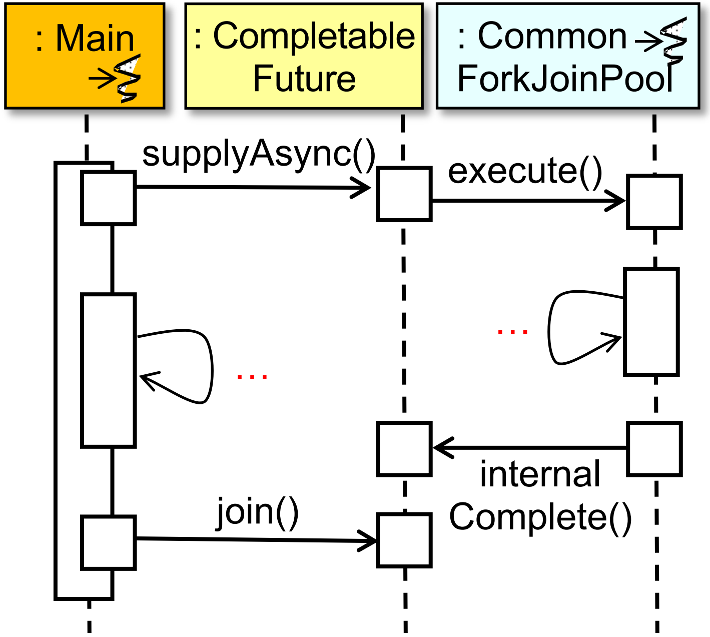
Define a supplier lambda that multiplies two BigFractions

Applying CompletableFuture Factory Methods

- Use supplyAsync() to multiply BigFraction objects

```
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());
```



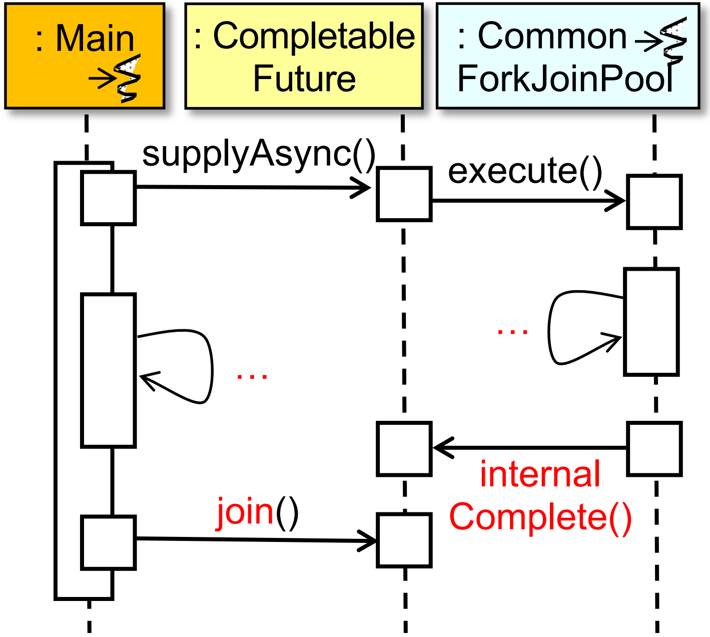
These computations run concurrently

Applying CompletableFuture Factory Methods

- Use supplyAsync() to multiply BigFraction objects

```
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

Evaluating Completable Future Factory Methods

Evaluating CompletableFuture Factory Methods

- Pros of using CompletableFuture.supplyAsync()

Evaluating CompletableFuture Factory Methods

- Pros of using CompletableFuture.supplyAsync()
  - No need to explicitly complete the future since supplyAsync() returns one

```
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());
```

Evaluating CompletableFuture Factory Methods

- Pros of using CompletableFuture.supplyAsync()
  - No need to explicitly complete the future since supplyAsync() returns one
  - Avoids the explicit creation/use of threads

```
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 supplier lambda runs in the Java common fork-join pool

Evaluating CompletableFuture Factory Methods

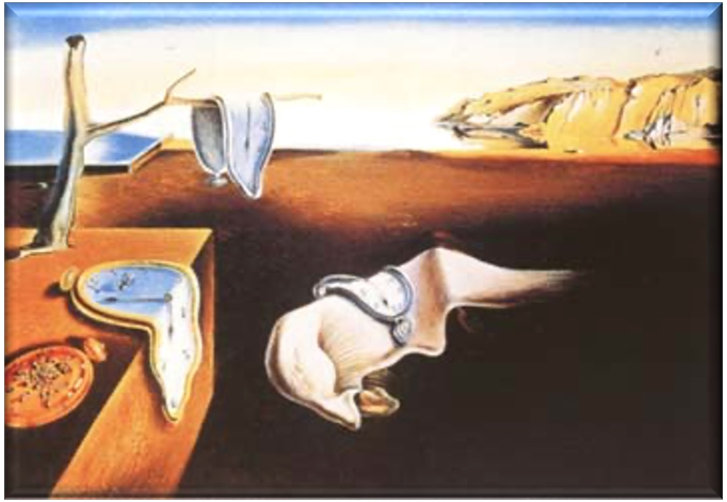
- Cons of using CompletableFuture.supplyAsync()

Evaluating CompletableFuture Factory Methods

- Cons of using CompletableFuture.supplyAsync()
  - We still must fix the problem with calling join()

```
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());
```

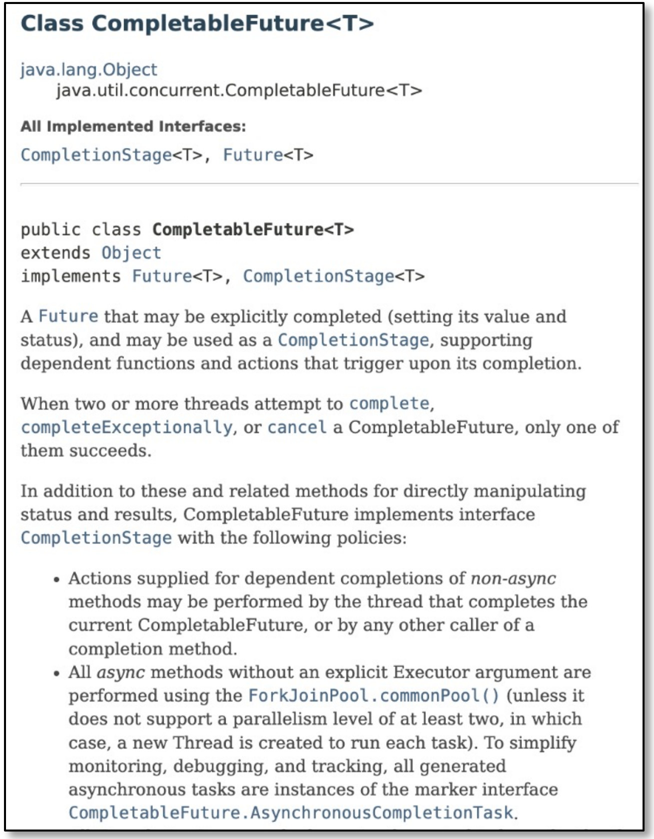


Evaluating CompletableFuture Factory Methods

- Cons of using CompletableFuture.supplyAsync()
  - We still must fix the problem with calling join()

```
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());
```



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:

- Actions supplied for dependent completions of *non-async* methods may be performed by the thread that completes the current CompletableFuture, or by any other caller of a completion method.
- All *async* methods without an explicit Executor argument are performed using the ForkJoinPool.commonPool() (unless it does not support a parallelism level of at least two, in which case, a new Thread is created to run each task). To simplify monitoring, debugging, and tracking, all generated asynchronous tasks are instances of the marker interface CompletableFuture.AsynchronousCompletionTask.

Addressing this issue motivates the advanced Java CompletableFuture features!

End of Advanced Java CompletableFuture Features: Applying Factory Methods