Overview of the BigFraction Case Studies

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 key classes in the Project Reactor API
- Understand key classes in the RxJava API
- Be aware of the structure & functionality of the BigFraction case studies

These case studies demonstrate many Project Reactor & RxJava features
Overview of the BigFraction Class
Overview of the BigFraction Class

• Upcoming lessons show how to apply Project Reactor & RxJava features in the context of a BigFraction class

Overview of the BigFraction Class

- Upcoming lessons show how to apply Project Reactor & RxJava features in the context of a BigFraction class
- Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator

```
BigFraction()
valueOf(Number): BigFraction
valueOf(Number, Number): BigFraction
valueOf(String): BigFraction
valueOf(Number, Number, boolean): BigFraction
reduce(BigFraction): BigFraction
getNumerator(): BigInteger
getDenominator(): BigInteger
add(Number): BigFraction
subtract(Number): BigFraction
multiply(Number): BigFraction
divide(Number): BigFraction
gcd(Number): BigFraction
toMixedString(): String
```

See docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html
Overview of the BigFraction Class

- Upcoming lessons show how to apply Project Reactor & RxJava features in the context of a BigFraction class
- Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
- Factory methods to “reduce” fractions
  - $\frac{44}{55} \rightarrow \frac{4}{5}$
  - $\frac{12}{24} \rightarrow \frac{1}{2}$
  - $\frac{144}{216} \rightarrow \frac{2}{3}$
Overview of the BigFraction Class

- Upcoming lessons show how to apply Project Reactor & RxJava features in the context of a BigFraction class
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods to “reduce” fractions
  - Factory methods to create “non-reduced” fractions (& then reduce them)
    - e.g., 12/24 (→ 1/2)
Overview of the BigFraction Class

- Upcoming lessons show how to apply Project Reactor & RxJava features in the context of a BigFraction class
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods to “reduce” fractions
  - Factory methods to create “non-reduced” fractions (& then reduce them)
  - Arbitrary-precision fraction arithmetic
    - e.g., \( \frac{18}{4} \times \frac{2}{3} = 3 \)
Overview of the BigFraction Class

- Upcoming lessons show how to apply Project Reactor & RxJava features in the context of a BigFraction class
- Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
- Factory methods to “reduce” fractions
- Factory methods to create “non-reduced” fractions (& then reduce them)
- Arbitrary-precision fraction arithmetic
- Create a mixed fraction from an improper fraction
  - e.g., 18/4 → 4 1/2

See www.mathsisfun.com/improper-fractions.html
Overview of the BigFraction Case Studies
Overview of the BigFraction Case Studies

- These case studies show how to reduce & multiply big fractions synchronously, asynchronously, & concurrently using Project Reactor & RxJava framework features.
Overview of the BigFraction Case Studies

- The Project Reactor Mono case studies show how to reduce & multiply big fractions using the following Mono features
  - e.g., fromCallable(), just(), zipWith(), doOnSuccess(), first(), when(), then(),.subscribeOn(), & various thread pools

```java
BigFraction unreducedFraction = makeBigFraction(...);

return Mono
  .fromCallable(() -> BigFraction
      .reduce(unreducedFraction))
  .subscribeOn
      (Schedulers.single())
  .map(result -> result.toMixedString())
  .doOnSuccess(result -> System.out.println("big fraction = " + result + "\n"))
  .then();
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Mono
Overview of the BigFraction Case Studies

- The Project Reactor Flux case studies show how to reduce & multiply big fractions using the following Flux features
  - e.g., fromIterable(), just(), map(), create(), doOnNext(), flatMap(), take(), interval(), subscribeOn(), collectList(), subscribe(), & various thread pools

**Flux**
- `create`
  - (bigFractionEmitter)
- `take`(sMAX_FRACTIONS)
- `flatMap`(unreducedFraction ->
  - reduceAndMultiplyFraction
    - (unreducedFraction, Schedulers.parallel()))
- `collectList()`
- `flatMap`(list ->
  - BigFractionUtils
    - .sortAndPrintList
      - (list, sb));

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flux](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flux)
Overview of the BigFraction Case Studies

• The Project Reactor Flux case studies show how to reduce & multiply big fractions using the following Flux features
  • e.g., fromIterable(), just(), map(), create(), doOnNext(), flatMap(), take(), interval(), subscribeOn(), collectList(), subscribe(), & various thread pools
• They also demonstrate how the Java streams framework can be used together with the Project Reactor framework
Overview of the BigFraction Case Studies

- The RxJava Single case studies show how to reduce & multiply big fractions using the following Single features
  - e.g., fromCallable(), zipWith(), doOnSuccess(), ignoreElement(), subscribeOn(), map(), & the parallel thread pool

```java
BigFraction unreducedFraction = makeBigFraction(...);

return Single
  .fromCallable(() -> BigFraction
    .reduce(unreducedFraction))
  .subscribeOn
    (Schedulers.single())
  .map(result ->
    result.toMixedString())
  .doOnSuccess(result ->
    System.out.println("big fraction = "+ result + "\n"))
  .ignoreElement();
```

Overview of the BigFraction Case Studies

The RxJava Observable case studies show how to reduce & multiply big fractions using the following Observable features:

- e.g., just(), map(), create(), interval(), filter(), doOnNext(), blockingSubscribe(), take(), doOnComplete(), subscribe(), flatMap(), fromIterable(), subscribeOn(), observeOn(), range(), count(), collectList(), & various thread pools

```
return Observable
  .fromCallable(() -> BigFraction
    .reduce(unreducedFraction))

  .subscribeOn(scheduler)

  .flatMap(reducedFraction ->
    Observable
      .fromCallable(() ->
        reducedFraction.multiply
          (sBigReducedFraction))

      .subscribeOn
        (scheduler));
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Observable
Overview of the Project
Reactor AsyncTester Class
Overview of the Project Reactor AsyncTester Class

- Most test methods in the BigFraction case studies run asynchronously via subscribeOn(), so these methods return before their computations complete.

```java
public static Mono<Void> testFractionReductionAsync() {
    BigFraction unreducedFraction = makeBigFraction(...);
    ...
    return Mono
        .fromCallable(() -> BigFraction.reduce(unreducedFraction))
        .subscribeOn(Schedulers.single())
        .map(result -> result.toMixedString())
        .doOnSuccess(result -> System.out.println("big fraction = "+result + "\n")
        .then();
}
```

See Reactive/Mono/ex2/src/main/java/MonoEx.java
Overview of the Project Reactor AsyncTester Class

- It's therefore helpful to define a single location in the main test driver code that waits for all asynchronously executing test methods to complete.

```java
public static void main (String[] argv) {... {
    AsyncTester
        .register(MonoEx::testFractionReductionAsync);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .block();

    ...
}
```

See Reactive/Mono/ex2/src/main/java/ex2.java
Overview of the Project Reactor AsyncTester Class

- The AsyncTester class provides an API to register non-blocking test methods that run \textit{asynchronously}

```java
public static void main (String[] argv) ... {
    AsyncTester
        .register(MonoEx::testFractionReductionAsync);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .block();
    ...
}
```
Overview of the Project Reactor AsyncTester Class

- The AsyncTester class provides an API to register non-blocking test methods that run *asynchronously*

```java
public static void main (String[] argv) ... {
    AsyncTester
        .register(MonoEx::testFractionReductionAsync);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .block();
    ...
}
```

This framework also handles test methods that run and/or block *synchronously*
Overview of the Project Reactor AsyncTester Class

• All registered test methods start running (a)synchronously when AsyncTester.runTests() is called

```java
public static void main (String[] argv) ... {
    AsyncTester
        .register(MonoEx::testFractionReductionAsync);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .block();
    ...
}
```
Overview of the Project Reactor AsyncTester Class

- The test driver then calls block() on the Mono returned from runTests() to wait for all the asynchronous processing to complete.

```java
public static void main (String[] argv) ... {
    AsyncTester
        .register(MonoEx::testFractionReductionAsync);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(MonoEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .block();
    ...
}
```
Overview of the Project Reactor AsyncTester Class

- The AsyncTester class provides a framework that asynchronously runs tests & ensures the test driver does not exit until all async processing completes.

**Class AsyncTester**

```java
java.lang.Object
utils.AsyncTester

public class AsyncTester
extends java.lang.Object

This class asynchronously runs tests that use the Project Reactor framework and ensures that the test driver doesn’t exit until all the asynchronous processing is completed.
```

**Method Summary**

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void</td>
<td>register (java.util.function.Supplier&lt;reactor.core.publisher.Mono&lt;java.lang.Void&gt;&gt; test)</td>
<td>Register the test test so that it can be run asynchronously.</td>
</tr>
<tr>
<td>static</td>
<td>runTests()</td>
<td>Run all the register tests.</td>
</tr>
</tbody>
</table>

See Reactive/Mono/ex2/src/main/java/utils/AsyncTester.java
Overview of the Project Reactor AsyncTester Class

- The AsyncTester class provides a framework that asynchronously runs tests & ensures the test driver does not exit until all async processing completes.

```java
import java.lang.Object;
import reactor.core.publisher.AsyncTester;

public class AsyncTester
extends java.lang.Object

This class asynchronously runs tests that use the Project Reactor framework and ensures that the test driver doesn't exit until all asynchronous processing is completed.

Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| static void       | register
(java.util.function.Supplier<reactor.core.publisherMono<java.lang.Void>> test) | Register the test to run so that it can be run asynchronously. |
| static            | runTests()                                  | Run all the register tests.                      |

We'll explore this AsyncTester class after covering Mono & Flux in more detail.
Overview of the RxJava AsyncTester Class
Overview of the RxJava AsyncTester Class

Most test methods in the BigFraction case studies run asynchronously via `subscribeOn()`, so these methods return before their computations complete.

```java
public static Completable testFractionReductionAsync() {
    BigFraction unreducedFraction = makeBigFraction(...);
    ...
    return Mono
            .fromCallable(() -> BigFraction.reduce(unreducedFraction))
            .subscribeOn(Schedulers.single())
            .map(result -> result.toMixedString())
            .doOnSuccess(result ->
                        System.out.println("big fraction = " + result + "\n"))
            .ignoreElement();
}
```

See Reactive/Single/ex2/src/main/java/SingleEx.java
Overview of the RxJava AsyncTester Class

• It’s therefore helpful to define a single location in the main test driver code that waits for all asynchronously executing test methods to complete.

```java
class AsyncTester {
    public static void main (String[] argv) {
        AsyncTester
            .register(SingleEx::testFractionReductionAsync);
        AsyncTester
            .register(SingleEx::testFractionMultiplicationCallable1);
        AsyncTester
            .register(SingleEx::testFractionMultiplicationCallable2);

        long testCount = AsyncTester
            .runTests()
            .blockingGet();
        ...
    }
}
```

See Reactive/Single/ex2/src/main/java/ex2.java
Overview of the RxJava AsyncTester Class

• The AsyncTester class provides an API to register non-blocking test methods that run *asynchronously*

```java
public static void main (String[] argv) ... {
    AsyncTester
        .register(SingleEx::testFractionReductionAsync);
    AsyncTester
        .register(SingleEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(SingleEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .blockingGet();
    ...
}
```
Overview of the RxJava AsyncTester Class

- The AsyncTester class provides an API to register non-blocking test methods that run *asynchronously*

```java
public static void main (String[] argv) ... {
    AsyncTester
    .register(SingleEx::testFractionReductionAsync);
    AsyncTester
    .register(SingleEx::testFractionMultiplicationCallable1);
    AsyncTester
    .register(SingleEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
    .runTests()
    .blockingGet();
    ...
}
```

This framework also handles test methods that run and/or block *synchronously*
Overview of the RxJava AsyncTester Class

• All registered test methods start running (a)synchronously when AsyncTester.runTests() is called

```java
public static void main (String[] argv) ... {
    AsyncTester
        .register(SingleEx::testFractionReductionAsync);
    AsyncTester
        .register(SingleEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(SingleEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .blockingGet();
...
}
```
Overview of the RxJava AsyncTester Class

- The test driver then calls blockingGet() on the Single returned from runTests() to wait for all the asynchronous processing to complete.

```java
public static void main(String[] argv) ... {
    AsyncTester
        .register(SingleEx::testFractionReductionAsync);
    AsyncTester
        .register(SingleEx::testFractionMultiplicationCallable1);
    AsyncTester
        .register(SingleEx::testFractionMultiplicationCallable2);

    long testCount = AsyncTester
        .runTests()
        .blockingGet();
    ...
}
```
Overview of the RxJava AsyncTester Class

- The AsyncTester class provides a framework that asynchronously runs tests & ensures the test driver does not exit until all async processing completes.

```java
public class AsyncTester
extends java.lang.Object

This class asynchronously runs tests that use the RxJava framework and ensures that the test driver doesn't exit until all the asynchronous processing is completed.

Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncTester()</td>
<td></td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| static void       | register
    (io.reactivex.rxjava3.functions.Supplier<
    io.reactivex.rxjava3.core.Completable>
    test)                  | Register the test test so that it can be run asynchronously. |
| static
    io.reactivex.rxjava3.core.Single<
    java.lang.Long>         | runTests()                                    | Run all the register tests.                      |

See Reactive/Single/ex2/src/main/java/utils/AsyncTester.java
Overview of the RxJava AsyncTester Class

- The AsyncTester class provides a framework that asynchronously runs tests & ensures the test driver does not exit until all async processing completes.

```java
public class AsyncTester
extends java.lang.Object

This class asynchronously runs tests that use the RxJava framework and ensures that the test driver doesn't exit until all the async processing is completed.

Constructor Summary

Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncTester()</td>
<td></td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| static void       | register
(io.reactivex.rxjava3.functions.Supplier<io.reactivex.rxjava3.core.Completable> test) | Register the test test so that it can be run asynchronously. |
| static
io.reactivex.rxjava3.core.Single<java.lang.Long> | runTests() | Run all the register tests. |

We’ll explore this AsyncTester class after covering Single & Observable in detail.
End of Overview of the BigFraction Case Studies