Evaluating Different Java Fork-Join Framework Programming Models

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

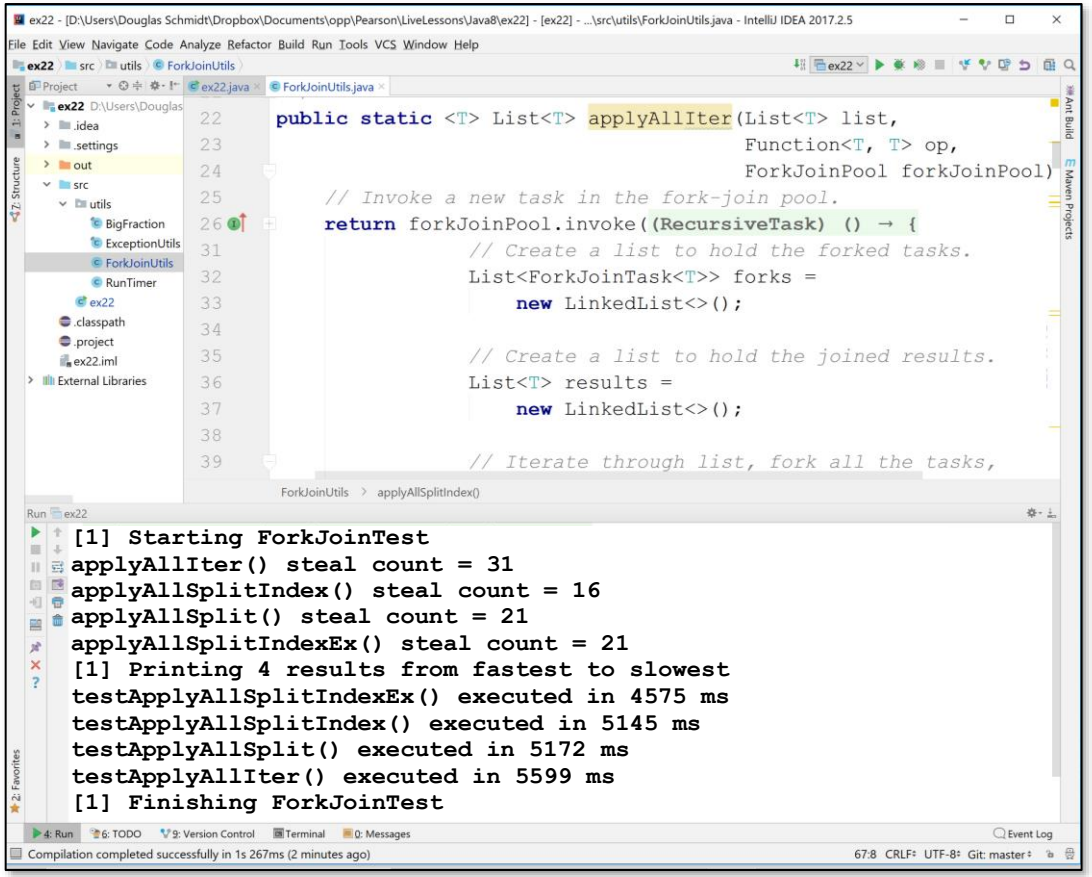
- Evaluate different fork-join framework programming models in practice

```
<T> List<T> applyAllIter(List<T> l, Function<T, T> op, ForkJoinPool fjp) {
  return fjp.invoke(new RecursiveTask<List<T>>() {
    protected List<T> compute() { ... }
   });
}

<T> List<T> applyAllSplit(List<T> l, Function<T, T> op, ForkJoinPool fjp) {
   class SplitterTask extends RecursiveTask<List<T>> { ... }
  return fjp.invoke(new SplitterTask(l));
}

<T> List<T> applyAllSplitIndex(List<T> l,
                               Function<T, T> op, ForkJoinPool fjp) {
  T[] res = (T[]) Array.newInstance(l.get(0).getClass(), l.size());
  class SplitterTask extends RecursiveAction { ... }
  fjp.invoke(new SplitterTask(0, l.size()));
   return Arrays.asList(res);
}
```

Applying the Java Fork-Join Framework

Applying the Java Fork-Join Framework

- Several different Java fork-join programming models are applied on a common dataset

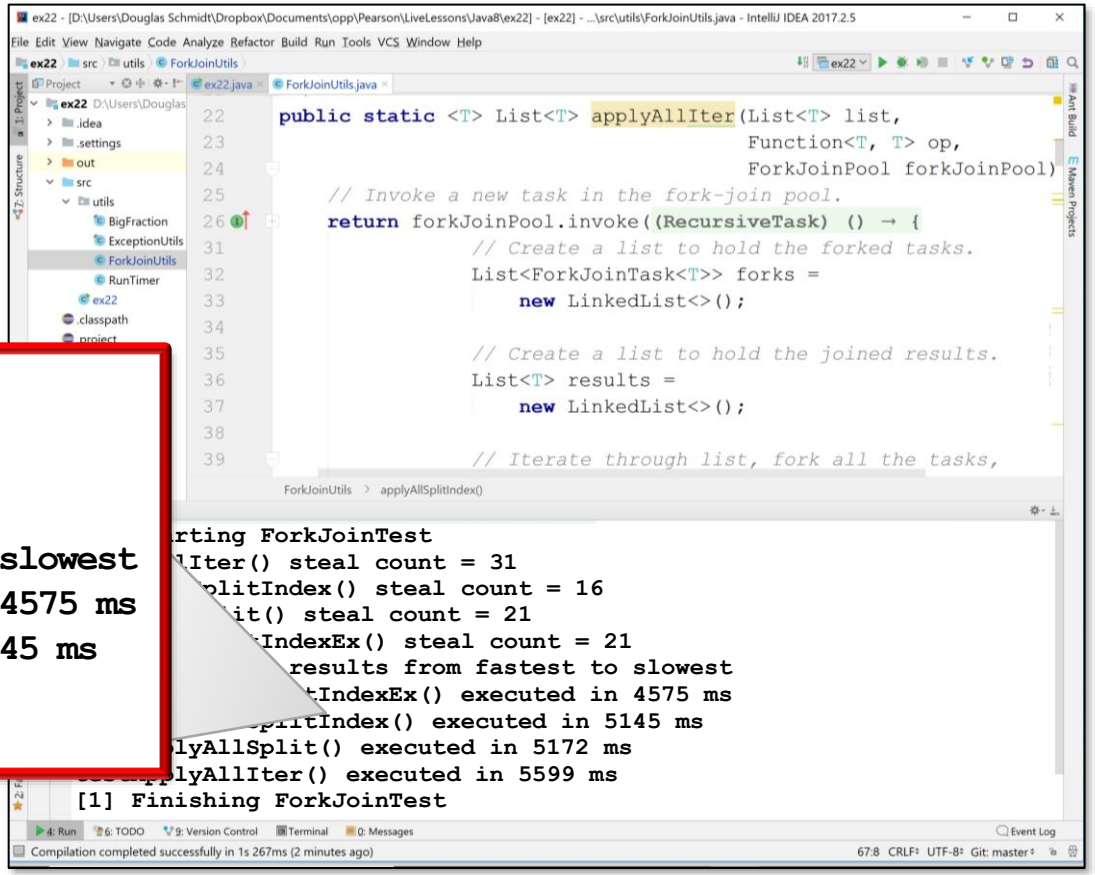


See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex22

Applying the Java Fork-Join Framework

- Several different Java fork-join programming models are applied on a common dataset
  - These models have different performance pros & cons

```
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
```



e.g., some incur more "stealing", copy more data, make more method calls, etc.

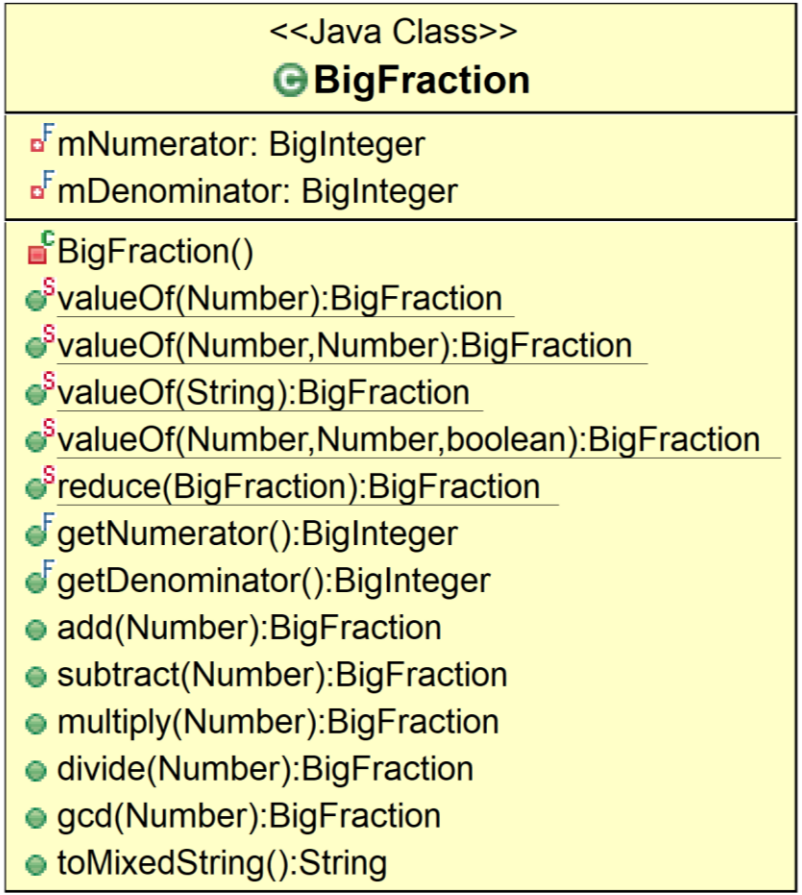
Applying the Java Fork-Join Framework

- Several different Java fork-join programming models are applied on a common dataset
  - These models have different performance pros & cons
  - Java functional programming & sequential streams features are used to simplify the code

```
List<BigFraction> fractionList =
  Stream
    .generate(() ->
       makeBigFraction(new Random(),
                       false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

Function<BigFraction,
         BigFraction> op =
   bigFraction -> BigFraction
     .reduce(bigFraction)
     .multiply(sBigReducedFraction);
```

Applying the Java Fork-Join Framework

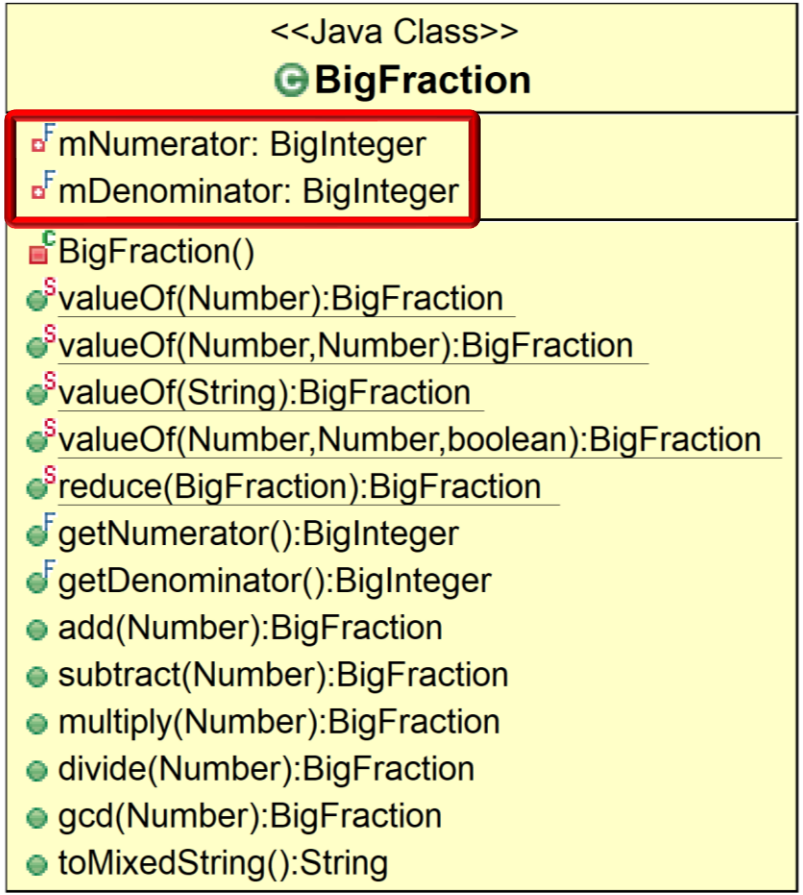
- The fork-join programming models perform the same operations on BigFraction objects



See LiveLessons/blob/master/Java8/ex22/src/utils/BigFraction.java

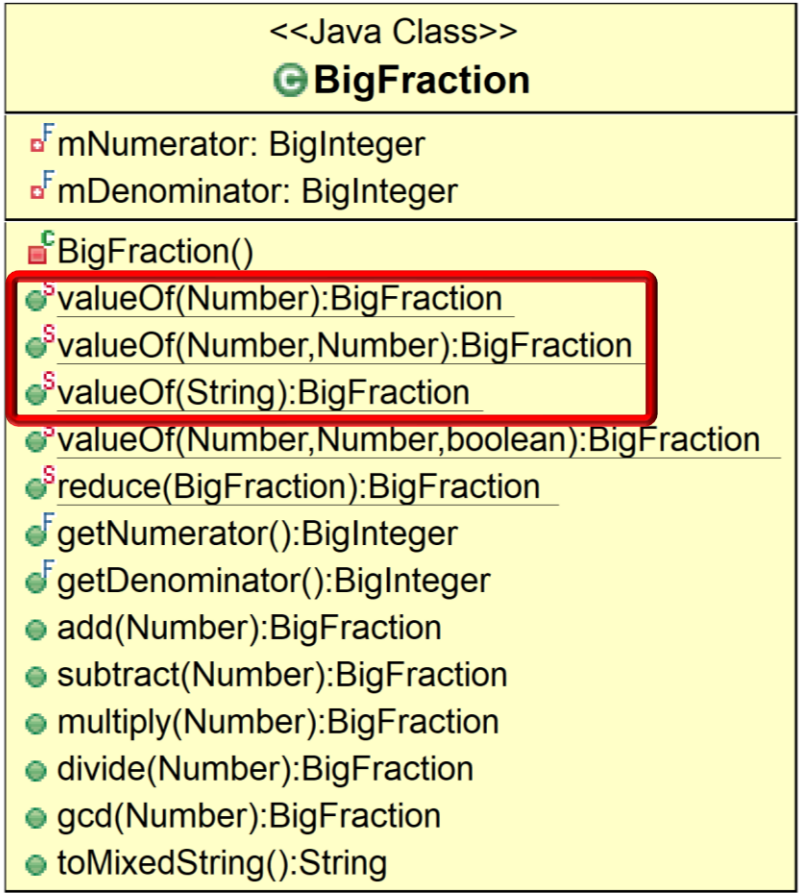
Applying the Java Fork-Join Framework

- The fork-join programming models perform the same operations on BigFraction objects
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator



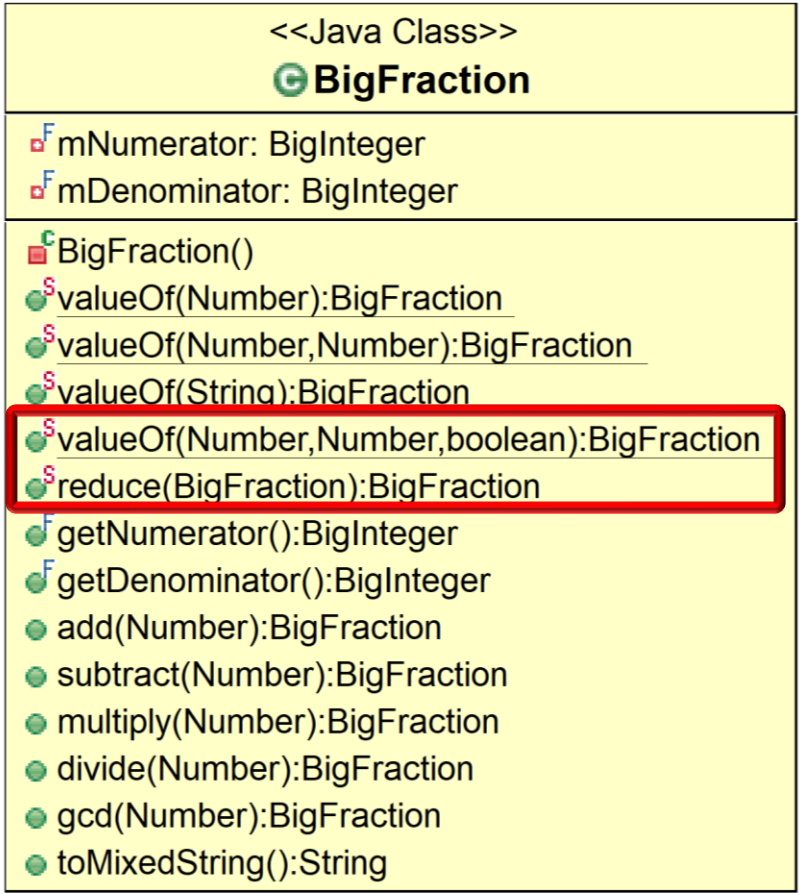
Applying the Java Fork-Join Framework

- The fork-join programming models perform the same operations on BigFraction objects
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating "reduced" fractions, e.g.
    - 44/55 → 4/5
    - 12/24 → 1/2
    - 144/216 → 2/3



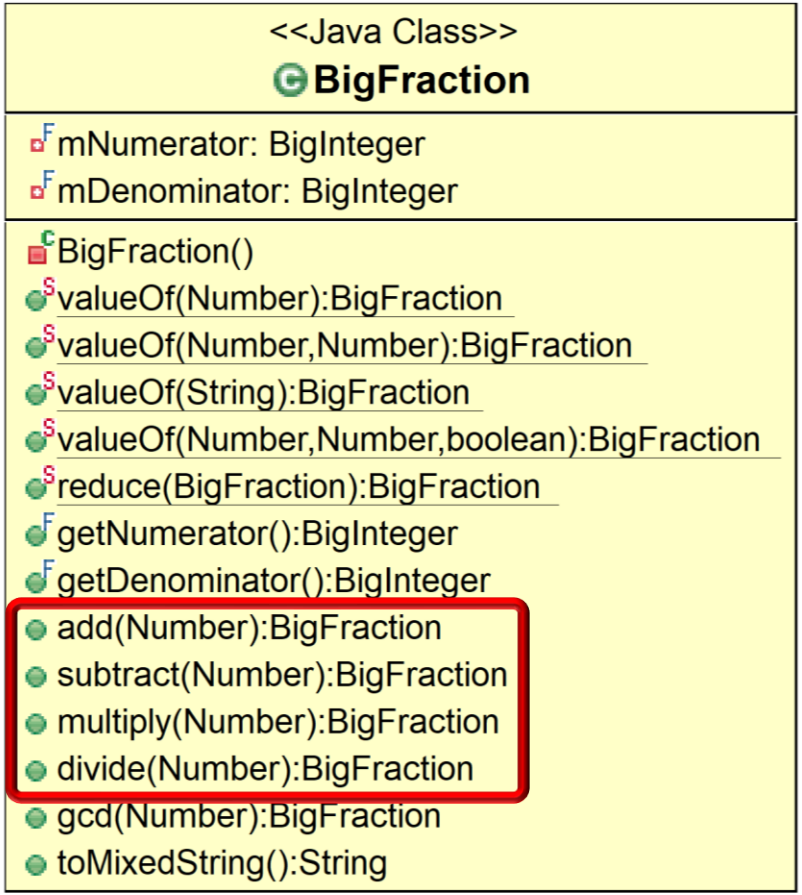
Applying the Java Fork-Join Framework

- The fork-join programming models perform the same operations on BigFraction objects
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (& reducing them)
    - e.g., 12/24 (→ 1/2)



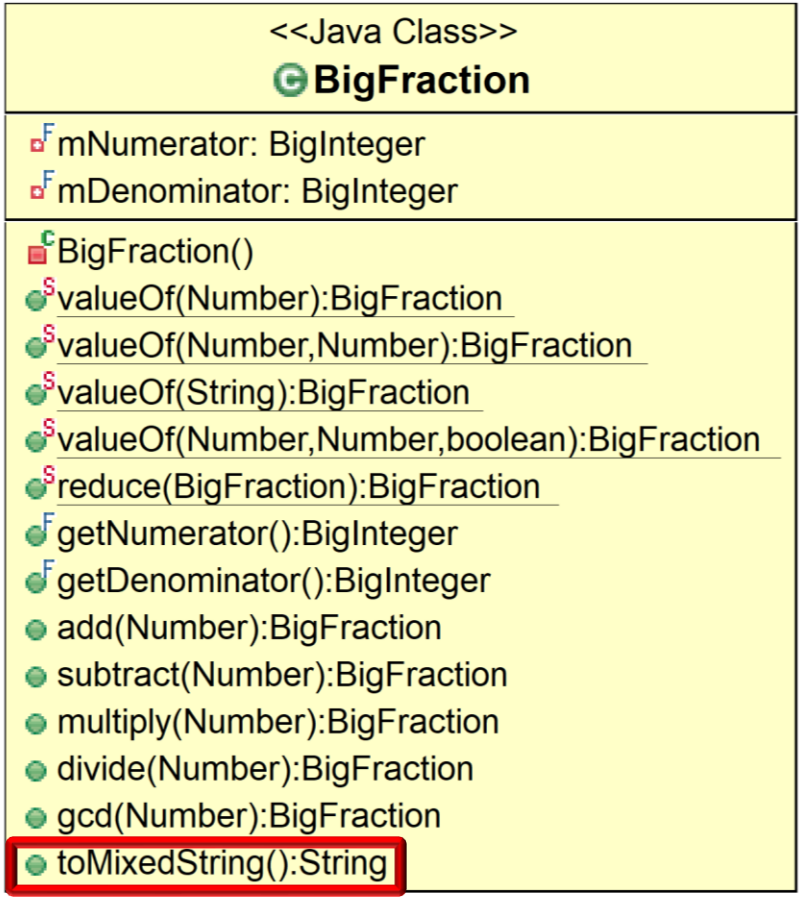
Applying the Java Fork-Join Framework

- The fork-join programming models perform the same operations on BigFraction objects
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (& reducing them)
  - Perform arbitrary-precision fraction arithmetic
    - e.g., 18/4 ×2/3 = 3



Applying the Java Fork-Join Framework

- The fork-join programming models perform the same operations on BigFraction objects
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (& reducing them)
  - Perform arbitrary-precision fraction arithmetic
  - Make mixed fraction from improper fraction
    - e.g., 18/4 → 4 1/2



See www.mathsisfun.com/improper-fractions.html

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

See LiveLessons/blob/master/Java8/ex22/src/ex22.java

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

Use a Java stream to generate random BigFractions up to sMAX_FRACTIONS

This is the primary use of Java streams in this example

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

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

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

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

Factory method that creates a large & random big fraction

See LiveLessons/blob/master/Java8/ex22/src/utils/BigFraction.java

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

```
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 the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

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

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

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

Make a denominator by dividing the numerator by random # between 1 & 10

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

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

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

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

Return a BigFraction w/the numerator & denominator

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

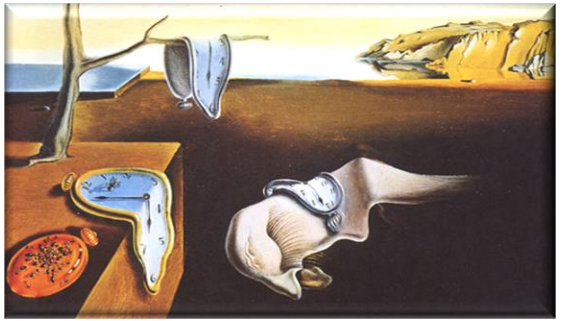
```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

A function that reduces & multiplies a big fraction

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```



This function takes a surprisingly long time to run!

Applying the Java Fork-Join Framework

- Program reduces & multiplies big fractions using the Java fork-join framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

Time various fork-join tests

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() helper methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplitIndex
                      (List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() helper methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplitIndex
                      (List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Each helper method uses a different means of applying the fork-join framework

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() helper methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }
```

Uses "work-stealing" to disperse tasks to worker threads

```
void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplitIndex
                      (List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

See upcoming lesson on "*Java Fork-Join Pool: Implementing applyAllIter()*"

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() helper methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }
```

Uses recursive decomposition to disperse tasks to worker threads

```
void testApplyAllSplitIndex
                      (List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

See upcoming lesson on "*Java Fork-Join Pool: Implementing applyAllSplit()*"

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() helper methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }
```

Uses optimized recursive decomposition to disperse tasks to worker threads

```
void testApplyAllSplitIndex
                      (List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

See upcoming lesson on "*Java Fork-Join Pool: Implementing applyAllSplitIndex()*"

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() helper methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplitIndex
                      (List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Each helper method gets its own fork-join pool sized to the # of processor cores

End of Evaluating Different Java Fork-Join Framework Programming Models