

Applying Key Operators in the Flux Class: Case Study ex2 (Part 1)

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

- Part 1 of case study ex2 shows how to use Flux operators `create()`, `interval()`, `map()`, `filter()`, `take()`, `subscribe()`, `subscribeOn()`, `then()`, `publishOn()`, & `doOnNext()` to create large random `BigInteger` objects & asynchronously check if they are prime in a background thread from the default parallel thread pool

Flux

```
.interval(sSLEEP_DURATION)

.subscribeOn(publisher)

.map(sGenerateRandomBigInteger)

.filter(sOnlyOdd)

.take(sMAX_ITERATIONS)

.subscribe(sink::next,
           err -> sink
                .complete(),
           sink::complete);
```

See github.com/douglasraigschmidt/LiveLessons/tree/master/Reactive/Flux/ex2

Learning Objectives in this Part of the Lesson

- Part 1 of case study ex2 shows how to use Flux operators `create()`, `interval()`, `map()`, `filter()`, `take()`, `subscribe()`, `subscribeOn()`, `then()`, `publishOn()`, & `doOnNext()` to create large random `BigInteger` objects & asynchronously check if they are prime in a background thread from the default parallel thread pool
- The `Mono.fromRunnable()` operator is also shown

Flux

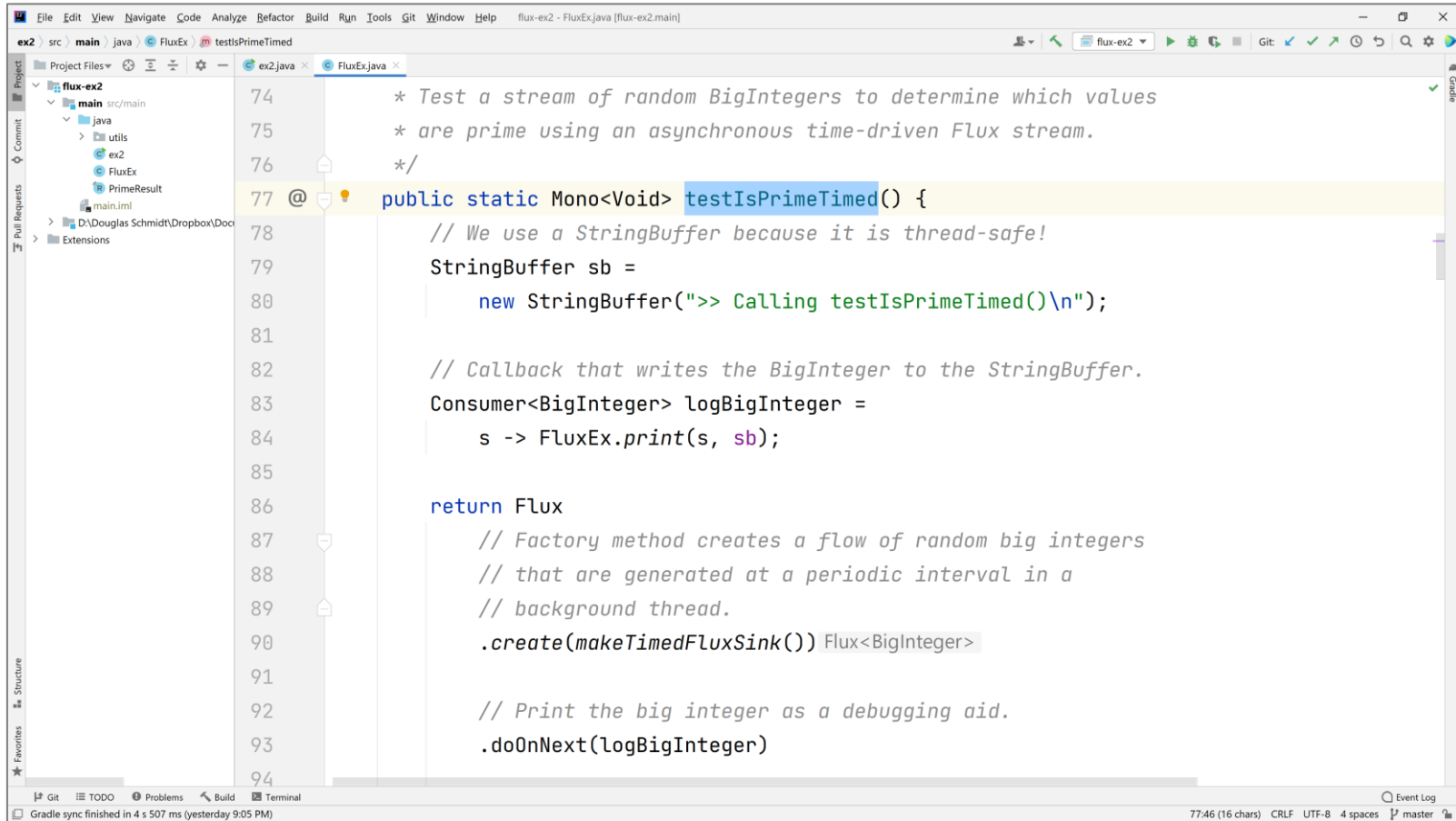
```
.create(makeTimedFluxSink(sb))
...
.map(BigInteger ->
    FluxEx.checkIfPrime
        (BigInteger, sb))

.doOnNext(BigInteger -> FluxEx
    .processResult
        (BigInteger, sb))
...
.then(Mono.fromRunnable(() ->
    BigFractionUtils
        .display
            (sb.toString())));
```

See github.com/douglasraigschmidt/LiveLessons/tree/master/Reactive/Flux/ex2

Applying Key Operators in the Flux Class to ex2

Applying Key Operators in the Flux Class to ex2



```
74      * Test a stream of random BigIntegers to determine which values
75      * are prime using an asynchronous time-driven Flux stream.
76      */
77      @ public static Mono<Void> testIsPrimeTimed() {
78          // We use a StringBuffer because it is thread-safe!
79          StringBuffer sb =
80              new StringBuffer(">> Calling testIsPrimeTimed()\n");
81
82          // Callback that writes the BigInteger to the StringBuffer.
83          Consumer<BigInteger> logBigInteger =
84              s -> FluxEx.print(s, sb);
85
86          return Flux
87              // Factory method creates a flow of random big integers
88              // that are generated at a periodic interval in a
89              // background thread.
90              .create(makeTimedFluxSink()) Flux<BigInteger>
91
92              // Print the big integer as a debugging aid.
93              .doOnNext(logBigInteger)
94      }
```

The screenshot shows an IDE window titled 'flux-ex2 - FluxEx.java [flux-ex2.main]'. The left sidebar shows a project structure with 'flux-ex2' as the root, containing 'main' (src/main), 'utils', 'ex2', 'FluxEx', 'PrimeResult', and 'main.iml'. The main editor displays the code for 'testIsPrimeTimed()' in 'FluxEx.java'. The code is a Java method that returns a 'Mono<Void>' and uses 'Flux' to create a stream of random big integers, testing for primality. Comments explain the use of 'StringBuffer' for thread-safety and the 'makeTimedFluxSink()' method for generating the stream. The IDE interface includes a menu bar, a toolbar, and a status bar at the bottom showing 'Gradle sync finished in 4 s 507 ms (yesterday 9:05 PM)' and '77:46 (16 chars) CRLF UTF-8 4 spaces master'.

See github.com/douglasraigschmidt/LiveLessons/tree/master/Reactive/flux/ex2

End of Applying Key Methods in the Flux Class: Case Study ex2 (Part 1)