Applying Key Methods in the Observable Class (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

- Recognize key methods in the Observable class & how they are applied in the case studies

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html

```
public abstract class Observable<T>
extends Object
implements ObservableSource<T>

The Observable class is the non-backpressured, optionally multi-valued base reactive class that offers factory methods, intermediate operators and the ability to consume synchronous and/or asynchronous reactive dataflows.
```
Learning Objectives in this Part of the Lesson

• Case study ex1 shows how to apply various RxJava operations \textit{synchronously} to reduce & multiply BigFraction objects
  • e.g., fromCallable(), repeat(), just(), map(), mergeWith() & subscribe()

Observable
  .just(BigFraction.valueOf(100,3),
       BigFraction.valueOf(100,4),
       BigFraction.valueOf(100,2),
       BigFraction.valueOf(100,1))
  .map(fraction -> fraction
        .multiply(sBigReducedFraction))
  .subscribe
       (fraction -> sb.append(" = "
                           + fraction.toMixedString()
                           + "\n"),
        error -> sb.append("error"),
        () -> BigFractionUtils
             .display(sb.toString()));

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Observable/ex1
Applying Key Methods in the Observable Class to ex1
Applying Key Methods in the Observable Class to ex1

- testFractionMultiplication1()
  - Performs BigFraction reduction & multiplication using a synchronous Observable stream

```java
Observable
  .just(BigFraction.valueOf(100, 3),
       BigFraction.valueOf(100, 4),
       BigFraction.valueOf(100, 2),
       BigFraction.valueOf(100, 1))
  .map(fraction -> fraction)
     .multiply(sBigReducedFrac)
  .blockingSubscribe
    (fraction -> sb.append(" = "+fraction.toMixedString()+"\n"),
     error -> sb.append("error"),
     () -> BigFractionUtils.display(sb.toString()));
```

See Reactive/Observable/ex1/src/main/java/ObservableEx.java
Applying Key Methods in the Observable Class to ex1

- testFractionMultiplication1()
  - Performs BigFraction reduction & multiplication using a synchronous Observable stream
  - Demonstrates the just(), map(), & blockingSubscribe() methods

```java
Observable
  .just(BigFraction.valueOf(100, 3),
       BigFraction.valueOf(100, 4),
       BigFraction.valueOf(100, 2),
       BigFraction.valueOf(100, 1))
  .map(fraction -> fraction
       .multiply(sBigReducedFrac))
  .blockingSubscribe
   (fraction -> sb.append(" = "
       + fraction.toMixedString()
       + "\n"),
    error -> sb.append("error"),
    () -> BigFractionUtils
        .display(sb.toString()));
```
Applying Key Methods in the Observable Class to ex1

- The `just()` method
  - Create a Observable that emits the given element(s) & then completes

```java
static <T> Observable<T> just(T... data)
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#just
The `just()` method

- Create a Observable that emits the given element(s) & then completes
- Multiple elements can be emitted, unlike the `Single.just()` method

```java
static <T> Observable<T> just (T... data)
```

The just() method

- Create a Observable that emits the given element(s) & then completes

- This factory method adapts non-reactive input sources into the reactive model

```java
Observable
.just(BigFraction.valueOf(100,3),
 BigFraction.valueOf(100,4),
 BigFraction.valueOf(100,2),
 BigFraction.valueOf(100,1))
...
Applying Key Methods in the Observable Class to ex1

- The just() method
  - Create a Observable that emits the given element(s) & then completes
  - This factory method adapts non-reactive input sources into the reactive model
  - Project Reactor’s Flux.just() method works the same

```java
Flux.just(BigFraction.valueOf(100, 3),
          BigFraction.valueOf(100, 4),
          BigFraction.valueOf(100, 2),
          BigFraction.valueOf(100, 1))
...```

See [projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#just](http://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#just)
• The just() method
  • Create a Observable that emits the given element(s) & then completes
  • This factory method adapts non-reactive input sources into the reactive model
  • Project Reactor’s Flux.just() method works the same
• Similar to Stream.of() factory method in Java Streams

```java
@SafeVarargs
static <T> Stream<T> of(T... values)

Returns a sequential ordered stream whose elements are the specified values.

Type Parameters:
T - the type of stream elements
Parameters:
values - the elements of the new stream
Returns:
the new stream
```

```java
Stream.of(BigFraction.valueOf(100,3),
          BigFraction.valueOf(100,4),
          BigFraction.valueOf(100,2),
          BigFraction.valueOf(100,1))
...```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of
Applying Key Methods in the Observable Class to ex1

- The map() method
- Transform the item(s) emitted by this Observable

```java
<V> Observable<V> map
    (Function<? super T,? extends V> mapper)
```

See [reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#map](http://reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#map)
Applying Key Methods in the Observable Class to ex1

- The **map()** method
- Transform the item(s) emitted by this Observable
- Applies a synchronous function to transform each item

```
<V> Observable<V> map
(Function<? super T, ? extends V> mapper)
```

**Interface Function<T,R>**

Type Parameters:
- T - the type of the input to the function
- R - the type of the result of the function

All Known Subinterfaces:
- UnaryOperator<T>

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

Applying Key Methods in the Observable Class to ex1

- The map() method
  - Transform the item(s) emitted by this Observable
    - Applies a synchronous function to transform each item
    - map() can terminate if mapper throws an exception

\[
\text{Observable}\langle V \rangle \ \text{map} \\
(\text{Function}\langle ? \text{super} \ T, ? \text{extends} \ V \rangle \ \text{mapper})
\]
Applying Key Methods in the Observable Class to ex1

- The map() method
- Transform the item(s) emitted by this Observable
  - Applies a synchronous function to transform each item
- The # of output items must match the # of input items
Applying Key Methods in the Observable Class to ex1

- The map() method
- Transform the item(s) emitted by this Observable
  - Applies a synchronous function to transform each item
  - The # of output items must match the # of input items
  - map() can transform the type of elements it processes
Applying Key Methods in the Observable Class to ex1

- The `map()` method
  - Transform the item(s) emitted by this Observable
- Project Reactor’s Observable `.map()` method works the same

See [projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#map](http://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#map)
Applying Key Methods in the Observable Class to ex1

- The map() method
  - Transform the item(s) emitted by this Observable
  - Project Reactor’s Observable.map() method works the same
  - Similar to Stream.map() method in Java Streams

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#map
Applying Key Methods in the Observable Class to ex1

- The blockingSubscribe() method
- Subscribe a Consumer to this Observable

```java
void blockingSubscribe
    (Consumer<? super T> consumer,
     Consumer<? super Throwable> errorConsumer,
     Runnable completeConsumer)
```

Applying Key Methods in the Observable Class to ex1

- The blockingSubscribe() method
  - Subscribe a Consumer to this Observable
  - This method consumes all elements in the sequence, handles errors, & reacts to completion

```java
void blockingSubscribe(
    Consumer<? super T> consumer,
    Consumer<? super Throwable> errorConsumer,
    Runnable completeConsumer)
```

---

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/functions/Consumer.html
Applying Key Methods in the Observable Class to ex1

- The blockingSubscribe() method
- Subscribe a Consumer to this Observable
  - This method consumes all elements in the sequence, handles errors, & reacts to completion
  - This subscription requests “unbounded demand”
    - i.e., Long.MAX_VALUE

```java
void blockingSubscribe
(Consumer<? super T> consumer,
 Consumer<? super Throwable>
 errorConsumer,
 Runnable completeConsumer)
```
The blockingSubscribe() method

- Subscribe a Consumer to this Observable
- This method consumes all elements in the sequence, handles errors, & reacts to completion
- This subscription requests unbounded demand
- Signals emitted to this method are represented by the following regular expression:

```
onNext()*(onComplete()|.onError())?
```
• The `blockingSubscribe()` method
  • Subscribe a Consumer to this Observable
  • Calling this method will block the caller thread until the upstream terminates normally or with an error
The blockingSubscribe() method

- Subscribe a Consumer to this Observable
- Calling this method will block the caller thread until the upstream terminates normally or with an error
- Oddly, there is no equivalent method in Project Reactor.
Applying Key Methods in the Observable Class to ex1

This class shows how to apply RxJava features synchronously to perform basic Observable operations, including fromCallable(), repeat(), just(), map(), mergeWith(), and blockingSubscribe().

```java
public class ObservableEx {

    public static CompletableFuture testFractionMultiplication1() {
        StringBuilder sb =
                new StringBuilder(">> Calling testFractionMultiplication1()\n");

        Observable
                // Use just() to generate a stream of big fractions.
                .just(BigFraction.valueOf(numerator: 100, denominator: 3),
                     BigFraction.valueOf(numerator: 100, denominator: 4),
                     BigFraction.valueOf(numerator: 100, denominator: 2),
                     BigFraction.valueOf(numerator: 100, denominator: 1))

                // Use map() to multiply each element in the stream by a constant.
                .map(fraction -> {
```

See [github.com/douglas craig schmidt/LiveLessons/tree/master/Reactive/Observable/ex1](https://github.com/douglas craig schmidt/LiveLessons/tree/master/Reactive/Observable/ex1)
Applying Key Methods in the Observable Class to ex1

- testFractionMultiplication2() 
- Performs BigFraction reduction & multiplication using two synchronous Observable streams that are merged together

```java
Observable<BigFraction> o1 = Observable.just(BigFraction.valueOf(100, 3), 
    BigFraction.valueOf(100, 4), 
    BigFraction.valueOf(100, 2), 
    BigFraction.valueOf(100, 1));

Observable<BigFraction> o2 = Observable.fromCallable(() -> BigFractionUtils.makeBigFraction(random, true)).repeat(4);

o1.mergeWith(o2)...
```

Applying Key Methods in the Observable Class to ex1

- testFractionMultiplication2()
  - Performs BigFraction reduction & multiplication using two synchronous Observable streams that are merged together
  - Demonstrates the just(), map(), blockingSubscribe(), fromCallable(), repeat(), & mergeWith() methods

```java
Observable<BigFraction> o1 = Observable
  .just(BigFraction.valueOf(100, 3),
       BigFraction.valueOf(100, 4),
       BigFraction.valueOf(100, 2),
       BigFraction.valueOf(100, 1));

Observable<BigFraction> o2 = Observable
  .fromCallable(() ->
               BigFractionUtils.makeBigFraction
                 (random, true))
  .repeat(4);

o1
  .mergeWith(o2)...
```

See Reactive/Observable/ex1/src/main/java/ObservableEx.java
Applying Key Methods in the Observable Class to ex1

- The `fromCallable()` method
- Returns an Observable that, when an observer subscribes to it, does certain things

static <T> Observable<T> fromCallable(Callable<? extends T> callable)

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#fromCallable
Applying Key Methods in the Observable Class to ex1

- The fromCallable() method
- Returns an Observable that, when an observer subscribes to it, does certain things
- Invokes a Callable param

```
static <T> Observable<T> fromCallable(Callable<? extends T> callable)
```

---

**Interface Callable<V>**

Type Parameters:
V - the result type of method call

All Known Subinterfaces:
DocumentationTool.DocumentationTask, JavaCompiler.CompilationTask

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)
The `fromCallable()` method

- Returns an Observable that, when an observer subscribes to it, does certain things
  - Invokes a Callable param
  - The returned Observable emits the value returned from the Callable

```
static <T> Observable<T>
fromCallable(Callable<? extends T> callable)
```
Applying Key Methods in the Observable Class to ex1

• The fromCallable() method
  • Returns an Observable that, when an observer subscribes to it, does certain things
  • This factory method adapts non-reactive input sources into the reactive model

```java
Observable
  .fromCallable(() -> BigFractionUtils.makeBigFraction(random, true))
```
Applying Key Methods in the Observable Class to ex1

• The fromCallable() method
  • Returns an Observable that, when an observer subscribes to it, does certain things
  • This factory method adapts non-reactive input sources into the reactive model
  • This method defers executing the Callable until an observer subscribes to the Observable
  • i.e., it is “lazy”

```javascript
Observable
  .fromCallable(()
    -> BigFractionUtils
      .makeBigFraction(random, true))
```
Applying Key Methods in the Observable Class to ex1

• The fromCallable() method
  • Returns an Observable that, when an observer subscribes to it, does certain things
  • This factory method adapts non-reactive input sources into the reactive model
  • This method defers executing the Callable until an observer subscribes to the Observable
  • i.e., it is “lazy”

Conversely, Observable.just() is “eager”

```java
Observable
  .just(BigFraction.valueOf(100, 3),
       BigFraction.valueOf(100, 4),
       BigFraction.valueOf(100, 2),
       BigFraction.valueOf(100, 1))
...```

See earlier discussion of Observable.just() in this lesson
The `fromCallable()` method

- Returns an Observable that, when an observer subscribes to it, does certain things.
- This factory method adapts non-reactive input sources into the reactive model.
- This method defers executing the Callable until an observer subscribes to the Observable.
- Project Reactor’s method `Mono.fromCallable()` is similar.

See [projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html#fromCallable](http://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html#fromCallable)
Applying Key Methods in the Observable Class to ex1

- The repeat() method $\text{Observable}\langle T\rangle$ \text{repeat}(long times)
- Returns an Observable that repeats the sequence of items emitted by the given Observable at most count # of times

See [reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#repeat](reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#repeat)
The `repeat()` method

- Returns an Observable that repeats the sequence of items emitted by the given Observable at most count # of times

- This method does not operate by default on a particular Scheduler

See [reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#repeat](http://reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#repeat)
The repeat() method

- Returns an Observable that repeats the sequence of items emitted by the given Observable at most count # of times
- This method does not operate by default on a particular Scheduler
- Project Reactor’s Flux.repeat() works the same

See projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#repeat
Applying Key Methods in the Observable Class to ex1

• The mergeWith() method
  Observable<T> mergeWith
  (ObservableSource<? extends T> other)

• Merges the sequence of items of the current Observable with the success value of the other ObservableSource param

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#mergeWith
Applying Key Methods in the Observable Class to ex1

- The `mergeWith()` method
- Merges the sequence of items of the current Observable with the success value of the other ObservableSource param
- Returns the new merged Observable instance

```java
Observable<T> mergeWith
(ObservableSource<? extends T> other)
```
Applying Key Methods in the Observable Class to ex1

• The `mergeWith()` method
  • Merges the sequence of items of the current Observable with the success value of the other ObservableSource param
  • This method combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
The `mergeWith()` method

- Merges the sequence of items of the current Observable with the success value of the other ObservableSource param

- This method combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource

- This merging may interleave the items
Applying Key Methods in the Observable Class to ex1

- The `mergeWith()` method
  - Merges the sequence of items of the current Observable with the success value of the other ObservableSource param
  - This method combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
    - This merging may interleave the items
  - Use `concatWith()` to avoid interleaving

See [reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#concatWith](reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#concatWith)
Applying Key Methods in the Observable Class to ex1

- The mergeWith() method
  - Merges the sequence of items of the current Observable with the success value of the other ObservableSource param
  - This method combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
  - Project Reactor’s method Flux. mergeWith() works the same

See [projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#mergeWith](http://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#mergeWith)
Applying Key Methods in the Observable Class to ex1

- The `mergeWith()` method
  - Merges the sequence of items of the current Observable with the success value of the other ObservableSource param
  - This method combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
  - Project Reactor’s method Flux. `mergeWith()` works the same
- Similar to the `Stream.concat()` method in Java Streams

```java
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#concat](docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#concat)
Applying Key Methods in the Observable Class to ex1

```java
public static Completable testFractionMultiplication2()
{
  StringBuilder sb =
    new StringBuilder("\n\n>> Calling testFractionMultiplication2()\n\n");

  // Random number generator.
  Random random = new Random();

  Observable<BigFraction> o1 = Observable
    .just(new BigFraction(100, 3))
    .flatMap(fff -> Observable.just(fff.multiply(new BigFraction(100, 4))))
    .concatWith(Observable.just(fff.multiply(new BigFraction(100, 2))))
    .concatWith(Observable.just(fff.multiply(new BigFraction(100, 1))));

  // Use fromCallable() to "lazily" generate a stream of random big fractions.
  Observable<BigFraction> o2 = Observable
    .fromCallable(() -> new BigFractionUtils.BigFraction
      .fromCallable(() -> BigFractionUtils.makeBigFraction(random, true))
      .repeatLong(4));

  return sb.append(o1).append(o2).toString().toCompletable();
```