This page will present some elementary RxJava puzzles and walk through some solutions as a way of introducing you to some of the RxJava operators.

Project Euler problem #1

There used to be a site called "Project Euler" that presented a series of mathematical computing conundrums (some fairly easy, others quite baffling) and challenged people to solve them. The first one was a sort of warm-up exercise:

> If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

There are several ways we could go about this with RxJava. We might, for instance, begin by going through all of the natural numbers below 1000 with `range` and then `filter` out those that are not a multiple either of 3 or of 5: ### Java

```
Observable<Integer> threesAndFives = Observable.range(1, 999).filter(e -> e % 3 == 0 || e %
```

Groovy

```
def threesAndFives = Observable.range(1,999).filter({ !((it % 3) && (it % 5)) });
```

Or, we could generate two Observable sequences, one containing the multiples of three and the other containing the multiples of five (by `map`ping each value onto its appropriate multiple), making sure to only generating new multiples while they are less than 1000 (the `takeWhile` operator will help here), and then `merge` these sets: ### Java

```
Observable<Integer> threes = Observable.range(1, 999).map(e -> e * 3).takeWhile(e -> e < 100
Observable<Integer> fives = Observable.range(1, 999).map(e -> e * 5).takeWhile(e -> e < 1000
Observable<Integer> threesAndFives = Observable.merge(threes, fives).distinct();
```

Groovy

```
def threes = Observable.range(1,999).map({it*3}).takeWhile({it<1000});
def fives = Observable.range(1,999).map({it*5}).takeWhile({it<1000});
def threesAndFives = Observable.merge(threes, fives).distinct();
```

Don't forget the `distinct` operator here, otherwise merge will duplicate numbers like 15 that are multiples of both 5 and 3.

Next, we want to sum up the numbers in the resulting sequence. If you have installed the optional `rxjava-math` module, this is elementary: just use an operator like `sumInteger` or `sumLong` on the `threesAndFives` Observable. But what if you don't have this module? How could you use standard RxJava operators to sum up a sequence and emit that sum?

There are a number of operators that reduce a sequence emitted by a source Observable to a single value emitted by the resulting Observable. Most of the ones that are not in the `rxjava-math` module emit boolean evaluations of the sequence; we want something that can emit a number. The `reduce` operator will do the job: ### Java

```
Single<Integer> summer = threesAndFives.reduce(0, (a, b) -> a + b);
```

Groovy

```
def summer = threesAndFives.reduce(0, { a, b -> a+b });
```

Here is how `reduce` gets the job done. It starts with 0 as a seed. Then, with each item that `threesAndFives` emits, it calls the closure `{ a, b -> a+b }`, passing it the current seed value as `a` and the emission as `b`. The closure adds these together and returns that sum, and `reduce` uses this returned value to overwrite its seed. When `threesAndFives` completes, `reduce` emits the final value returned from the closure as its sole emission:

iteration

seed

emission

reduce

1

0

3

3

2

3

5

8

3

8

6

14

...

466

232169

999

233168

Finally, we want to see the result. This means we must subscribe to the Observable we have constructed:

Java

```
summer.subscribe(System.out::print);
```

Groovy

```
summer.subscribe({println(it);});
```

Generate the Fibonacci Sequence

How could you create an Observable that emits the Fibonacci sequence?

The most direct way would be to use the `create` operator to make an Observable "from scratch," and then use a traditional loop within the closure you pass to that operator to generate the sequence. Something like this: ### Java

```
Observable<Integer> fibonacci = Observable.create(emitter -> {
    int f1 = 0, f2 = 1, f = 1;
    while (!emitter.isDisposed()) {
        emitter.onNext(f);
        f = f1 + f2;
        f1 = f2;
        f2 = f;
    }
});
```

Groovy

```
def fibonacci = Observable.create({ emitter ->
  def f1=0, f2=1, f=1;
  while(!emitter.isDisposed()) {
    emitter.onNext(f);
    f  = f1+f2;
    f1 = f2;
    f2 = f;
  };
});
```

But this is a little too much like ordinary linear programming. Is there some way we can instead create this sequence by composing together existing Observable operators?

Here's an option that does this: ### Java

```
Observable<Integer> fibonacci =
        Observable.fromArray(0)
                  .repeat()
                  .scan(new int[]{0, 1}, (a, b) -> new int[]{a[1], a[0] + a[1]})
                  .map(a -> a[1]);
```

Groovy

```
def fibonacci = Observable.from(0).repeat().scan([0,1], { a,b -> [a[1], a[0]+a[1]] }).map({i
```

It's a little janky. Let's walk through it:

The `Observable.from(0).repeat()` creates an Observable that just emits a series of zeroes. This just serves as grist for the mill to keep `scan` operating. The way `scan` usually behaves is that it operates on the emissions from an Observable, one at a time, accumulating the result of operations on each emission in some sort of register, which it emits as its own emissions. The way we're using it here, it ignores the emissions from the source Observable entirely, and simply uses these emissions as an excuse to transform and emit its register. That register gets `[0,1]` as a seed, and with each iteration changes the register from `[a,b]` to `[b,a+b]` and then emits this register.

This has the effect of emitting the following sequence of items: `[0,1]`, `[1,1]`, `[1,2]`, `[2,3]`, `[3,5]`, `[5,8]`...

The second item in this array describes the Fibonacci sequence. We can use `map` to reduce the sequence to just that item.

To print out a portion of this sequence (using either method), you would use code like the following: ### Java

```
fibonacci.take(15).subscribe(System.out::println);
```

Groovy

```
fibonnaci.take(15).subscribe({println(it)})];
```

Is there a less-janky way to do this? The `generate` operator would avoid the silliness of creating an Observable that does nothing but turn the crank of `seed`, but this operator is not yet part of RxJava. Perhaps you can think of a more elegant solution?