



# Testing & RxJava2

Sasa Sekulic

Lead Developer at the United Nations/World Food Programme/ShareTheMeal  
co-author of the Manning book “Grokking Rx”

@sasa\_sekulic

# UN/World Food Programme - ShareTheMeal

We provide food to children in need – over 22.000.000 meals shared!

It costs only 0.60 CHF to feed one child for a day.

*Google Play 2017 Award Winner, SXSW 2016 Innovation Award for New Economy, Google Play Best of 2015*



ShareTheMeal



**World Food  
Programme**

# Simple RxJava2 testing – using `test()`

- For simple, fluent tests

```
Observable.just(1)
    .test()
    .assertValue(1)
```

- But cannot test subjects:

```
val subject = PublishSubject.create<Int>()

subject.onNext(1)
subject.test()
    .assertValueCount(1)
    .assertValue(1)
```

# Simple RxJava2 testing – using TestObserver

- For simple tests

```
val testObserver = Observable.just(1).test()  
testObserver.assertValue(1)
```

- Also for testing Subjects

```
val subject = PublishSubject.create<Int>()  
val testObserver = subject.test()
```

```
subject.onNext(1)  
testObserver.assertValueCount(1)  
testObserver.assertValue(1)
```

```
subject.onNext(2)  
testObserver.assertValueCount(2)  
testObserver.assertValues(1, 2)
```

# Testing methods - values

```
Observable.just(1)
    .test()
    .assertValueCount(1)
    .assertValue(1)
    .assertValue({ it == 1 })
    .assertNoValues()
    .assertValuesOnly(1)
```

```
Observable.just(1, 2)
    .test()
    .assertValueCount(2)
    .assertValues(1, 2)
    .assertValueAt(0, 1)
    .assertValueAt(0) { it == 1 }
    .assertNever(3)
```

# Testing methods – values, cont'd

```
Observable.just(1)
    .test()
    .assertTerminated()
    .assertComplete()
    .assertNoErrors()
    .assertSubscribed()
    .assertNotComplete()
    .assertNotTerminated()
    .assertEmpty()
    .assertError(PaymentException("test"))
    .assertError(PaymentMethodNotSupportedException::class.java)
    .assertError({error -> error.cause == PaymentException("test")})
    .assertErrorMessage("test")
```

# Testing methods - compound

- `assertResult() = assertSubscribed().assertValues().assertNoErrors().assertComplete()`

```
Observable.just(1, 2).test()  
    .assertResult(1, 2)
```

- `assertFailure() = assertSubscribed().assertValues().assertError().assertNotComplete()`

```
Observable.error<Int>(RuntimeException("test")).test()  
    .assertFailure(RuntimeException::class.java)  
    .assertFailure(Predicate { it -> it == RuntimeException() })  
    .assertFailureAndMessage(RuntimeException::class.java, "test")
```

# Testing methods – compound, cont'd

- `assertFailure()` with values:

```
Observable.error<Int>(RuntimeException("test"))  
    .startWith(Observable.just(1, 2))  
    .test()  
    .assertFailure(RuntimeException::class.java, 1, 2)  
    .assertFailure(RuntimeException::class.java)  
    .assertFailure(Predicate { it -> it == RuntimeException() })  
    .assertFailureAndMessage(RuntimeException::class.java, "test")
```



# Testing methods - notes

When using compound operators like zip, if you have multiple errors, only the first one to be propagated is evaluated:

```
val zippedObservable1 = Observable.error<Int>(Throwable("error 1"))
val zippedObservable2 = Observable.error<Int>(Throwable("error 2"))

Observable.zip(zippedObservable1, zippedObservable2, BiFunction({first:
    Int, second: Int -> first + second}))
    .test()
    .assertErrorMessage("error 1")
    .assertErrorMessage("error 2")
```

# Waiting for values

```
Observable.just(1).test()  
    .await()  
    .await(1, TimeUnit.SECONDS)
```

```
Observable.just(1).test()  
    .awaitTerminalEvent()
```

```
Observable.just(1, 2)  
    .delay (2, TimeUnit.SECONDS)  
    .test()  
    .awaitDone(2, TimeUnit.SECONDS)  
    .awaitCount(2, BaseTestConsumer.TestWaitStrategy.SLEEP_1000MS, 2000)  
    .assertTimeout()
```

# Testing with Schedulers

```
Observable.just(1)
    .observeOn(Schedulers.io())
    .test()
    .assertValue(1)
```

## Two approaches:

- **Setting global Schedulers:** easier & quicker, but global
- **Making custom Schedulers “provider”:** local and explicit, but impacts all code

# Testing with Schedulers – global setting

```
RxJavaPlugins.setIOSchedulerHandler { Schedulers.trampoline() }  
Observable.just(1)  
    .observeOn(Schedulers.io())  
    .test()  
    .assertValue(1)
```

## Setting global Schedulers:

```
RxJavaPlugins.setIOSchedulerHandler { Schedulers.trampoline() }  
RxJavaPlugins.setComputationSchedulerHandler { Schedulers.trampoline() }  
RxJavaPlugins.setNewThreadSchedulerHandler { Schedulers.trampoline() }
```

## Setting Android Schedulers:

```
RxAndroidPlugins.reset()  
RxAndroidPlugins.setInitMainThreadSchedulerHandler  
    { Schedulers.trampoline() }
```

# Testing with Schedulers – testing @Rule

## Making @Rule for testing:

```
class TrampolineSchedulerRule : TestRule {  
    override fun apply(base: Statement, d: Description): Statement {  
        return object : Statement() {  
            @Throws(Throwable::class)  
            override fun evaluate() {  
                RxJavaPlugins.setIoSchedulerHandler...  
                try {  
                    base.evaluate()  
                } finally {  
                    RxJavaPlugins.reset()  
                }  
            }  
        }  
    }  
}
```

# Testing with Schedulers – using @Rule

Using @Rule for testing:

```
@Rule
```

```
val schedulerRule = TrampolineSchedulerRule()
```

Problems:

- There's no more concurrency because we're using the same scheduler
- Using same scheduler for all tests in class

```
https://medium.com/@fabioCollini/testing-asynchronous-rxjava-code-using-mockito-8ad831a16877
```

```
https://www.infoq.com/articles/Testing-RxJava
```

# Testing with Schedulers – passing provider

Use your own Schedulers provider – local and more explicit!

```
interface SchedulerProvider {  
    fun ui(): Scheduler  
    fun computation(): Scheduler  
    fun io(): Scheduler  
}  
  
class AppSchedulerProvider : SchedulerProvider {  
    override fun ui() = AndroidSchedulers.mainThread()  
    override fun computation() = Schedulers.computation()  
    override fun io() = Schedulers.io()  
}  
  
class TrampolineSchedulerProvider : SchedulerProvider {  
    override fun ui() = Schedulers.trampoline()  
    override fun computation() = Schedulers.trampoline()  
    override fun io() = Schedulers.trampoline()  
}
```

# Testing with Schedulers – passing provider

## Problems:

- Impacts architecture – you have to pass it to every class manually or via dependency injection

```
class UsesSchedulerProvider (schedulers: SchedulerProvider) {  
    ...  
}
```



# Testing scheduled actions:

```
fun methodWithWorker(scheduler: Scheduler): AtomicInteger {  
    val counter = AtomicInteger()  
    val worker = scheduler.createWorker()  
    worker.schedule({ counter.incrementAndGet() })  
    worker.schedule({ counter.incrementAndGet() })  
    return counter  
}
```

```
val scheduler = TestScheduler()  
val counter = methodWithWorker(scheduler)
```

```
Assert.assertEquals(2, counter.get())
```

```
testScheduler.triggerActions()  
Assert.assertEquals(2, counter.get())
```

# Time travel using TestScheduler

How to test a timeout?

```
val timedoutObservable = Observable.never<Int>()  
    .timeout(5, TimeUnit.SECONDS)
```

```
timedoutObservable.test()  
    .assertError(TimeoutException::class.java)
```

By default, `timeout()` uses computation scheduler:

```
timeout(5, TimeUnit.SECONDS) ==  
timeout(5, TimeUnit.SECONDS, Schedulers.computation())
```

# Time travel using TestScheduler, cont'd

Create TestScheduler (or get a reference):

```
val scheduler = TestScheduler()
```

Use @Rule or SchedulersProvider to pass it to the Observable:

```
val timedoutObservable = Observable.never<Int>()  
    .timeout(5, TimeUnit.SECONDS, scheduler)
```

Time travel using advanceTimeBy():

```
scheduler.advanceTimeBy(5, TimeUnit.SECONDS)  
timedoutObservable.test()  
    .assertError(TimeoutException::class.java)
```

# More TestScheduler magic:

Create TestScheduler with time:

```
val scheduler = TestScheduler(2, TimeUnit.SECONDS)
```

Time travel using `advanceTimeTo()`:

```
Scheduler.advanceTimeTo(5, TimeUnit.SECONDS)
```

Check TestScheduler time:

```
val currentTime = scheduler.now(TimeUnit.SECONDS)
```

# That's (not) all folks!

There's always something new:

- <https://github.com/ReactiveX/RxJava/wiki/What's-different-in-2.0#testing>
- <https://github.com/Froussios/Intro-To-RxJava/blob/master/Part%204%20-%20Concurrency/2.%20Testing%20Rx.md>

**And don't forget: It costs only 0.60 CHF to feed  
one child for a day.**

**[www.sharethemeal.org](http://www.sharethemeal.org)**