Reactive Programming

•••

Events are data and data are events.

History of ReactiveX

- First reactive programming framework introduced by Erik Meijer from Microsoft. Reactive Extensions.
- Ported to java mostly by Ben Christensen from Netflix.
- RxJava was the backbone to many other ports, such as RxAndroid, RxJetty, RxScala, RxKotlin and so on.

“”If we pick a book as an example, the book is the data that can be represented as a model, the reactiveness comes in play as a stream of words being consumed by your eyes.””

Terminology

- Observable pushes things.
- Observer consumes things.
- Subscription is the reference returned after an Observer subscribes to an observable.
- Subscriber is an implementation of an Observer and Subscription permitting it to manual unsubscribe from observables calling unsubscribe().

```
Observable.from(1,2,3).subscribe(::println)

Output:
1
2
3
```

Cold observables vs Hot observables

- Think of ColdObservables as music CD's, where can be replayed for every single listener. They will replay emissions for each observer and call onComplete()

  **Our network calls for instance, they will just be fired once a presenter has subscribed, and refired after each subscription.*

- A hotObservable works more like a radio station, given that it broadcasts the same emissions to all Observers at the same time.

*A good example is a editText that has more than on textChangeListener, for a address lookup and also a form validation for example.

A simple observable and observer

```
val countObservable = Observable.create {
        countObservable.onNext(“Hello”)
        countObservable.onNext(“ “)
        countObservable.onNext(“World!”)
        countObservable.onComplete()

}

countObservable.subscribe(::print)
```

Output:

```
Hello World!
```

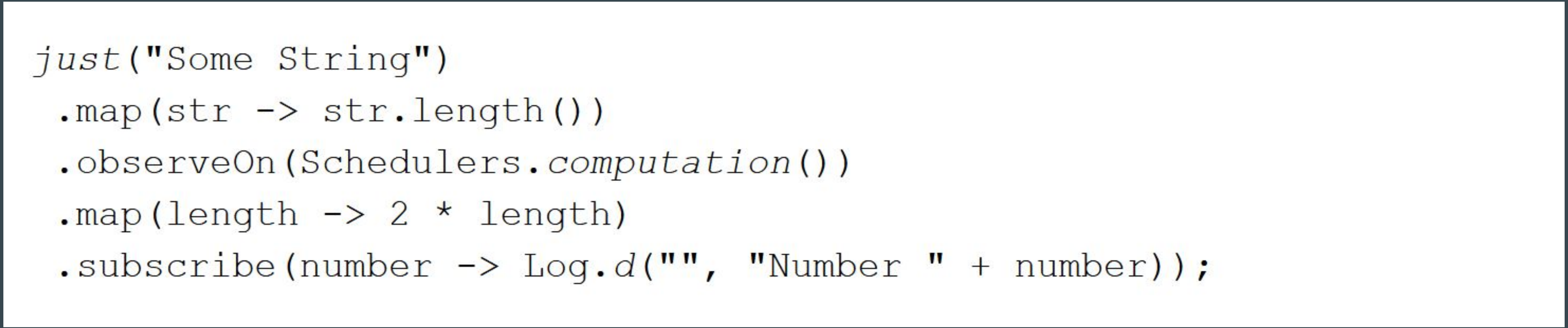
map() vs flatMap()

- map operates on each emission and transforms one event to another whereas flatmap transforms an event to zero or more events.
- For each on next that the source observable emits, a FlatMap will be chained to the stream. *So… what does that mean?*

…would be nice to see an example

observeOn() and subscribeOn()

- subscribeOn acts upon the whole stream and the initial execution. (Position does not matter)
- observeOn only applies to operators further downstream. (calls after)



```
just("Some String")
 .map(str -> str.length())
 .observeOn(Schedulers.computation())
 .map(length -> 2 * length)
 .subscribe(number -> Log.d("", "Number " + number));
```

Useful operators

- Filter
- Debounce
- Reduce
- many others...

Things to fix and improve

Use doOnNext() or doOnSuccess() instead of mapping or flatMapping and returning the same object.

Wrong way 1:

```
val logmeIn = observable.login()  //returns a loginResponse
        .flatMap {
                updateSessionStore(loginResponse)
                Observable.just(loginReponse)
        }
        .subscribe {
                if (it.isSuccess) doStuff....
        }
```

Wrong way 2:

```
val logmeIn = observable.login()  //returns a loginResponse
        .map {
                updateSessionStore(loginResponse)
                loginReponse
        }
        .subscribe {
                if (it.isSuccess) doStuff....
        }
```

Better way:

```
val logmeIn = observable.login()  //returns a loginResponse
        .doOnSuccess {
                updateSessionStore(loginResponse)
        }
        .subscribe {
                if (it.isSuccess) doStuff....
        }
```

We could event check if it was sucessfull and throw an exception

```
val logmeIn = observable.login()  //returns a loginResponse
        .doOnSuccess {
                updateSessionStore(loginResponse)
        }
        .doOnSuccess {
                if (!it.isSucess) throw LoginException()
        }
        .flatMapCompletable()
        .subscribe {}
```

*That will give us only the status of what happened, if the onComplete call is called we know it worked and the session is up-to-date, avoiding then the need of having the logic spread across the different calls.

Things to fix and improve

Use compose() to avoid code duplication and also avoid mistakes.

Compose executes immediately when you create the Observable stream, as if you had written the operators inline. flatMap() executes when its onNext() is called, each time it is called. In other words, flatMap() transforms each item and creates a new observable, whereas compose() transforms the stream.

```
observable.from(1,2,3,4,5)
.map(String::toString)
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe{println(it)}
```

Use

```
.compose(applySchedulers())
```

```
<T> Transformer<T, T> applySchedulers() {
    return observable -> observable.subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread());
}
```

Thanks!