Rx-Java 2

von 0 auf 100 in 45 min

Das Ziel:
“Ist doch eigentlich ganz einfach”

“Dafür hätte ich nicht so lange da bleiben müssen”

Jetzt geht es wirklich los!

Warum Reactive Programming?

Szenario 1

Verbindungsaufbau
Steuerung

GET 1(-3): Port-Guessing (80?, 443?, 8080?)

GET 4: PLC-Infos abrufen

GET 5: Berechtigungs-Infos

GET 6: Daten-Modell

GET 7: Visu-Modell

GET 8: Customizations

GET 9: Initiale Daten

9(+) Requests à 30 ms → kein Problem

Aber: VPN + Mobilfunk → …

Was tun? Klar: Parallelisieren

→ Observer-Pattern / Callbacks (???)

Aber: Calls bauen (teilweise) aufeinander auf

Welcome to Callbacks-Hell

Szenario 2

UI-Anbindung
(Android, Swing, JavaFX)

Aktionen im Hintergrund

Aber: Progress anzeigen

Aber: Abbrechen durch Benutzer

Update der UI nur im UI-Thread

Aber: Nicht zu viele Updates der UI!

Observer-Pattern (???)

Eigentlich richtig - in der Praxis zu kompliziert

Vorlesung “Design Patterns”

Typisches Pattern:
90% Pattern - 10% Probleme / Grenzen

Observer-Pattern
10% Pattern - 90% Probleme / Grenzen

Niemand implementiert das “richtig”

Was dem Observer-Pattern fehlt...

... löst Reactive Programing

Schritt 1

aus Sicht des Observers

Observer

Verarbeitet

Events
Nachrichten
Objekte

UI
Microservice
Monitoring
Backup-System

über die Zeit

Observable

Erzeugt / liefert

Events
Nachrichten
Objekte

HTTP
Datenbank
Berechnungen
(andere) Micro-Services

Grundsätzlicher Ansatz Reactive:

return Observable

Wenn du nicht mehr weiter weißt, füge einen Layer ein...

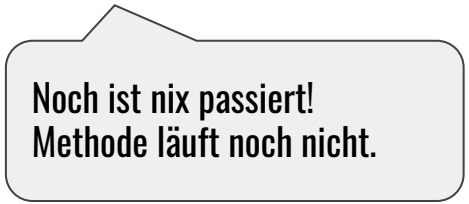
Grundprinzip

```
public Observable<String> fetchBestMovies(){...}
```

Noch ist nix passiert

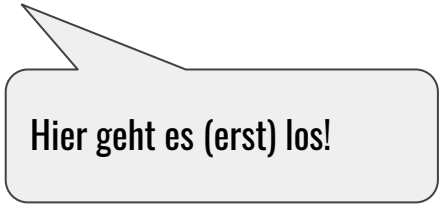
```
public Observable<String> fetchBestMovies(){...}
```

```
Observable<String> myObservable = fetchBestMovies();
```



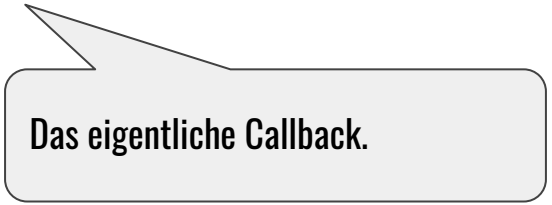
Jetzt geht es los!

```
public Observable<String> fetchBestMovies(){...}


Observable<String> myObservable = fetchBestMovies();

//...

myObservable.subscribe([...]);
```



Callback

```
myObservable.subscribe(new Observer<String>(){

[...]

});
```



Jetzt geht es los

```
myObservable.subscribe(new Observer<String>(){
    public void onSubscribe(Disposable d) { }
    public void onNext(String s) {}
    public void onError(Throwable e) {}
    public void onComplete() {}
});
```

Ganz am Anfang 1x aufgerufen

“Normale” Listener-Methode

```
myObservable.subscribe(new Observer<String>(){
    public void onSubscribe(Disposable d) { }
    public void onNext(String s) {}
    public void onError(Throwable e) {}
    public void onComplete() {}
});
```

Call für jedes Element

Fehlerbehandlung

```
myObservable.subscribe(new Observer<String>(){
    public void onSubscribe(Disposable d) { }
    public void onNext(String s) {}
    public void onError(Throwable e) {}
    public void onComplete() {}
});
```

Falls ein Fehler auftritt

Habe fertich

```
myObservable.subscribe(new Observer<String>(){
    public void onSubscribe(Disposable d) { }
    public void onNext(String s) {}
    public void onError(Throwable e) {}
    public void onComplete() {}
});
```

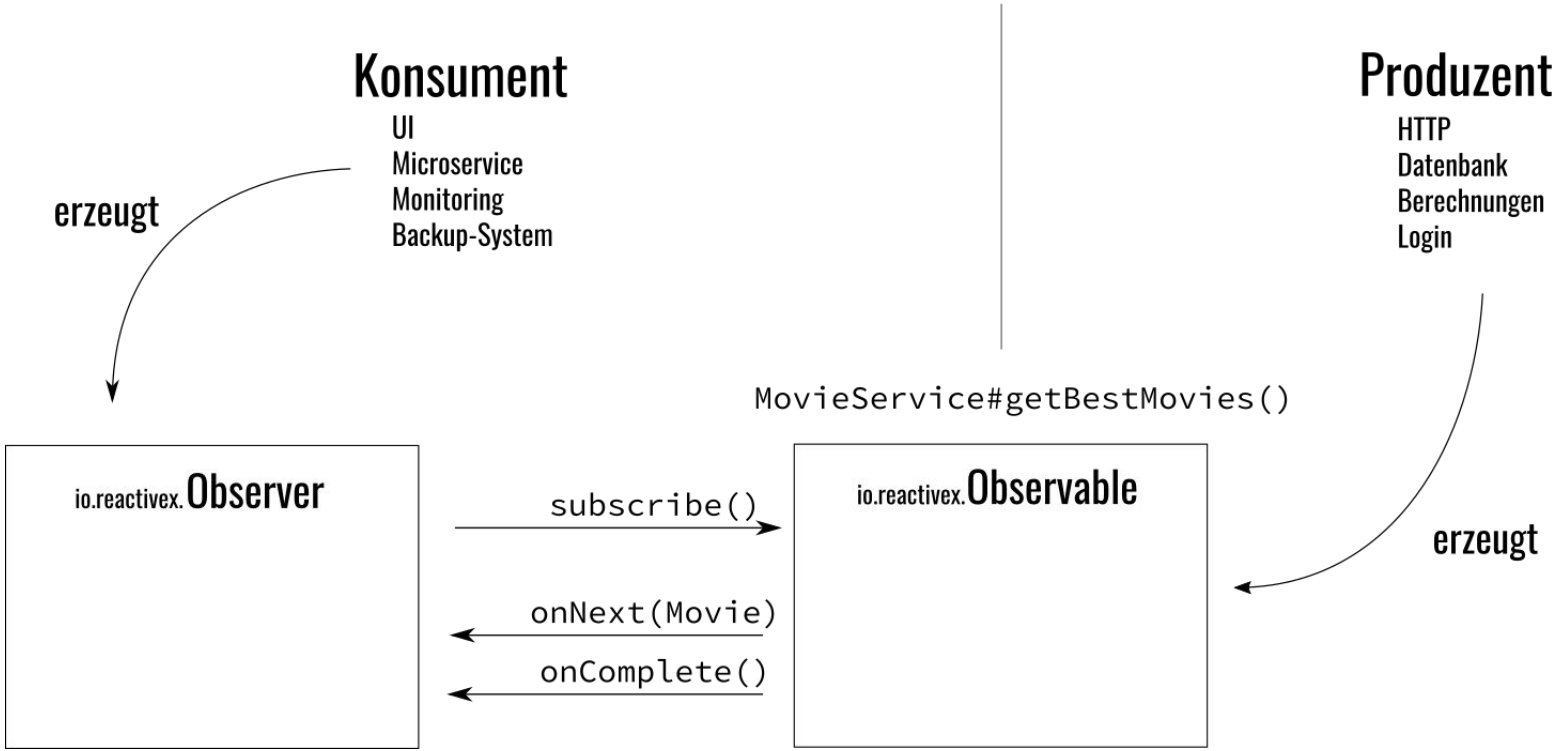
Damit wird gemeldet, dass keine Elemente mehr kommen

Abbruch durch Consumer (z.B. UI-Interaktion)

```
---

myObservable.subscribe(new Observer<String>(){
       @Override public void onSubscribe(Disposable d) {
          //Remember that disposable
       }
[...]
});

disposable.dispose();
```

Observer kann (irgendwann) abbrechen!

Reactive Programming: Overview



Callback auf welchem Thread?

Observer entscheidet!

Callback auf welchem Thread?

```
public Observable<String> fetchBestMovies(){...}


Observable<String> myObservable = fetchBestMovies();

myObservable
.observeOn(SwingScheduler.INSTANCE)
.subscribe();
```

Wir “observieren” auf dem Swing Thread

Callback auf welchem Thread?

```
public Observable<String> getFunnyNames(){...}


Observable<String> myObservable = getFunnyNames();

myObservable
.observeOn(Schedulers.IO)
.subscribe();
```

Wir “observieren” auf einem (großen) Thread-Pool

Callback auf welchem Thread?

```
public Observable<String> getFunnyNames(){...}


Observable<String> myObservable = getFunnyNames();

myObservable
.observeOn(Schedulers.COMPUTE)
.subscribe();
```

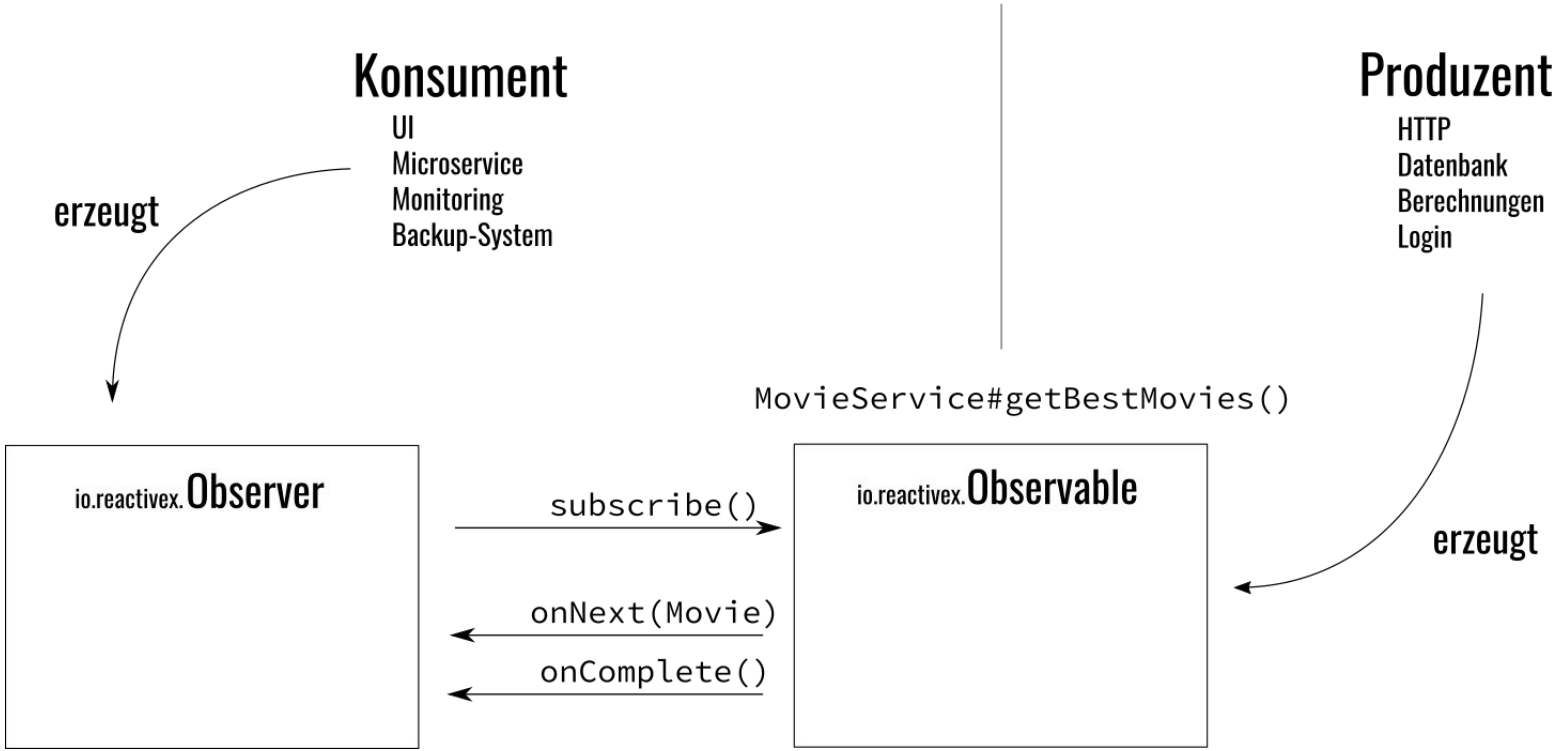
Wir “observieren” auf einem Thread-Pool mit Größe == Anzahl Kerne

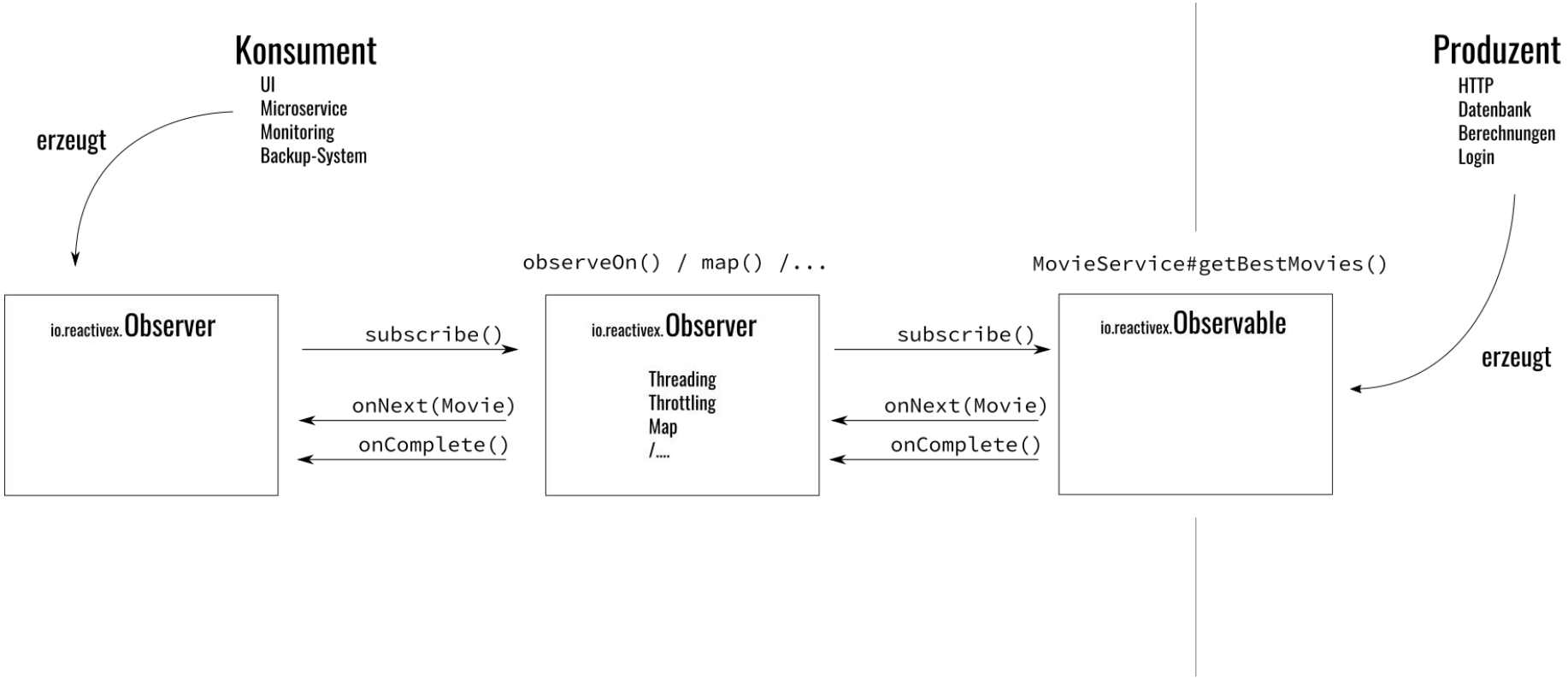
Callback auf welchem Thread?

```
public Observable<String> fetchBestMovies(){...}


Observable<String> myObservable = fetchBestMovies();

myObservable
.observeOn(Schedulers.COMPUTE)
.subscribe();
```

Jeder Call wrappt das ursprüngliche Observable!

Reactive Programming: Overview




Konsument
UI
Microservice
Monitoring
Backup-System
erzeugt
Produzent
HTTP
Datenbank
Berechnungen
Login
observeOn() / map() /...
MovieService#getBestMovies()
io.reactivex.Observer
subscribe()
onNext(Movie)
onComplete()
io.reactivex.Observer
Threading
Throttling
Map
/....
subscribe()
onNext(Movie)
onComplete()
io.reactivex.Observable
erzeugt

Demo 1

Hello World - aus Observer-Sicht

Wiederholung Basics

Wie war das noch?

Observable zurück geben

```
return Observable
```

Anstatt wie klassisch ein Callback zu übergeben

Erst bei subscribe()

Geht die Reise los

Dadurch mehr Kontrolle / Konfigurationsmöglichkeit beim Aufrufer

Alles da was wir brauchen

Übliche Methoden sind vorhanden

```
onNext(T)

onError(Throwable)

onComplete()
```

Threading ist gelöst

(auch eigene Schedulers möglich)

```
observeOn(Scheduler)
```

Observer kann abbrechen

Disposable

Z.B. beim Verlassen einer Seite...

Falls ein anderer Server schneller antwortet...

Schritt 2

aus Sicht des “Observable”

Observer

Verarbeitet

Events
Nachrichten
Objekte

UI
Microservice
Monitoring
Backup-System

über die Zeit

Observable

Erzeugt / liefert

Events
Nachrichten
Objekte

HTTP
Datenbank
Berechnungen
(andere) Micro-Services

Grundsätzlicher Ansatz Reactive:

return Observable

Wenn du nicht mehr weiter weißt, füge einen Layer ein...

Aus Sicht des Observierten

```
---
public Observable<String> fetchBestMovies(){
    [...]
    //Hier passiert noch nichts teures/langes
    return observable;
}
```

Erst bei **subscribe()** geht es los

Vorsicht!
Oder: Was (fast) alle Tutorials falsch machen!

Macht das Sinn??? Deshalb Rective???

```
public Observable<String> fetchBestMovies(){

    return Observable.fromArray(“a”, “b”, “c”);

}
```

Schnell zurückkehren???
Keine Arbeit direkt machen???

Brauche ich dafür eine fette Bibliothek???

Das ist ein (sinnfreies) Beispiel um ein Observable zu bekommen

Besseres Minimal-Beispiel

→ Eigentlich lohnt es sich jetzt schon

Aber das Beste kommt ja noch

Reactive Operatoren

(zu) hohe Frequenz?

Throtteling ist schon da

Bitte etwas langsamer!

```
fetchBestMovies()
.throttleLatest(500L, TimeUnit.MILLISECONDS, true)
.observeOn(SwingScheduler.INSTANCE) //Maybe a ui thread?
.subscribe(new Observer<String>() {
```

Liefert den letzten/jüngsten Wert alle 500 ms

Demo 3

Operatoren, Operatoren, Operatoren

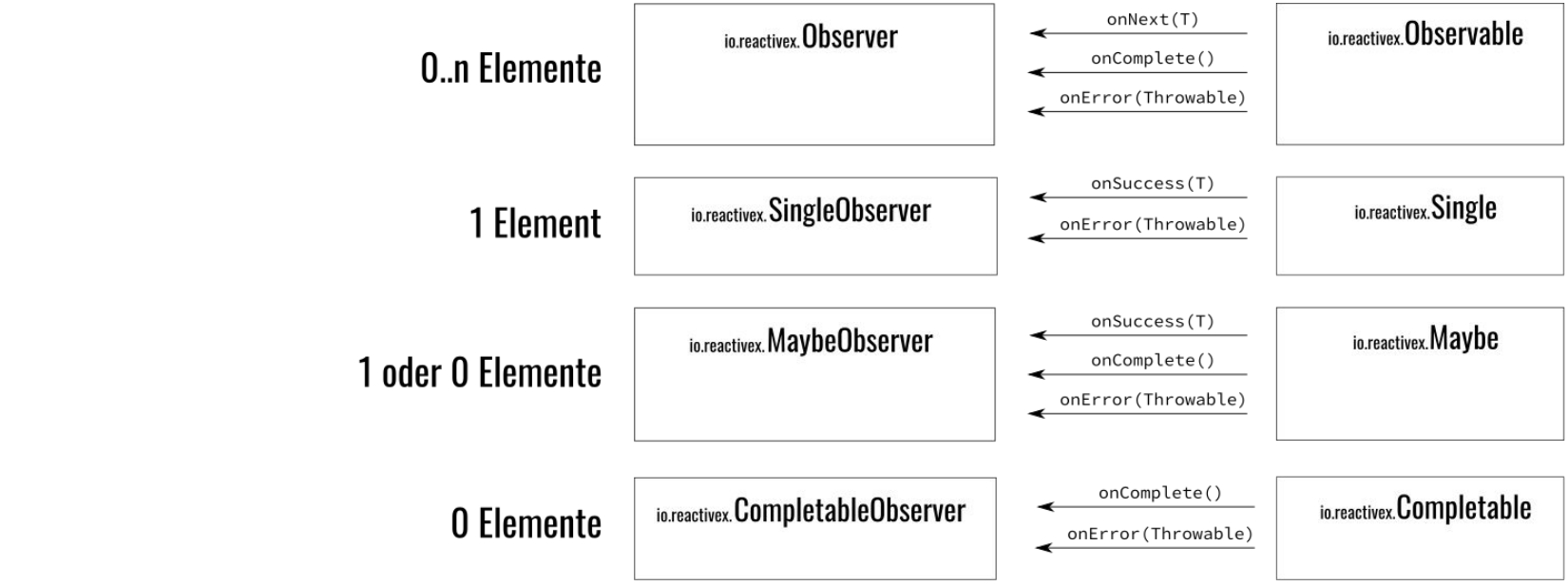
Operatoren sind schwer zu verstehen

deshalb:

https://rxmarbles.com/

Typen von “Observables”

Observable / Flowable
Maybe / Single / Completable



Backpressure (Konsument gibt das Tempo vor)

Szenario 3

Pagination für eine Suche
Endless Loading

Eine Query mit vieeeeeeeeeeeeelen Results

UI entscheidet, wann die nächste Seite kommt

Demo Flowable

Alles perfekt?

Naja...

Performance(!?)

Kein Ersatz für For-Schleifen!

Debugging

Laaaange StackTraces

wie früher bei Hibernate ;-)

Komplexität, Komplexität

“Aus vielen Operatoren folgt große Verantwortung”

Best Practices

Was “unbedingt” zu empfehlen ist

observeOn() wird häufig benötigt

… subscribeOn() sehr selten

doOn...()-Methoden sind sehr hilfreich

Operatoren: Möglichst mit Einzeilern (vergleiche Streaming API)

Abstrakte Basis-Klassen für Observer erstellen

Es gibt für alles bereits einen Operator ;-)

Demos / Tests erstellen

Fragen / Anregungen / Ergänzungen?

Demo Code unter:

https://github.com/jschneider/rxjava2-examples