

Reaktive Programmierung
Vorlesung 8 vom 10.06.14: Reactive Streams II

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

Fahrplan

- ▶ Teil I: Grundlegende Konzepte
- ▶ Teil II: Nebenläufigkeit
 - ▶ Futures and Promises
 - ▶ Reaktive Datenströme I
 - ▶ Reaktive Datenströme II
 - ▶ Funktional-Reaktive Programmierung
 - ▶ Aktoren
 - ▶ Aktoren und Akka
- ▶ Teil III: Fortgeschrittene Konzepte

Rückblick: Observables

- ▶ Observables sind „asynchrone Iterables“
- ▶ Asynchronität wird durch **Inversion of Control** erreicht
- ▶ Es bleiben drei Probleme:
 - ▶ Die Gesetze der Observable können leicht verletzt werden.
 - ▶ Ausnahmen beenden den Strom - **Fehlerbehandlung?**
 - ▶ Ein zu schneller Observable kann den Empfangenden Thread **überfluten**

Datenstromgesetze

- ▶ `onNext*(onError|onComplete)`

- ▶ Kann leicht verletzt werden:

```
Observable[Int] { observer =>
  observer.onNext(42)
  observer.onCompleted()
  observer.onNext(1000)
  Subscription()
}
```

- ▶ Wir können die Gesetze erzwingen: CODE DEMO

Fehlerbehandlung

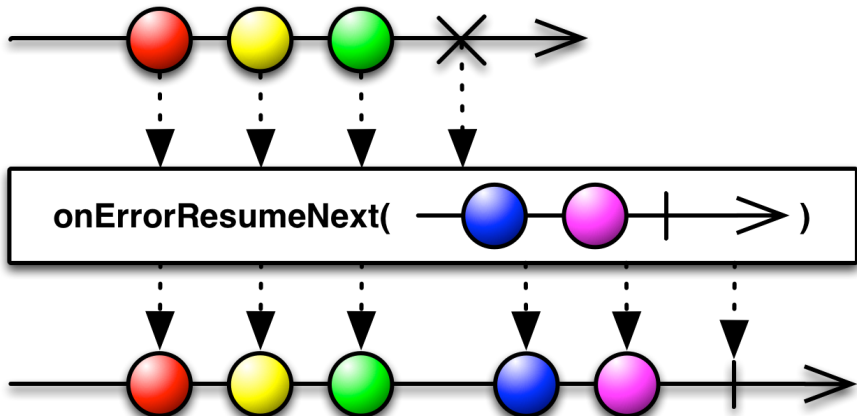
- ▶ Wenn Datenströme Fehler produzieren, können wir diese möglicherweise behandeln.
- ▶ Aber: *Observer.onError* beendet den Strom.

```
observable.subscribe(  
    onNext = println,  
    onError = ???,  
    onCompleted = println("done"))
```

- ▶ *Observer.onError* ist für die Wiederherstellung des Stroms ungeeignet!
- ▶ Idee: Wir brauchen mehr Kombinatoren!

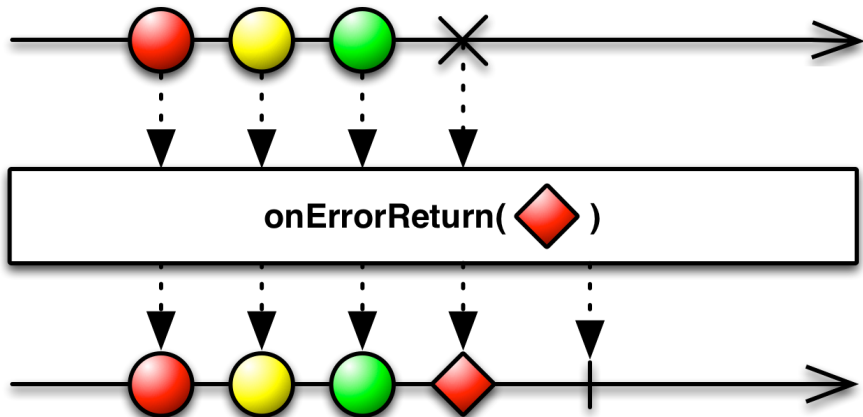
onErrorResumeNext

```
def onErrorResumeNext(f: => Observable[T]): Observable[T]
```



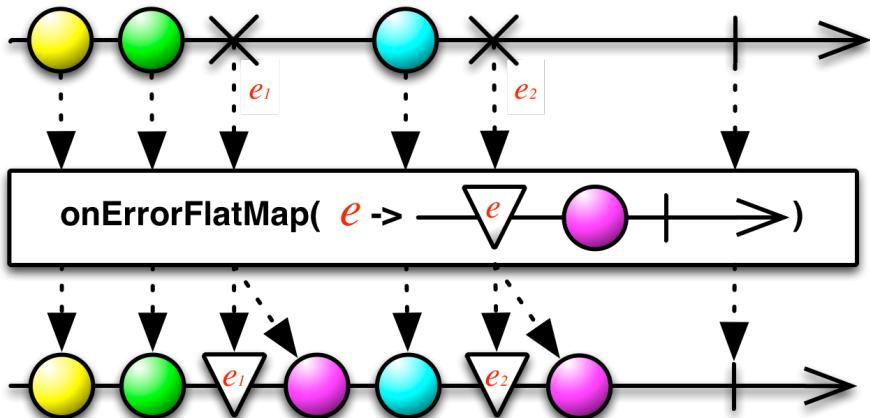
onErrorReturn

```
def onErrorReturn(f: => T): Observable[T]
```



onErrorFlatMap

```
def onErrorFlatMap(f: Throwable => Observable[T]): Observable[T]
```



Schedulers

- ▶ Nebenläufigkeit über Scheduler

```
trait Scheduler {  
  def schedule(work: ⇒ Unit): Subscription  
}
```

```
trait Observable[T] {  
  ...  
  def observeOn(schedule: Scheduler): Observable[T]  
}
```

- ▶ CODE DEMO

Little's Gesetz

- ▶ In einer stabilen Warteschlange gilt:

$$L = \lambda \times W$$

- ▶ Länge der Warteschlange = Ankunftsrate \times Durchschnittliche Wartezeit
- ▶ Ankunftsrate = $\frac{\text{Länge der Warteschlange}}{\text{Durchschnittliche Wartezeit}}$

Little's Gesetz

- ▶ In einer stabilen Warteschlange gilt:

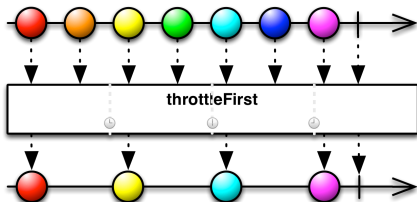
$$L = \lambda \times W$$

- ▶ Länge der Warteschlange = Ankunftsrate \times Durchschnittliche Wartezeit
- ▶ Ankunftsrate = $\frac{\text{Länge der Warteschlange}}{\text{Durchschnittliche Wartezeit}}$
- ▶ Wenn ein Datenstrom über einen längeren Zeitraum mit einer Frequenz $> \lambda$ Daten produziert, haben wir ein Problem!

Throttling / Debouncing

- ▶ Wenn wir L und W kennen, können wir λ bestimmen. Wenn λ überschritten wird, müssen wir etwas unternehmen.
- ▶ Idee: Throttling

```
stream.throttleFirst(lambda)
```

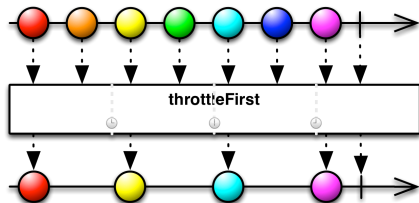
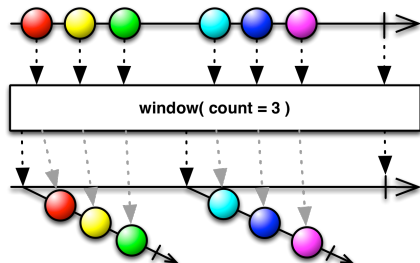


- ▶ Problem: Kurzzeitige Überschreitungen von λ sollen nicht zu Throttling führen.

Throttling / Debouncing

- ▶ Besser: Throttling erst bei längerer Überschreitung der Kapazität:

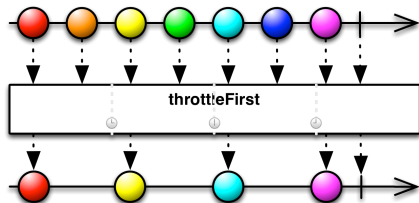
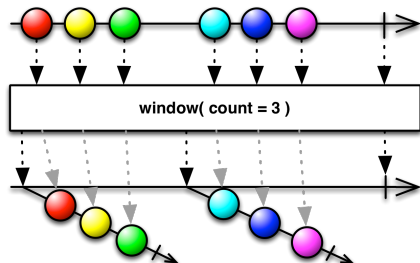
```
stream.window(count = L)  
  .throttleFirst(lambda * L)
```



Throttling / Debouncing

- ▶ Besser: Throttling erst bei längerer Überschreitung der Kapazität:

```
stream.window(count = L)  
    .throttleFirst(lambda * L)
```



- ▶ Was ist wenn wir selbst die Daten Produzieren?

Back Pressure

- ▶ Wenn wir Kontrolle über die Produktion der Daten haben, ist es unsinnig, sie wegzuworfen!
- ▶ Wenn der Konsument keine Daten mehr annehmen kann soll der Produzent aufhören sie zu Produzieren.

- ▶ Erste Idee: Wir können den produzierenden Thread blockieren

```
observable.observeOn(producerThread)
    .subscribe(onNext = someExpensiveComputation)
```

- ▶ Reaktive Datenströme sollen aber gerade verhindern, dass Threads blockiert werden!

Back Pressure

- ▶ Bessere Idee: der Konsument muss mehr Kontrolle bekommen!

```
trait Subscription {  
  def isUnsubscribed: Boolean  
  def unsubscribe(): Unit  
  def requestMore(n: Int): Unit  
}
```

- ▶ Aufwändig zu Implementieren!
- ▶ Siehe <http://www.reactive-streams.org/>

Zusammenfassung

- ▶ Die Konstruktoren in der Rx Bibliothek wenden viel *Magie* an um Gesetze einzuhalten
- ▶ Fehlerbehandlung durch Kombinatoren ist einfach zu implementieren
- ▶ Observables eignen sich nur bedingt um **Back Pressure** zu implementieren, da Kontrollfluss unidirektional konzipiert.
- ▶ Dafür sind Aktoren sehr gut geeignet! (Coming Soon)
- ▶ Nächstes mal: (Rein-)Funktional-Reaktive Programmierung