

# Reaktive Programmierung

## Vorlesung 10 vom 24.05.15: Reactive Streams (Observables)

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017

# Fahrplan

- ▶ Einführung
- ▶ Monaden als Berechnungsmuster
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren I: Grundlagen
- ▶ Aktoren II: Implementation
- ▶ Bidirektionale Programmierung
- ▶ Meta-Programmierung
- ▶ **Reaktive Ströme I**
- ▶ Reaktive Ströme II
- ▶ Functional Reactive Programming
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ Robustheit und Entwurfsmuster
- ▶ Theorie der Nebenläufigkeit, Abschluss

# Klassifikation von Effekten

	Einer	Viele
Synchron	Try [T]	Iterable [T]
Asynchron	Future [T]	Observable [T]

- ▶ Try macht Fehler explizit
- ▶ Future macht Verzögerung explizit
- ▶ Explizite Fehler bei Nebenläufigkeit unverzichtbar
- ▶ Heute: Observables

## Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

## Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

- ▶  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$

# Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

- ▶  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶ Umgedreht:  
 $\text{Unit} \Rightarrow (\text{Unit} \Rightarrow \text{Try}[T])$

# Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

- ▶  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶ Umgedreht:  
 $\text{Unit} \Rightarrow (\text{Unit} \Rightarrow \text{Try}[T])$
- ▶  $() \Rightarrow ( () \Rightarrow \text{Try}[T] )$

# Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

- ▶  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶ Umgedreht:  
 $\text{Unit} \Rightarrow (\text{Unit} \Rightarrow \text{Try}[T])$
- ▶  $() \Rightarrow () \Rightarrow \text{Try}[T]$
- ▶  $\approx \text{Try}[T]$



# Try vs Future

- ▶ Try[T]: Blockieren  $\longrightarrow$  Try[T]
- ▶ Future[T]: Callback  $\longrightarrow$  Try[T] (**Reaktiv**)

## Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

# Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

► () ⇒

## Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

- ▶  $() \Rightarrow () \Rightarrow \text{Try}[\text{Option}[T]]$

# Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

- ▶  $() \Rightarrow () \Rightarrow \text{Try}[\text{Option}[T]]$
- ▶ Umgedreht:  
 $(\text{Try}[\text{Option}[T]] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$

# Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

- ▶  $() \Rightarrow () \Rightarrow \text{Try}[\text{Option}[T]]$
- ▶ Umgedreht:  
 $(\text{Try}[\text{Option}[T]] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶  $( T \Rightarrow \text{Unit}, \text{Throwable} \Rightarrow \text{Unit}, () \Rightarrow \text{Unit} ) \Rightarrow \text{Unit}$

# Observable[T] ist dual zu Iterable [T]

```
trait Iterable[T] {  
  def iterator:  
    Iterator[T]  
}
```

```
trait Iterator[T] {  
  def hasNext: Boolean  
  def next(): T  
}
```

```
trait Observable[T] {  
  def subscribe(Observer[T]  
    observer):  
    Subscription  
}
```

```
trait Observer[T] {  
  def onNext(T value): Unit  
  def onError(Throwable error): Unit  
  def onComplete(): Unit  
}
```

```
trait Subscription {  
  def unsubscribe(): Unit  
}
```

## Warum Observables?

```
class Robot(var pos: Int, var battery: Int) {  
  def goldAmounts = new Iterable[Int] {  
    def iterator = new Iterator[Int] {  
      def hasNext = world.length > pos  
      def next() = if (battery > 0) {  
        Thread.sleep(1000)  
        battery -= 1  
        pos += 1  
        world(pos).goldAmount  
      } else sys.error("low battery")  
    }  
  }  
}  
  
(robotA.goldAmounts zip robotB.goldAmounts)  
  .map(_ + _).takeUntil(_ > 5)
```



# Observable Robots

```
class Robot(var pos: Int, var battery: Int) {  
  def goldAmounts = Observable { obs =>  
    var continue = true  
    while (continue && world.length > pos) {  
      if (battery > 0) {  
        Thread.sleep(1000)  
        pos += 1  
        battery -= 1  
        obs.onNext(world(pos).gold)  
      } else obs.onError(new Exception("low battery"))  
    }  
    obs.onCompleted()  
    Subscription(continue = false)  
  }  
}
```

```
(robotA.goldAmounts zip robotB.goldAmounts)
```

DEMO

# Observable Contract

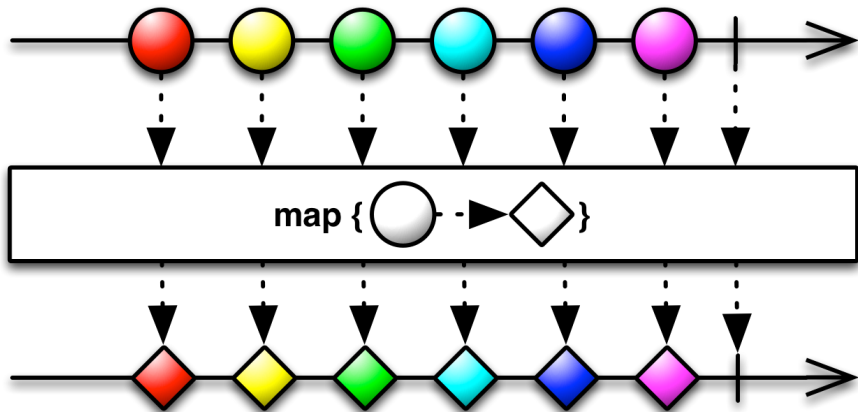
- ▶ die `onNext` Methode eines Observers wird beliebig oft aufgerufen.
- ▶ `onCompleted` oder `onError` werden nur einmal aufgerufen und schließen sich gegenseitig aus.
- ▶ Nachdem `onCompleted` oder `onError` aufgerufen wurde wird `onNext` nicht mehr aufgerufen.

`onNext*(onCompleted|onError)?`

- ▶ Diese Spezifikation wird durch die Konstruktoren erzwungen.

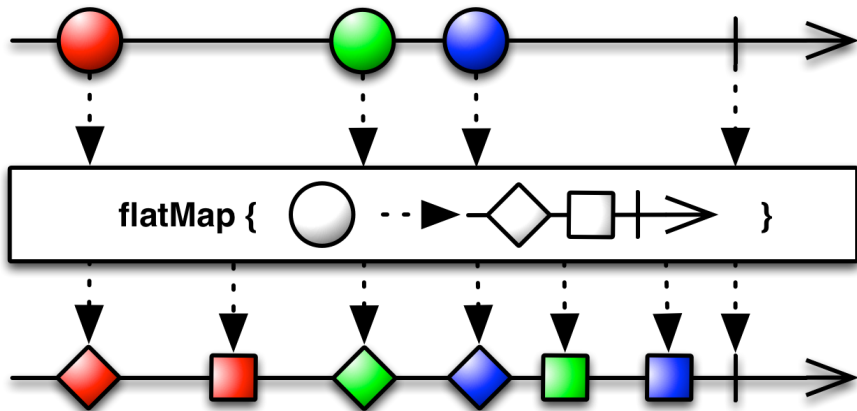
# map

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



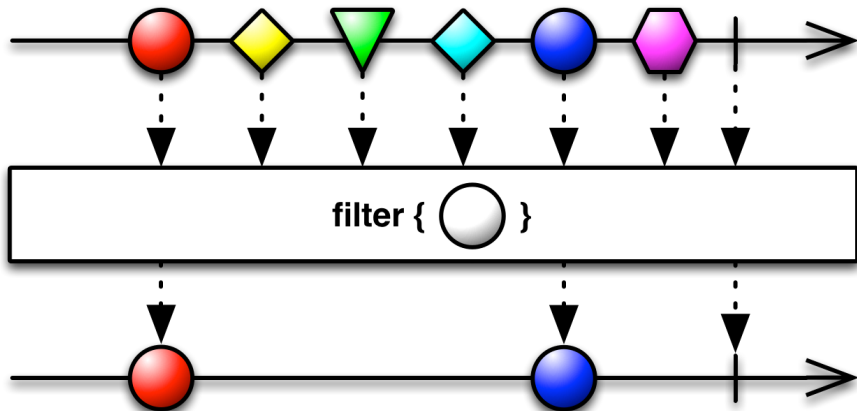
# flatMap

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



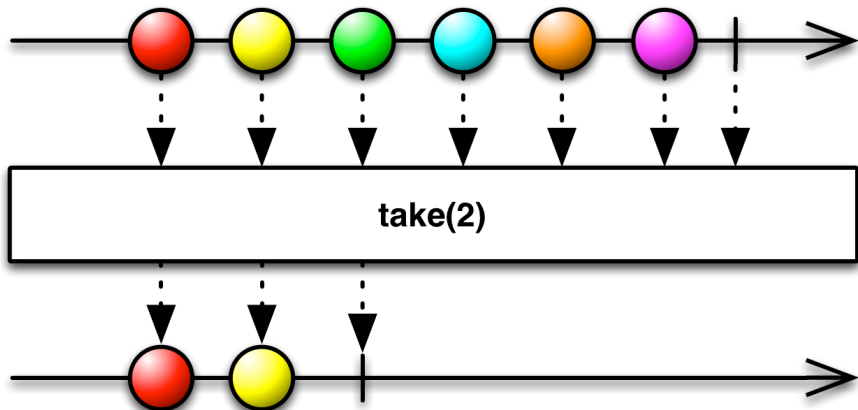
# filter

```
def filter(f: T ⇒ Boolean): Observable[T]
```



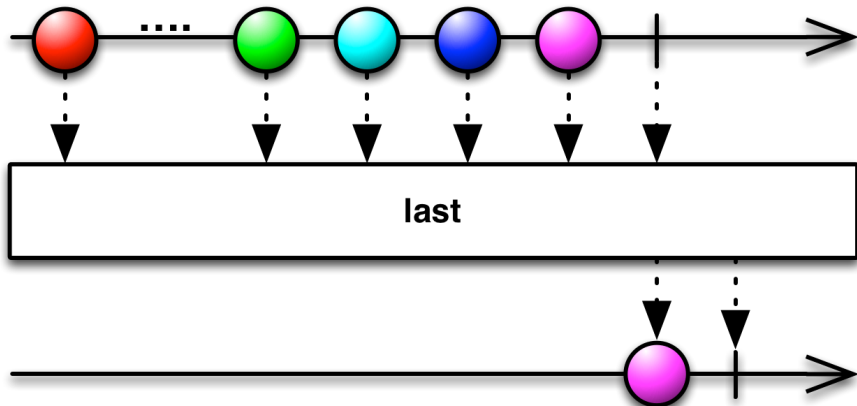
# take

```
def take(count: Int): Observable[T]
```



# last

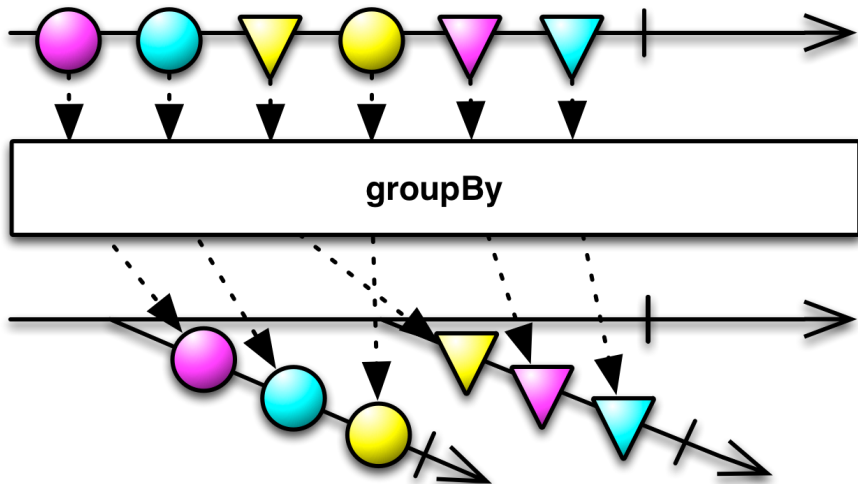
```
def last: Observable[T]
```





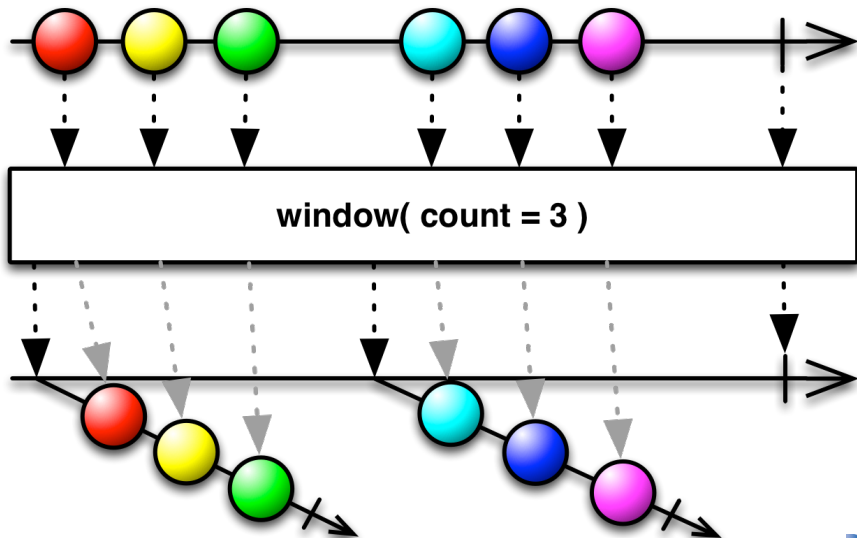
## groupBy

```
def groupBy[U](T => U): Observable[Observable[T]]
```



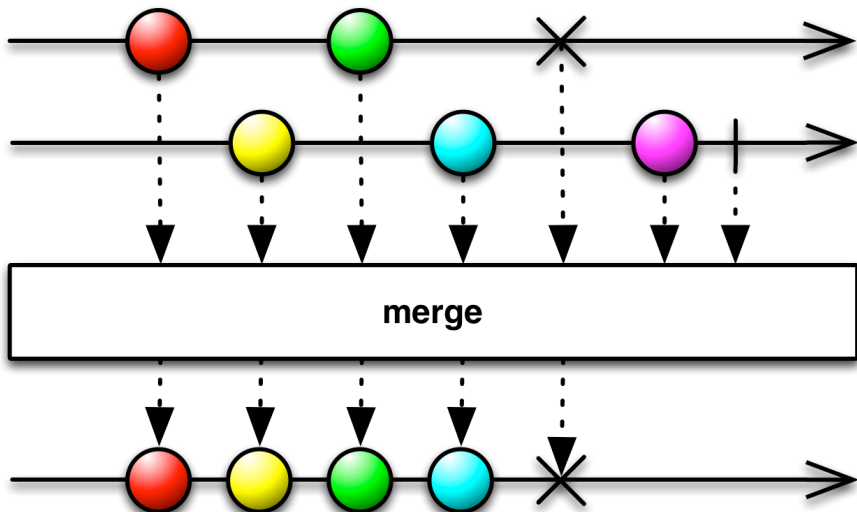
# window

```
def window(count: Int): Observable[Observable[T]]
```



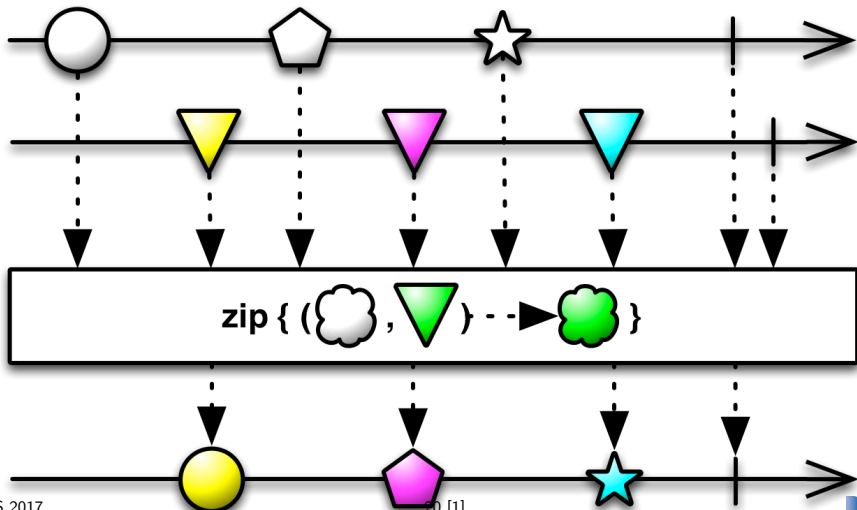
# merge

```
def merge[T](obss: Observable[T]*): Observable[T]
```



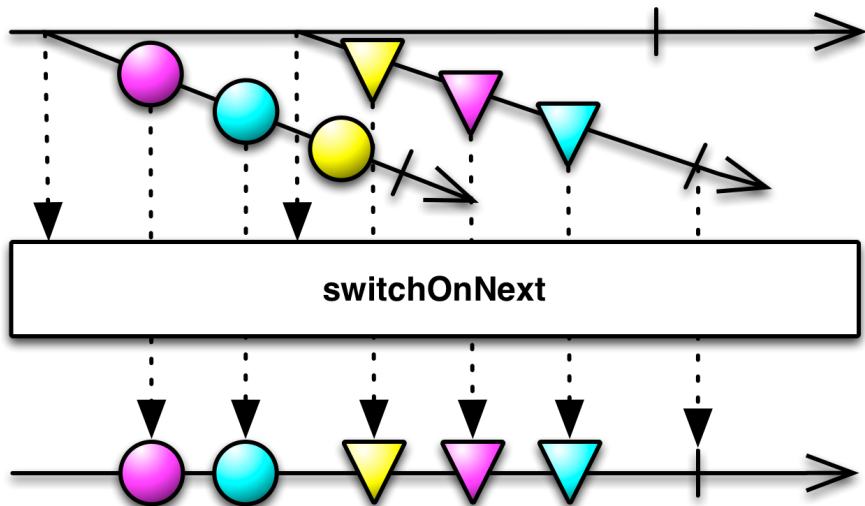
# zip

```
def zip[U,S](obs: Observable[U], f: (T,U) => S):  
  Observable[S]
```



# switch

```
def switch(): Observable[T]
```



# Subscriptions

- ▶ Subscriptions können mehrfach gecancelt werden. Deswegen müssen sie idempotent sein.

```
Subscription(cancel: ⇒ Unit)
```

```
BooleanSubscription(cancel: ⇒ Unit)
```

```
class MultiAssignmentSubscription {  
  def subscription_=(s: Subscription)  
  def subscription: Subscription  
}
```

```
CompositeSubscription(subscriptions: Subscription*)
```

# Schedulers

- ▶ Nebenläufigkeit über Scheduler

```
trait Scheduler {  
  def schedule(work:  $\Rightarrow$  Unit): Subscription  
}  
  
trait Observable[T] {  
  ...  
  def observeOn(schedule: Scheduler): Observable[T]  
}
```

- ▶ Subscription.cancel() muss synchronisiert sein.

# Hot vs. Cold Streams

- ▶ **Hot Observables** schicken allen Observern die gleichen Werte zu den gleichen Zeitpunkten.

z.B. Maus Klicks

- ▶ **Cold Observables** fangen erst an Werte zu produzieren, wenn man ihnen zuhört. Für jeden Observer von vorne.

z.B. `Observable.from(Seq(1,2,3))`



# Observables Bibliotheken

- ▶ Observables sind eine Idee von Eric Meijer
- ▶ Bei Microsoft als .net *Reactive Extension* (Rx) entstanden
- ▶ Viele Implementierungen für verschiedene Plattformen
  - ▶ RxJava, RxScala, RxClosure (Netflix)
  - ▶ RxPY, RxJS, ... (ReactiveX)
- ▶ Vorteil: Elegante Abstraktion, Performant
- ▶ Nachteil: Push-Modell ohne Bedarfsrückkopplung

# Zusammenfassung

- ▶ Futures sind dual zu Try
- ▶ Observables sind dual zu Iterable
- ▶ Observables abstrahieren viele Nebenläufigkeitsprobleme weg:  
Außen **funktional** (Hui) - Innen **imperativ** (Pfui)
- ▶ Nächstes mal: **Back Pressure** und noch mehr reaktive Ströme