

Reaktive Programmierung
Vorlesung 7 vom 31.05.2022
Reaktive Ströme (Observables)

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

Fahrplan

- ▶ Einführung
- ▶ Monaden und Monadentransformer
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren: Grundlagen & Implementierung
- ▶ Bidirektionale Programmierung
- ▶ Meta-Programmierung
- ▶ **Reaktive Ströme I**
- ▶ Reaktive Ströme II
- ▶ Funktional-Reaktive Programmierung
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ CRDTs
- ▶ Robustheit, Entwurfsmuster und Theorie der Nebenläufigkeit, Abschluss
- ▶ Reaktive Programmierung in der Praxis

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
```

- ▶ (Try[T] =>Unit)=>Unit
- ▶ Umgedreht:
Unit =>(Unit =>Try[T])
- ▶ () =>(() =>Try[T])
- ▶ ≈ Try[T]

Try vs Future

- ▶ Try[T]: Blockieren → Try[T]
- ▶ Future[T]: Callback → Try[T] (**Reaktiv**)

Was ist dual zu Iterable?

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

- ▶ () =>(() =>Try[Option[T]])
- ▶ Umgedreht:
(Try[Option[T]] =>Unit)=>Unit
- ▶ (T =>Unit, Throwable =>Unit, ()=>Unit)=>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 then  
        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 do
      if battery > 0 then
        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)
  .map(_ + _).takeUntil(_ > 5)
```

Observables Intern

DEMO

Observable Contract

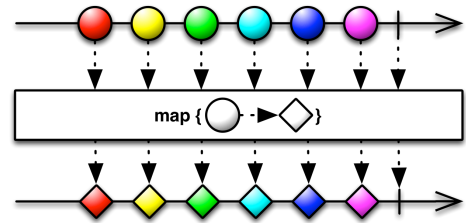
- ▶ die `onNext` Methode eines Observers wird beliebig oft aufgerufen.
- ▶ `onCompleted` oder `onError` werden nur einmal aufgerufen und schließen sich gegenseitig aus.
- ▶ Nachdem `onCompleted` oder `onError` aufgerufen wurde wird `onNext` nicht mehr aufgerufen.

`onNext*(onCompleted|onError)?`

- ▶ Diese Spezifikation wird durch die Konstruktoren erzwingen.

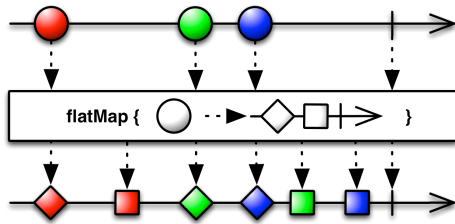
map

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



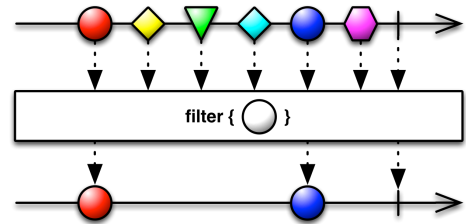
flatMap

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



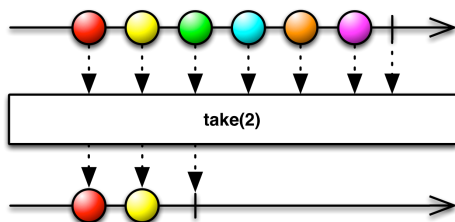
filter

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



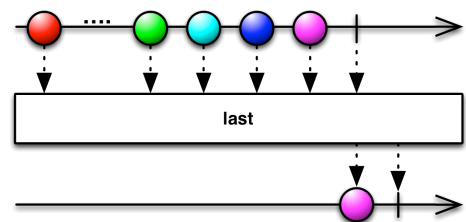
take

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



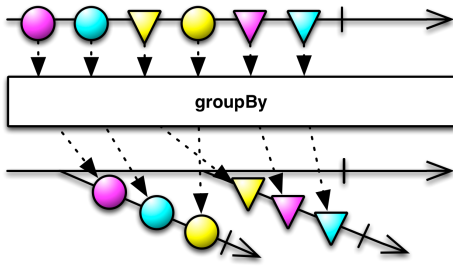
last

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



groupBy

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



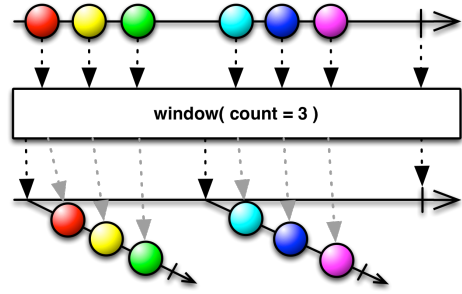
RP SS 2022

17 [26]



window

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



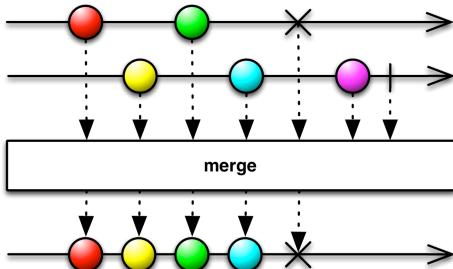
RP SS 2022

18 [26]



merge

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



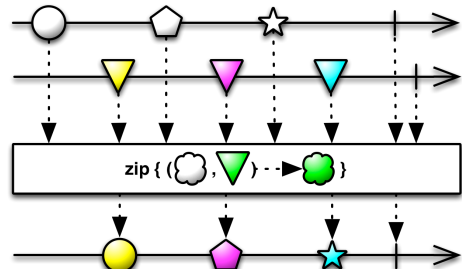
RP SS 2022

19 [26]



zip

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



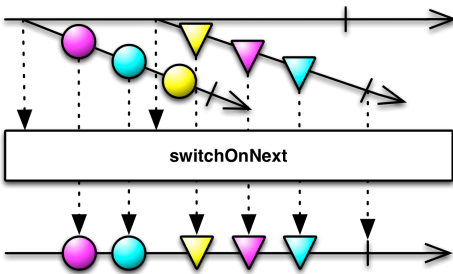
RP SS 2022

20 [26]



switch

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



RP SS 2022

21 [26]



Subscriptions

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

```
trait Subscription:
  def cancel(): Unit
```

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

```
trait MultiAssignmentSubscription extends Subscription:
  def subscription_(s: Subscription)
  def subscription: Subscription
```

RP SS 2022

22 [26]



Schedulers

- Nebenläufigkeit über Scheduler

```
trait Scheduler:
  def schedule(work: => Unit): Subscription
```

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

- Subscription.cancel() muss synchronisiert sein.

RP SS 2022

23 [26]



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))

RP SS 2022

24 [26]



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