# Reaktive Programmierung
# Vorlesung 11 vom 09.06.15: Reactive Streams II

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

# Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Das Aktorenmodell
  - Aktoren und Akka
  - Reaktive Datenströme - Observables
  - Reaktive Datenströme - Back Pressure und Spezifikation
  - Reaktive Datenströme - Akka Streams
- 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

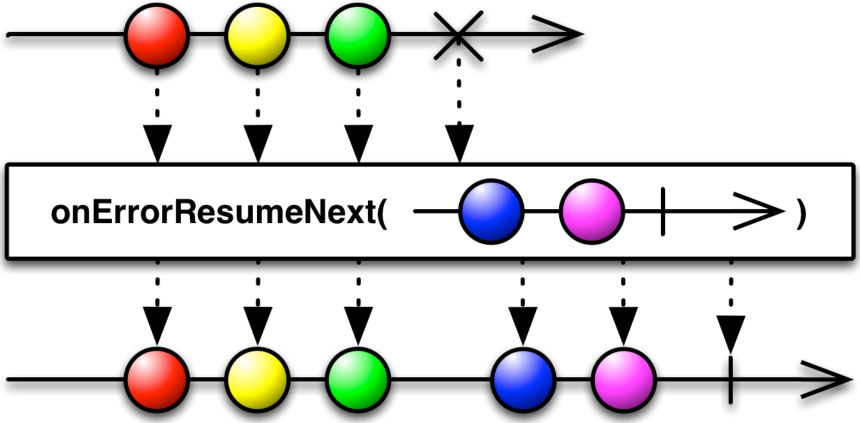
- Wenn Datenströme Fehler produzieren, können wir diese möglicherweise behandeln.
- Aber: *Observer.onError* beendet den Strom.

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

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

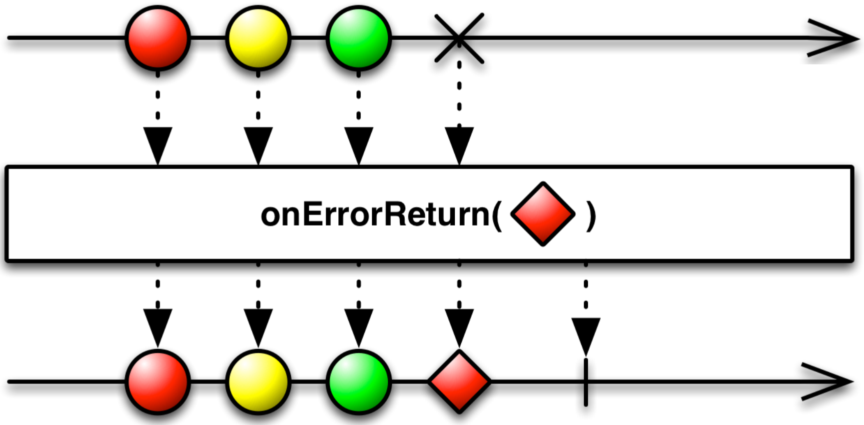
# onErrorResumeNext

```
def onErrorResumeNext(f: ⇒ Observable[T]): Observable[T]
```



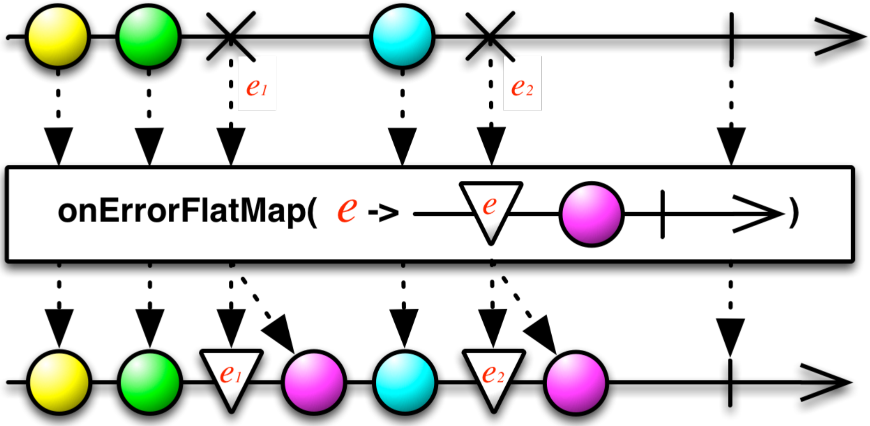
# onErrorReturn

```
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

# Littles Gesetz

- In einer stabilen Warteschlange gilt:

$$L = \lambda \times W$$

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

# Littles Gesetz

- In einer stabilen Warteschlange gilt:

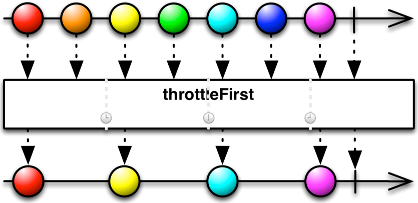
$$L = \lambda \times W$$

- Länge der Warteschlange $=$ Ankunftsrate $\times$ Durschnittliche 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 $\lambda$ bestimmen. Wenn $\lambda$ überschritten wird, müssen wir etwas unternehmen.
- Idee: Throttling

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

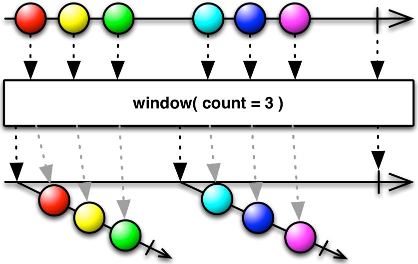


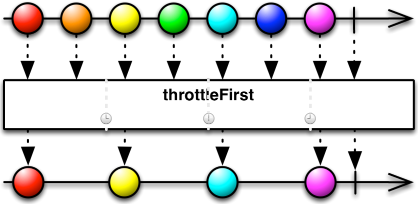

- Problem: Kurzzeitige Überschreigungen von $\lambda$ sollen nicht zu Throttling führen.

# Throttling / Debouncing

- Besser: Throttling erst bei längerer Überschreitung der Kapazität:

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

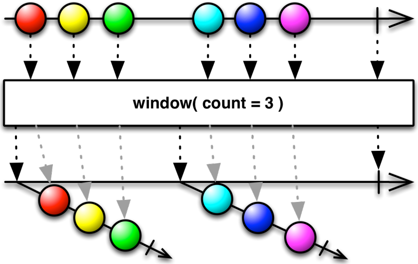


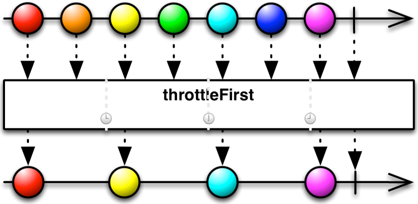



# Throttling / Debouncing

- 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 wegzuwerfen!
- 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 in Observables zu implementieren!
- Siehe `http://www.reactive-streams.org/`

# Reactive Streams Initiative

- Ingenieure von Kaazing, Netflix, Pivotal, RedHat, Twitter und Typesafe haben einen offenen Standard für reaktive Ströme entwickelt
- Minimales Interface (Java + JavaScript)
- Ausführliche Spezifikation
- Umfangreiches Technology Compatibility Kit
- Führt unterschiedlichste Bibliotheken zusammen
  - JavaRx
  - akka streams
  - Slick 3.0 (Datenbank FRM)
  - ...
- Außerdem in Arbeit: Spezifikationen für Netzwerkprotokolle

# Reactive Streams: Interfaces

- `Publisher[O]` – Stellt eine potentiell unendliche Sequenz von Elementen zur Verfügung. Die Produktionsrate richtet sich nach der Nachfrage der `Subscriber`
- `Subscriber[I]` – Konsumiert Elemente eines `Pubilshers`
- `Subscription` – Repräsentiert ein eins zu eins Abonnement eines `Subscribers` an einen `Publisher`
- `Processor[I,O]` – Ein Verarbeitungsschritt. Gleichzeitig `Publisher` und `Subscriber`

# Reactive Streams: 1. Publisher[T]

```
def subscribe(s: Subscriber[T]): Unit
```

1. The total number of onNext signals sent by a Publisher to a Subscriber MUST be less than or equal to the total number of elements requested by that Subscriber's Subscription at all times.
2. A Publisher MAY signal less onNext than requested and terminate the Subscription by calling onComplete or onError.
3. onSubscribe, onNext, onError and onComplete signaled to a Subscriber MUST be signaled sequentially (no concurrent notifications).
4. If a Publisher fails it MUST signal an onError.
5. If a Publisher terminates successfully (finite stream) it MUST signal an onComplete.
6. If a Publisher signals either onError or onComplete on a Subscriber, that Subscriber's Subscription MUST be considered cancelled.

# Reactive Streams: 1. `Publisher[T]`

```
def subscribe(s: Subscriber[T]): Unit
```

7. Once a terminal state has been signaled (`onError`, `onComplete`) it is REQUIRED that no further signals occur.
8. If a `Subscription` is cancelled its `Subscriber` MUST eventually stop being signaled.
9. `Publisher.subscribe` MUST call `onSubscribe` on the provided `Subscriber` prior to any other signals to that `Subscriber` and MUST return normally, except when the provided `Subscriber` is null in which case it MUST throw a java.lang.NullPointerException to the caller, for all other situations the only legal way to signal failure (or reject the `Subscriber`) is by calling `onError` (after calling `onSubscribe`).
10. `Publisher.subscribe` MAY be called as many times as wanted but MUST be with a different `Subscriber` each time.
11. A `Publisher` MAY support multiple Subscribers and decides whether each `Subscription` is unicast or multicast.

# Reactive Streams: 2. Subscriber[T]

```
def onComplete: Unit
def onError(t: Throwable): Unit
def onNext(t: T): Unit
def onSubscribe(s: Subscription): Unit
```

1. A Subscriber MUST signal demand via Subscription.request(long n) to receive onNext signals.
2. If a Subscriber suspects that its processing of signals will negatively impact its Publisher's responsivity, it is RECOMMENDED that it asynchronously dispatches its signals.
3. Subscriber.onComplete() and Subscriber.onError(Throwable t) MUST NOT call any methods on the Subscription or the Publisher.
4. Subscriber.onComplete() and Subscriber.onError(Throwable t) MUST consider the Subscription cancelled after having received the signal.
5. A Subscriber MUST call Subscription.cancel() on the given Subscription after an onSubscribe signal if it already has an active Subscription.

# Reactive Streams: 2. Subscriber[T]

```scala
def onComplete: Unit
def onError(t: Throwable): Unit
def onNext(t: T): Unit
def onSubscribe(s: Subscription): Unit
```

6. A Subscriber MUST call Subscription.cancel() if it is no longer valid to the Publisher without the Publisher having signaled onError or onComplete.
7. A Subscriber MUST ensure that all calls on its Subscription take place from the same thread or provide for respective external synchronization.
8. A Subscriber MUST be prepared to receive one or more onNext signals after having called Subscription.cancel() if there are still requested elements pending. Subscription.cancel() does not guarantee to perform the underlying cleaning operations immediately.
9. A Subscriber MUST be prepared to receive an onComplete signal with or without a preceding Subscription.request(long n) call.
10. A Subscriber MUST be prepared to receive an onError signal with or without a preceding Subscription.request(long n) call.

# Reactive Streams: 2. Subscriber[T]

```
def onComplete: Unit
def onError(t: Throwable): Unit
def onNext(t: T): Unit
def onSubscribe(s: Subscription): Unit
```

11. A `Subscriber` MUST make sure that all calls on its `onXXX` methods happen-before the processing of the respective signals. I.e. the `Subscriber` must take care of properly publishing the signal to its processing logic.
12. `Subscriber.onSubscribe` MUST be called at most once for a given `Subscriber` (based on object equality).
13. Calling onSubscribe, `onNext`, `onError` or `onComplete` MUST return normally except when any provided parameter is null in which case it MUST throw a `java.lang.NullPointerException` to the caller, for all other situations the only legal way for a `Subscriber` to signal failure is by cancelling its `Subscription`. In the case that this rule is violated, any associated `Subscription` to the `Subscriber` MUST be considered as cancelled, and the caller MUST raise this error condition in a fashion that is adequate for the runtime environment.

# Reactive Streams: 3. `Subscription`

```
def cancel(): Unit
def request(n: Long): Unit
```

1. `Subscription.request` and `Subscription.cancel` MUST only be called inside of its Subscriber context. A `Subscription` represents the unique relationship between a Subscriber and a Publisher.
2. The `Subscription` MUST allow the Subscriber to call `Subscription.request` synchronously from within `onNext` or `onSubscribe`.
3. `Subscription.request` MUST place an upper bound on possible synchronous recursion between Publisher and Subscriber.
4. `Subscription.request` SHOULD respect the responsivity of its caller by returning in a timely manner.
5. `Subscription.cancel` MUST respect the responsivity of its caller by returning in a timely manner, MUST be idempotent and MUST be thread-safe.
6. After the `Subscription` is cancelled, additional `Subscription.request(long n)` MUST be NOPs.

# Reactive Streams: 3. Subscription

```
def cancel(): Unit
def request(n: Long): Unit
```

7. After the `Subscription` is cancelled, additional `Subscription.cancel()` MUST be NOPs.
8. While the `Subscription` is not cancelled, `Subscription.request(long n)` MUST register the given number of additional elements to be produced to the respective subscriber.
9. While the `Subscription` is not cancelled, `Subscription.request(long n)` MUST signal onError with a java.lang.IllegalArgumentException if the argument is $\leq 0$. The cause message MUST include a reference to this rule and/or quote the full rule.
10. While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call onNext on this (or other) subscriber(s).
11. While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call onComplete or onError on this (or other) subscriber(s).

# Reactive Streams: 3. `Subscription`

```
def cancel(): Unit
def request(n: Long): Unit
```

12. While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the Publisher to eventually stop signaling its Subscriber. The operation is NOT REQUIRED to affect the `Subscription` immediately.

13. While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the Publisher to eventually drop any references to the corresponding subscriber. Re-subscribing with the same Subscriber object is discouraged, but this specification does not mandate that it is disallowed since that would mean having to store previously cancelled subscriptions indefinitely.

14. While the `Subscription` is not cancelled, calling `Subscription.cancel` MAY cause the `Publisher`, if stateful, to transition into the shut-down state if no other `Subscription` exists at this point.

# Reactive Streams: 3. `Subscription`

```
def cancel(): Unit
def request(n: Long): Unit
```

16. Calling `Subscription.cancel` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.

17. Calling `Subscription.request` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.

18. A `Subscription` MUST support an unbounded number of calls to request and MUST support a demand (sum requested - sum delivered) up to $2^{63} - 1$ (`java.lang.Long.MAX_VALUE`). A demand equal or greater than $2^{63} - 1$ (`java.lang.Long.MAX_VALUE`) MAY be considered by the Publisher as “effectively unbounded”.

# Reactive Streams: 4. Processor[I,O]

```
def onComplete: Unit
def onError(t: Throwable): Unit
def onNext(t: I): Unit
def onSubscribe(s: Subscription): Unit
def subscribe(s: Subscriber[O]): Unit
```

1. A Processor represents a processing stage — which is both a Subscriber and a Publisher and MUST obey the contracts of both.

2. A Processor MAY choose to recover an onError signal. If it chooses to do so, it MUST consider the Subscription cancelled, otherwise it MUST propagate the onError signal to its Subscribers immediately.

# Akka Streams

- Vollständige Implementierung der Reactive Streams Spezifikation
- Basiert auf Datenflussgraphen und Materialisierern
- Datenflussgraphen werden als Aktornetzwerk materialisiert
- Fast final (aktuelle Version 1.0-RC3)

# Akka Streams - Grundkonzepte

Datenstrom (Stream) – Ein Prozess der Daten überträgt und transformiert
Element – Recheneinheit eines Datenstroms
Back-Presure – Konsument signalisiert (asynchron) Nachfrage an Produzenten
Verarbeitungsschritt (Processing Stage) – Bezeichnet alle Bausteine aus denen sich ein Datenfluss oder Datenflussgraph zusammensetzt.
Quelle (Source) – Verarbeitungsschritt mit genau einem Ausgang
Abfulss (Sink) – Verarbeitungsschritt mit genau einem Eingang
Datenfluss (Flow) – Verarbeitungsschritt mit jeweils genau einem Ein- und Ausgang
Ausführbarer Datenfluss (RunnableFlow) – Datenfluss der an eine Quelle und einen Abfluss angeschlossen ist

# Akka Streams - Beispiel

```scala
implicit val system = ActorSystem("example")
implicit val materializer = ActorFlowMaterializer()

val source = Source(1 to 10)
val sink = Sink.fold[Int,Int](0)(_ + _)
val sum: Future[Int] = source runWith sink
```

# Datenflussgraphen

- Operatoren sind Abzweigungen im Graphen
- z.B. Broadcast (1 Eingang, n Ausgänge) und Merge (n Eingänge, 1 Ausgang)
- Scala DSL um Graphen darzustellen

```
val g = FlowGraph.closed() { implicit builder ⇒
  val in = source
  val out = sink
  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))
  val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

  in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
              bcast ~> f4 ~> merge
}
```

# Zusammenfassung

- Die Konstruktoren in der Rx Bibliothek wenden viel Magie an um Gesetze einzuhalten
- Fehlerbehandlung durch Kombinatoren ist einfach zu implementieren
- Observables eigenen sich nur bedingt um Back Pressure zu implementieren, da Kontrollfluss unidirektional konzipiert.
- Die *Reactive Streams*-Spezifikation beschreibt ein minimales Interface für Ströme mit Back Pressure
- Für die Implementierung sind Aktoren sehr gut geeignet $\Rightarrow$ akka streams
- Nächstes mal: Mehr Akka Streams und Integration mit Aktoren