SIEMENS

Programmier-Übungen

zum Tutorial

Reactive-Streams in Java

Marwan Abu-Khalil
2021

Übungen zu Kapitel 1: Reactive-Streams: Pogrammiermodell, Paradigma und Player

SIEMENS

Übungsthema 1: Einfache Streams

RX 1.1: Minimal Rx-Stream
Einen ganz einfachen RxJAva Stream bauen

Einfach

Übungsthema 2: API des Reactive Stream Standards

- **RX 1.2 Subscriber API und Lifecycle**
  Subscriber implementieren

Mittel

- **RX 1.3 Publisher, Subscription, Subscriber bauen**
  Komplette Pipeline programmieren, Regeln implementieren

Schwer

Übungsthema 3: JDK Flow

- **RX 1.4 JDK Flow Simple Stream**
  Stream mit JDK Flow bauen

Einfach

Übungen zu Kapitel 2:
RxJava APIs, Operatoren, Konzepte

SIEMENS

Übungsthema 1: Pipeline Operatoren

- **RX 2.1: Simple Stream**
  Stream mit einer oder mehreren Stufen bauen — Einfach

Übungsthema 2: Parallel-Basics

- **RX 2.2 Multiple Subscribers**
  Verzweigen des Streams durch Anmelden mehrerer Subscriber — Mittel
- **RX 2.3 flatMap() - groupBy(): Verzweigen und Zusammenführen**
  Basis für die Parallelisierung — Schwer
- **RX 3.1: API für Parallelität und Nebenläufigkeit**
  Einsatz von subscribeOn() und observeOn() — Mittel

Übungsthema 4: Datenmanagement

- **RX 2.5 Back-Pressure Basics**
  Funktionsweise von Back-Pressure verstehen — Mittel
- **RX 2.6 Stream-Source ohne Back-Pressure bauen**
  Flowable.create() — Mittel
- **RX 2.7 Stream-Source mit Back-Pressure bauen**
  Flowable.generate() — Schwer
- **RX 2.8 Cold und Hot Sources**
  Unterschiede zwischen Datenquellen verstehen — Mittel

Übungen zu Kapitel 3:
Concurrency und Parallelität in RxJava

SIEMENS

Übungsthema 1: API für Parallelität und Nebenläufigkeit

- **RX 3.1: API für Parallelität und Nebenläufigkeit** — Mittel
  Einsatz von subscribeOn() und observeOn()
- **RX 3.2: Scheduler für unterschiedliche Anforderungen** — Mittel
  Verhalten von Schedulers.computing und Schdulers.io()

Übungsthema 2: Nebenläufigkeit und Nichtfunktionale Anforderungen

- **RX 3.3 Performance durch Nebenläufigkeit** — Mittel
  Stream durch observeOn() beschleunigen
- **RX 3.4 Back-Pressure: Concurrency and Memory Consumption** — Schwer
  Back-Pressure einsetzen, um kritische Situationen zu beheben

Übungsthema 3: Parallelität durch Verzweigen und Zusammenführen

- **RX 3.5 groupBy()-flatMap() Parallelisierung** — Schwer
  Parallelisierung durch explizite Verzweigung
- **RX 3.6: Parallel Performance** — Mittel
  Performancesteigerung mit flatMap() und groupBy()

Übungen zu Kapitel 4:
Anforderungen und Architekturen in RxJava

SIEMENS

Übungsthema 1: Nicht-Funktionale Anforderungen

- **RX 4.1 Skalierbarkeit**
  Langsame und schnelle Subscriber behindern sich nicht

Schwer

- **RX 3.6: Parallel Performance (Wdh aus Kapitel 3)**
  Performancesteigerung mit flatMap() und groupBy()

Mittel

Übungsthema 2: Architektur von Reactive-Streams RxJava

Schwer

- **RX 1.3 Publisher, Subscription, Subscriber bauen (Wdh aus Kapitel 1)**
  Komplette Pipeline programmieren, Regeln implementieren

RX 1.1: Minimal Rx-Stream
Einen ganz einfachen RxJava Stream bauen

Einfach

SIEMENS

Bauen Sie einen einfachen Stream mit RxJava.

Ein Observable versendet die Zahlen 1-100, ein Subscriber empfängt sie, und schreibt sie auf die Shell.

- **Hinweis:**
  - Observable.range(1, 100) gibt ein Observable zurück, das die Daten emittiert
  - Observable.subscribe(System.out::printl) realisiert einen Subscriber, der die Daten auf die Shell schreibt**.**

- **Lernziel:** Stream Programmierung kennenlernen

- **Musterlösung:** seminar_reactive_streams/exercises/rx_1_standard/rx_1_1_minimal_stream_rx

RX 1.2 Subscriber API und Lifecycle
org.reactivestreams.Subscriber implementieren

Mittel

SIEMENS

Programmieren Sie einen Subscriber.
Implementieren Sie dafür das Interface org.reactivestreams.Subscriber.

a) Subcribieren Sie Ihren Subscriber an ein RxJava Flowable.range(1,100)

b) Zeigen Sie durch Shell-Ausgaben, wann welche Subscriber Methoden aufgerufen werden.

HINWEIS:

- Sie müssen sich die Subscription „merken“, die im onSubscribe() Aufruf übergeben wird
- Sie müssen Subscription.request() aufrufen, damit Daten ausgesendet werden.

- **Lernziel:** Subscriber API des Reactive-Streams Standards und deren Lifecycle kennenlernen

- **Musterlösung:** seminar_reactive_streams\exercises\rx_1_standard\rx_1_2_subscriber_api

RX 1.3 Publisher, Subscription, Subscriber bauen
Komplette Pipeline programmieren

SIEMENS

Schwer

Implementieren Sie die drei Interfaces aus der Klasse concurrent.Flow (Publisher, Subscription, Subscriber) und bauen Sie aus diesen eine Pipeline, die einen unendlichen Datenfluss verarbeitet.

In der Ausgangsbasis zur Übung ist die Pipeline vorhanden. Beheben Sie folgende Fehler:

a) In der Methode MySimpleSubscription.request() werden onNext() Aufrufe synchron abgesetzt, das führt zu einem Stack Overflow. Dieses Verhalten widerspricht der Regel 3.3: „Subscription.request MUST place an upper bound on ... recursion between Publisher and Subscriber."
Beheben sie dies durch Nutzung eines Executors (das ist ein Thread-Pool)

b) In der Methode MySipmleSubscriber.onNext(Integer item) wird bei jedem Aufruf nur ein einziges Element über den Aufruf request(1) angefordert.
Dies widerpricht der Empfehlung aus Regel 1.1, da es ein sogenanntes Stop-And-Wait Protokoll implementiert, was nicht effizient ist.
Beheben Sie dies, indem Sie einen Batch neuer Items bestellen (request(BUFFER_SIZE)).

- **Lernziel:** Leistung der Pipeline-Implementierung hinsichtlich Nebenläufigkeit erkennen
- **Musterlösung:** seminar_reactive_streams\exercises\rx_1_standard\rx_1_3_pipeline_impl

Siehe auch: https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html

RX 1.4 JDK Flow Simple Stream
Stream mit JDK Flow bauen

Einfach

SIEMENS

Bauen Sie einen einfachen Stream mit den Mitteln des JDK, der Zahlen emittiert und auf die Shell schreibt.

- Benutzen Sie als Publisher die Klasse:
  - `SubmissionPublisher`
- Versenden Sie über den Publisher die Zahlen 1-100
  - `SubmissionPublisher.submit()`
- Registrieren Sie einen Subscriber, der die Zahlen empfängt und auf die Shell schreibt:
  - `SubmissionPublisher.consume()`

- **Lernziel:** JDK Version der Reactive-Streams API kennenlernen

- **Musterlösung:** seminar_reactive_streams\exercises\rx_1_standard\rx_1_4_jdk_flow

RX 2.1: Simple Stream
Stream mit einer oder mehreren Stufen bauen

Einfach

SIEMENS

Bauen Sie einen einfachen Stream mit Observable oder Flowable als Source.

a. Programmieren Sie eine Source, die die Zahlen 1-100 versendet, und einen Subscriber der diese empfängt und auf die Konsole schreibt.

b. Bauen Sie mehrere Stufen in den Stream ein, nutzen Sie map() um die Quadratzahlen zu bilden und filter() um die geraden Quadratzahlen zu finden

c. Addieren Sie die Zahlen von 1-100, nutzen Sie dafür reduce()

- **Lösungshinweis**

```
Flowable.range(0, 1000)
.map(i -> i/2)
.filter (i -> (i%2) == 0)
.subscribe(System.out::println);
```

- **Lernziel:** Das RxJava Stream-API kennenlernen
- **Musterlösung:** seminar_reactive_streams\exercises\rx_2_api\rx_2_1_stream_api

RX 2.2 Multipe Subscribers: Verzweigen des Streams durch Anmelden mehrerer Subscriber

SIEMENS

Mittel

Verzweigen Sie einen Stream, indem Sie mehrere Subscriber anmelden. Source ist z.B. Observable.generate(1, 100).

a) In welcher Reihenfolge und in welchem Thread laufen die Subscriber ab?

b) Wie laufen die Subscriber, wenn sie subscribeOn() verwenden?

- **Lösungshinweis:** Um den Thread zu sehen, der einen Subscriber ausführt:

```
.subscribe(
  i ->  System.out.println("subscribe() " +  i  + " " +
    Thread.currentThread()));
```

- **Lernziel:** Nebenläufigkeitsverhalten von Streams und die Wirkung von subscribeOn() kennenlernen

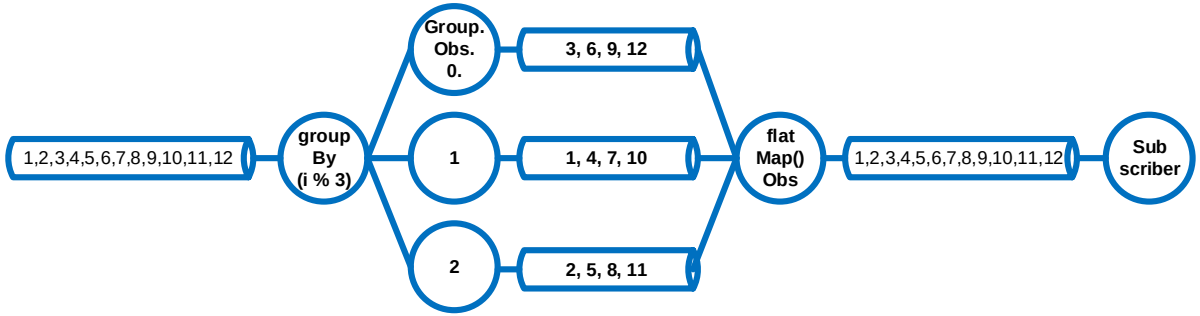
- **Musterlösung:** seminar_reactive_streams\exercises\rx_2_api\rx_2_2_multiple_subscribers

RX 2.3 flatMap() - groupBy(): Verzweigen und Zusammenführen als Basis für die Parallelisierung

Schwer

SIEMENS

- Programmieren Sie einen Stream und gruppieren Sie dessen Daten mit groupBy() zu Gruppen. Verarbeiten Sie jede dieser Gruppen. Fügen Sie die Resultate mit flatMap() wieder zu einem einzigen Stream zusammen.



- **Schrittweise Lösung**
  - Erzeugen Sie mit Flowable.range(1,100) ein Flowable, das 100 Elemente emittiert.
  - Gruppieren Sie diese Elemente mit groupBy(i -> i% 4) in vier Gruppen
  - Führen Sie eine triviale map-Operation auf den Elementen jeder Gruppe aus, z.B: map(i -> i)
  - Benutzen Sie flatMap() um die Gruppen wieder zusammenzuführen
  - Erzeugen Sie Shell-Output, der zeigt, dass jedes Element in genau einer Gruppe verarbeitet wird.

- **Lernziel:** Verzweigen und Zusammenführen als Basis für die Parallelisierung von Streams kennen
- **Musterlösung:**seminar_reactive_streams\exercises\rx_2_api\rx_2_3_flatmap_groupby_split_merge

RX 3.1: API für Parallelität und Nebenläufigkeit Einsatz von subscribeOn() und observeOn()

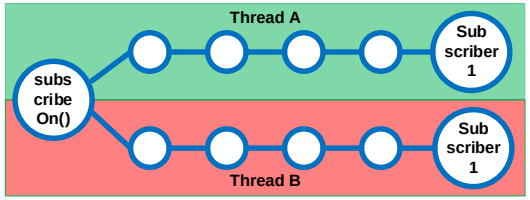
SIEMENS

Mittel

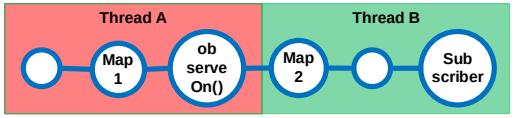
Bauen Sie einen Stream mit vier Stufen:
Flowable.range().map().map().subscribe()

a) **Sequentialität**: Zeigen Sie durch Shell-Ausgaben, dass dieser sequentiell im Main Thread abläuft. Verändern Sie dieses Verhalten durch subscribeOn().

b) **subscribeOn()-Parallelität**: Melden Sie einen zweiten Subscriber an dem Stream an, und nutzen Sie subscribeOn() um beide Subscriber parallel zu bedienen. Zeigen Sie die Parallelität durch Shell-Ausgaben.



c) **observeOn()- Nebenläufigkeit**: Entkoppeln Sie die beiden map() Stufen durch Einsatz von observeOn() voneinander. Zeigen Sie durch Shell-Ausgaben, dass die Stufen asynchron zueinander laufen.



- **Lernziel**: Grundlegende Rx API für Parallelität und Nebenläufigkeit verstehen
- **Musterlösung**: seminar_reactive_streams\exercises\rx_3_parallel\rx_3_1_parallel_api

RX 2.4 groupBy() Stand Alone

Mittel

SIEMENS

Teilen Sie einen Stream mittels groupBy() in Gruppen auf, und verarbeiten Sie die Elemente dieser Gruppen.

(Das ist in Isolation zwar nicht sinnvoll, aber es erleichtert das Verständnis von groupBy())

Schrittweise Lösung

- Gruppieren Sie diese Elemente eines Streams mit groupBy(i -> i% 4) in vier Gruppen
- Führen Sie eine triviale map-Operation auf den Elementen jeder Gruppe aus, z.B: map(i -> i)
- Erzeugen Sie Shell-Output, der zeigt, dass jedes Element in genau einer Gruppe verarbeitet wird.
- Subskribieren Sie Dummy-Subscriber an den Gruppen
- Subskribieren Sie einen Subscriber an groupBy(), um den Ablauf zu starten

- **Lernziel:**
  Verstehen, wie groupBy() in Isolation funktioniert, um die Leistung von flatMap() zu erkennen

- **Musterlösung:** seminar_reactive_streams\exercises\rx_2_api\rx_2_4_groupby_standalone

RX 2.5 Back-Pressure Basics
Funktionsweise von Back-Pressure verstehen

Mittel

SIEMENS

Zeigen Sie, wie Back-Pressure das Verhalten eines Streams verändert

Programmieren Sie dafür einen Stream mit folgenden Eigenschaften

- Stream Source erzeugt Integers 1 ... 100_000_000
- map() - Schritt bearbeitet die Daten
- observeOn() entkoppelt map() vom Subscriber
- Einen Subscriber anmelden, und diese **verlangsamen** (z.B. durch Thread.sleep() oder println)
- Fügen Sie Shell-Ausschriften ein, um das Verhalten von map() und onNext() zu zeigen.

a) Die Stream Source ist ein Observable. Zeigen Sie, dass Source und Subscriber beliebig weit auseinanderlaufen.

b) Die Stream Source ist ein Flowable. Zeigen Sie, dass Source und Subscriber nur geringfügig auseinander laufen, nämlich nie weiter als 128 Elemente.

- **Lernziel:** Verhalten von Back-Pressure verstehen

- **Musterlösung:** seminar_reactive_streams\exercises\rx_2_api\rx_2_5_back_pressure_basics

RX 2.6 Stream-Source ohne Back-Pressure bauen Flowable.create()

SIEMENS

Mittel

Schreiben Sie eine Source für einen Stream ohne Back-Pressure.

a) Schreiben Sie die Stream-Source mit Flowable.create(), der die Zahlen 1 bis 1_000_000 emittiert.

Programmierbeispiel

```
Flowable<Integer> source = Flowable.create(emitter -> {
  for(int i = 0; i < 200; ++i) {
    emitter.onNext(i);
  }
},BackpressureStrategy.BUFFER);
```

b) Zeigen Sie, dass diese Source keinen Back-Pressure realisiert, selbst wenn Sie einen Parameter für die BackpressureStrategy übergeben.

- **Lernziel**: Stream Source ohne Back-Pressure erzeugen können
- **Musterlösung:** seminar_reactive_streams/exercises/rx_2_api/rx_2_6_genrator_no_backpressure

RX 2.7 Stream-Source mit Back-Pressure bauen Flowable generate()

SIEMENS

Schwer

Erzeugen Sie eine Stream Source unter Verwendung von Flowable.generate(), die eine Zahlenfolge emittiert.

a) Erzeugen Sie die Stream Source nach folgendem Pattern.

```
Flowable<Integer> sourceFlowable =
Flowable.generate(
  () -> new AtomicInteger(0),
  (state, emitter) -> {
      emitter.onNext(state.incrementAndGet());
    }
);
```

a) Zeigen Sie, dass diese Stream-Source Back-Pressure realisiert

Lösungshinweise

- Benutzen Sie den Emitter, um mit onNext() Daten in den Stream zu pushen.
- Machen Sie sich klar, der Consumer, den Sie übergeben, für jedes angeforderte Element ein Mal aufgerufen wird
- Für den Back-Pressure brauchen Sie observeOn() im Stream zur Entkopplung
- Verlangsamen Sie den Subscriber, um Back-Pressure sichtbar zu machen:
  Map und Subscribe sollten nicht weiter als 128 Elemente auseinanderlaufen

- **Lernziel**: Verstehen, wie man eine Datenquelle mit Back-Pressure selbst implementiert
- **Musterlösung**: seminar_reactive_streams\exercises\rx_2_api\rx_2_7_generator_with_backpressure

RX 2.8 Cold und Hot Sources
Unterschiede zwischen Datenquellen

SIEMENS

Mittel

Machen Sie aus einer Cold-Datenquelle eine Hot-Datenquelle und zeigen Sie die Unterschiede

a) Zeigen Sie, dass bei einer Cold Datenquelle, z.B: Flowable.range(), alle Clients identische Daten erhalten, unabhängig vom Zeitpunkt ihrer Anmeldung (*).

b) Machen Sie aus dieser Datenquelle eine Hot Datenquelle, und zeigen Sie, dass nun ein später angemeldeter Subscriber weniger Daten erhält, als ein früh angemeldeter Subscriber. Verwenden Sie folgendes API:
   - publish(): Um die eine Cold Datenquelle in eine Hot Datenquelle zu verwandeln
   - connect(): Um die Emittierung der Daten zu starten

Lösungsschritte

- Datenquelle „Hot“ machen: publish()
- Ersten Subscriber anmelden
- connect() aufrufen, um die Emittierung der Daten zu starten
- Eine Zeit lang warten
- Zweiten Subscriber anmelden
- Im Output sichtbar machen, dass der Zweite Subscriber nicht alle Daten erhält, die der erste erhalten hat

- **Lernziel**: Unterschied zwischen einer Cold und einer Hot Datenquelle verstehen
- **Musterlösung**: seminar_reactive_streams\exercises\rx_2_api\rx_2_8_cold_hot_datasource

* Das ist nicht ganz korrekt, genügt hier aber als Annäherung. Genau genommen werden bei einer Cold Datenquelle, die Daten für jeden Subscriber gesondert generiert

RX 3.1: API für Parallelität und Nebenläufigkeit Einsatz von subscribeOn() und observeOn()

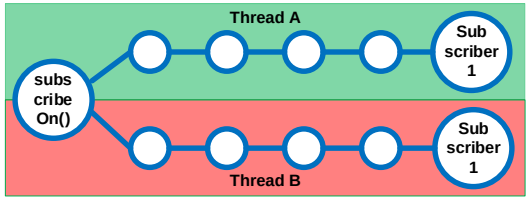
SIEMENS

Mittel

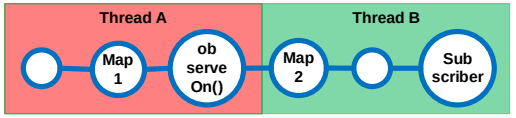
Bauen Sie einen Stream mit vier Stufen:
Flowable.range().map().map().subscribe()

a) **Sequentialität**: Zeigen Sie durch Shell-Ausgaben, dass dieser sequentiell im Main Thread abläuft. Verändern Sie dieses Verhalten durch subscribeOn().

b) **subscribeOn()-Paralleliät**: Melden Sie einen zweiten Subscriber an dem Stream an, und nutzen Sie subscribeOn() um beide Subscriber parallel zu bedienen. Zeigen Sie die Parallelität durch Shell-Ausgaben.



c) **observeOn()- Nebenläufigkeit**: Entkoppeln Sie die beiden map() Stufen durch Einsatz von observeOn() voneinander. Zeigen Sie durch Shell-Ausgaben, dass die Stufen asynchron zueinander laufen.



- **Lernziel**: Grundlegende Rx API für Parallelität und Nebenläufigkeit verstehen
- **Musterlösung**: seminar_reactive_streams\exercises\rx_3_parallel\rx_3_1_parallel_api

RX 3.2: Scheduler für unterschiedliche Anforderungen Schedulers.computing und Schdulers.io()

SIEMENS

Mittel

a) Zeigen Sie, dass der Scheduler Schedulers.computation() nur eine feste Anzahl von Threads verwendet, auch wenn mehr Aufgaben als Threads vorhanden sind.

b) Zeigen Sie, dass Schedulers.io() sich anders verhält, indem er weitere Threads startet

Verwenden Sie dafür folgendes Vorgehen

- Bauen Sie einen Stream, der Schedulers.computaion() verwendet (subscribeOn(Schedulers.computation()))
- Melden Sie an diesem Stream viele Subscriber an, mehr als Ihr Rechner CPU-Kerne hat, z.B. 10 Subscriber bei 8 Kernen.
- Sorgen Sie dafür, dass jeder Subscriber sehr lange läuft, eventuell unendlich lange.
- Zeigen Sie, dass nicht alle Subscriber zum Zuge kommen.
- Verändern Sie dieses Verhalten, indem Sie Schedulers.io() anstelle von Schedulers.computation() verwenden.
- Welches Risiko besteht aber nun?

- **Lernziel**: Das unterschiedliche Verhalten der beiden Scheduler in Grenzsituationen verstehen

- **Musterlösung**: seminar_reactive_streams\exercises\rx_3_parallel\rx_3_2_scheduler

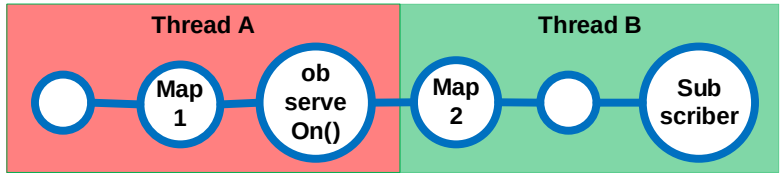
RX 3.3 Performance durch Nebenläufigkeit
Stream durch observeOn() beschleunigen

Mittel

SIEMENS

Erzeuge Sie einen Stream, mit lang laufenden Stream-Stufen. Entkoppeln Sie diese Stufen durch observeOn(). Zeigen Sie, dass ihr Stream sich dadurch beschleunigt.

Benutzen Sie als Ausgangsstream z.B:
Observable.range(0,100).map(i->i).map(i->i)
Gehen Sie dann in folgenden zwei Schritten vor:



- **a)** Sorgen Sie dafür, dass die beiden Map-Stufen asynchron zueinander ausgeführt werden. Zeigen Sie das durch Shell-Ausgaben, die die jeweiligen Threads ausgeben.
  Lösungshinweis: observeOn(Scheduler.computing())

- **b)** Fügen Sie in beide Map-Stufen eine langlaufende Funktion ein, und zeigen Sie, dass die asynchrone Version schneller ist, als die sequentielle. Wie viel schneller kann sie werden?
  Lösungshinweis: Sie können die langlaufende Funktion cpuIntesiveCall() aus der Musterlösung verwenden.

- **Lernziel**: Erkennen, dass ein asynchroner Stream einen Performance-Vorteil bieten kann
- **Musterlösung:**
  seminar_reactive_streams\exercises\rx_3_parallel\rx_3_3_concurrency_performance

RX 3.4 Back-Pressure: Concurrency & Memory
Kritische Situationen beheben

SIEMENS

Schwer

a) Zeigen Sie, dass asynchrone Stufen in einem Stream dazu frühen können, dass unbegrenzt viele Elemente in einer Queue gehalten werden.

b) Zeigen Sie, dass dies zum Absturz des Programmes frühen kann, wenn diese Elemente viel Memory beanspruchen.

c) Beheben Sie dies durch Back-Pressure.

- **Lösungshinweise**
  - Erzeugen Sie eine Pipeline mit einem Observable als Source: Observable.range(1, 100_000_000_0)
  - Geben Sie in einer ersten Map-Stufe für jedes Element des Streams ein speicherintensives Objekt zurück, z.B. int[10000].
  - Geben Sie der Pipeline eine zweite Map-Stufe, und verlangsamen Sie diese
  - Entkoppeln Sie die beiden Stufen mit .observeOn(Schedulers.*computation())*
  - Zeigen Sie, dass Sie ihr Programm dadurch zum Absturz bringen können (OutOfMemoryError)
  - Ersetzen Sie nun das Observable durch ein Flowable und beheben damit das Problem
- **Lernziel**: Erkennen, dass Back-Pressure ein fundamentales Mittel ist, um Risiken hinsichtlich des Ressourcenverbrauches entgegenzuwirken
- **Musterlösung**: seminar_reactive_streams\exercises\rx_3_parallel\rx_3_4_concurrency_memory_backpressure

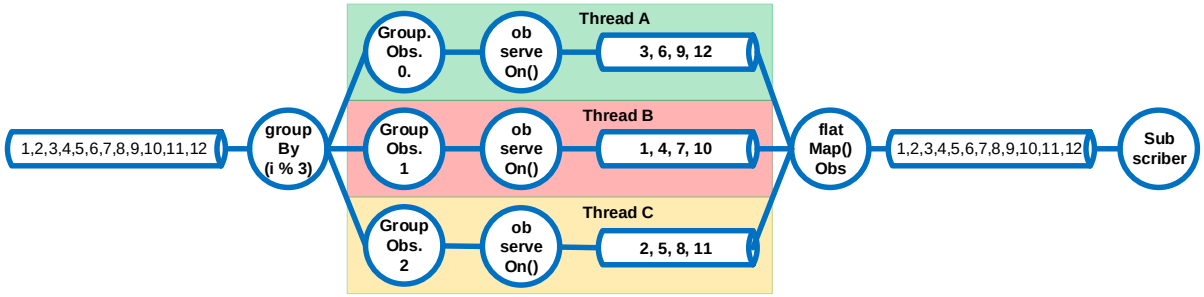
RX 3.5 groupBy()-flatMap() Parallelisierung durch Verzweigung und Zusammenführung

Schwer

SIEMENS

Parallelisieren Sie den Stream Observable.range(1,100).map(i -> i).subscribe(System.out::println), indem Sie Gruppen bilden, und diese Gruppen in eigenen Threads verarbeiten.

Zeigen Sie die Parallelität durch Shell-Ausgaben.



Lösungshinweis: Verwenden Sie folgende APIs

- gorupBy() zerlegt einen Stream in mehrere GroupedObservables
- flatMap() führt die GroupedObservables wieder zu einem Observable zusammen
- observeOn() führt jede diese Gruppen in einem eigenen Thread aus

- **Lernziel**: Parallelisierung von Streams durch Verzweigen und Zusammenfürhen verstehen.

- **Musterlösung**: seminar_reactive_streams/exercises/rx/parallel/rx_3_5_parallel_flatmap_groupby

RX 3.6: Parallel Performance mit flatMap() und groupBy()

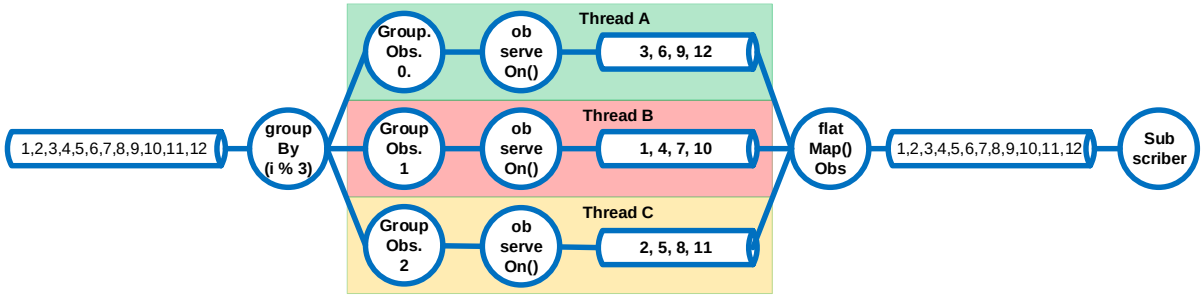
Mittel

SIEMENS

Zeigen Sie, dass sich ein Stream durch Parallelisierung um den Faktor beschleunigen lässt, der der Anzahl der CPU-Cores (z.B. 8) in Ihrem Rechner entspricht.

Anleitung zur Lösung:

- Bauen Sie einen Stream, der die Zahlen 0-100 emittiert
- Führen Sie einen Map-Schritt mit zeitaufwendiger Berechnung ein, z.B. cpuIntensiveCall(100) aus der Musterlösung. Der Stream läuft nun ca. 10 Sekunden.
- groupBy(): Zerlegen Sie den Stream in 8 Substreams, unter Verwendung von groupBy()
- observeOn() Parallelisierung: Führen Sie jede Gruppe in einem eigenen Thread aus
- flatMap(): Führen Sie die Gruppen wieder zusammen
- Ihr Stream sollte nun nur ca. 1-2 Sekunden lang laufen



- **Lernziel**: Performance-Potential der Stream-Parallelisierung erkennen.
- **Musterlösung**: Ausgangsbasis liegt in **EASY** und **HARD** Variante vor. seminar_reactive_streams\exercises\rx_3_parallel\rx_3_6_parallel_flatmap_groupby_performance

Übung RX 4.1 Skalierbarkeit: Langsame und schnelle Subscriber behindern sich nicht

SIEMENS

Schwer

Langsame Subscriber sollten schnelle Subscriber nicht behindern, wenn beide bei dem selben Publisher subskribiert sind. Zeigen Sie das Problem, und finden Sie eine Lösung.

- 1. **PROBLEM**: Zeigen Sie, dass es passieren kann, dass ein potentiell schneller Subscriber nicht laufen kann, weil viele "langsame" Subscriber Threads nicht freigeben. „Langsam" bedeutet hier, dass in onNext() eine lang laufende Aufgabe bearbeitet wird.
- 2. **LÖSUNG**: Finden Sie eine Lösung, die sicherstellt, dass der schnelle Subscriber ungehindert laufen kann, auch wenn mehrere langsame Subscriber beim Publisher angemeldet sind.

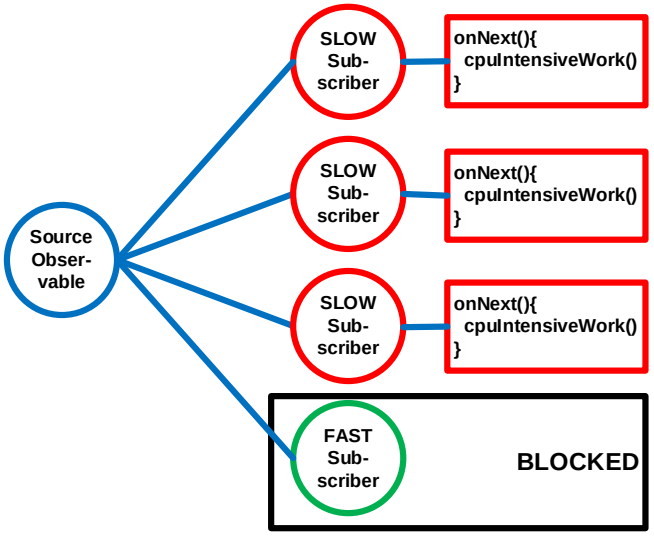
- **Lösungshinweise**
  - zu 1: Starten Sie zuerst viele „langsame“ Subscriber, die in onNext() eine CPU-intensive Methode aufrufen, starten Sie dann einen „schnellen“ Subscriber
  - zu 2: Implementieren Sie eine Subscriber Klasse, die in onNext() die langlaufende Methode in einen anderen Thread auslagert und sofort zurückkehrt. Nutzen Sie Subscription.request() um neue Elemente anzufordern. Nutzen Sie ein Flowable als Source.

- **Lernziel**: Potential von non-blocking Back-Pressure für die Skalierbarkeit verstehen

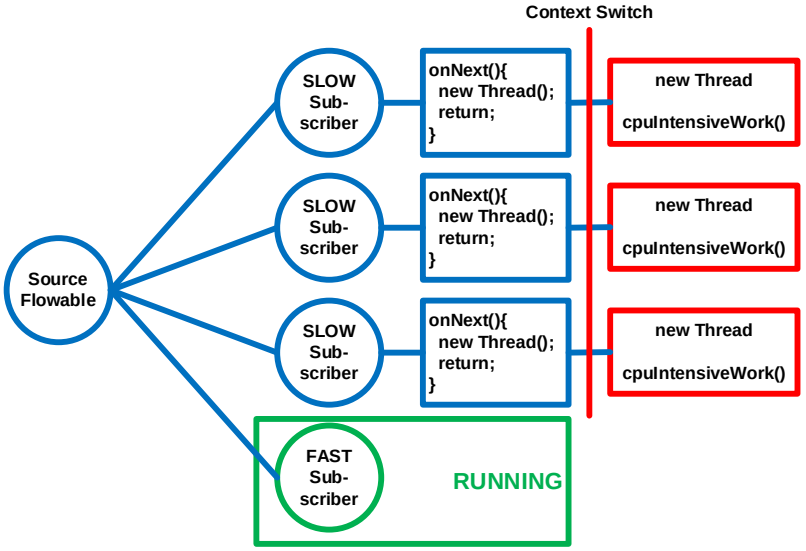
- **Musterlösung**: seminar_reactive_streams\exercises\rx_4_architecture\rx_4_1_slow_fast_subscriber_blocking

Lösungshinweise zu 4.1

- **Problem**



- **Solution**



RX 3.6: Parallel Performance mit flatMap() und groupBy()

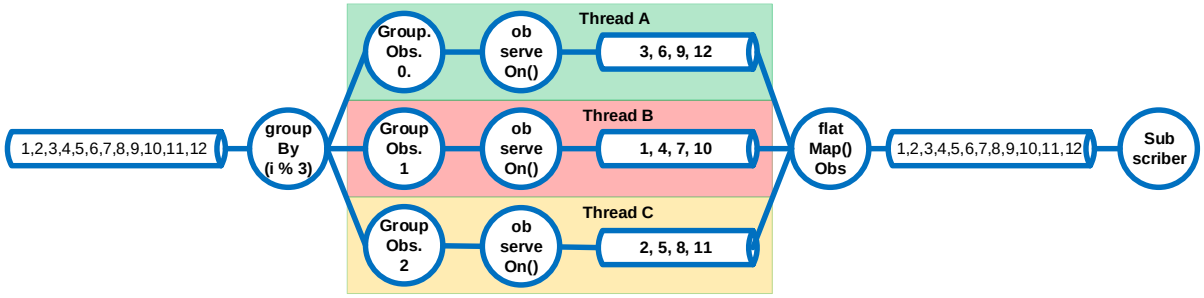
Mittel

SIEMENS

Zeigen Sie, dass sich ein Stream durch Parallelisierung um den Faktor beschleunigen lässt, der der Anzahl der CPU-Cores (z.B. 8) in Ihrem Rechner entspricht.

Anleitung zur Lösung:

- Bauen Sie einen Stream, der die Zahlen 0-100 emittiert
- Führen Sie einen Map-Schritt mit zeitaufwendiger Berechnung ein, z.B. cpuIntensiveCall(100) aus der Musterlösung. Der Stream läuft nun ca. 10 Sekunden.
- groupBy(): Zerlegen Sie den Stream in 8 Substreams, unter Verwendung von groupBy()
- observeOn() Parallelisierung: Führen Sie jede Gruppe in einem eigenen Thread aus
- flatMap(): Führen Sie die Gruppen wieder zusammen
- Ihr Stream sollte nun nur ca. 1-2 Sekunden lang laufen



- **Lernziel**: Performance-Potential der Stream-Parallelisierung erkennen.
- **Musterlösung**: Ausgangsbasis liegt in **EASY** und **HARD** Variante vor. seminar_reactive_streams\exercises\rx_3_parallel\rx_3_6_parallel_flatmap_groupby_performance

RX 1.3 Publisher, Subscription, Subscriber bauen
Komplette Pipeline programmieren

SIEMENS

Schwer

Implementieren Sie die drei Interfaces aus der Klasse concurrent.Flow (Publisher, Subscription, Subscriber) und bauen Sie aus diesen eine Pipeline, die einen unendlichen Datenfluss verarbeitet.

In der Ausgangsbasis zur Übung ist die Pipeline vorhanden. Beheben Sie folgende Fehler:

a) In der Methode MySimpleSubscription.request() werden onNext() Aufrufe synchron abgesetzt, das führt zu einem Stack Overflow. Dieses Verhalten widerspricht der Regel 3.3: „Subscription.request MUST place an upper bound on ... recursion between Publisher and Subscriber."
Beheben sie dies durch Nutzung eines Executors (das ist ein Thread-Pool)

b) In der Methode MySipmleSubscriber.onNext(Integer item) wird bei jedem Aufruf nur ein einziges Element über den Aufruf request(1) angefordert.
Dies wiederpricht der Empfehlung aus Regel 1.1, da es ein sogenanntes Stop-And-Wait Protokoll implementiert, was nicht effizient ist.
Beheben Sie dies, indem Sie einen Batch neuer Items bestellen (request(BUFFER_SIZE)).

- **Lernziel:** Leistung der Pipeline-Implementierung hinsichtlich Nebenläufigkeit erkennen
- **Musterlösung:** seminar_reactive_streams\exercises\rx_1_standard\rx_1_3_pipeline_impl

Siehe auch: https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html