HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Parallele Programmierung
Reactive Programming

Vorlesung 13

Christoph Amrein
Prof. Dr. Luc Bläser

Heute: Einfache Parallelisierung

- Automatische Parallelisierbarkeit
  - Ohne Programm-Redesign
  - Erfordert geeignetes Programmiermodell
- Einfache Skalierung
  - Durchsatz: Grössere Datenmenge bewältigen
  - Bearbeitung auf mehreren Cores ermöglichen

Welche Programmiermodelle haben wir hierzu schon behandelt?

Modelle für implizite Parallelisierung

- Imperativ
  - Task/Daten-Parallelisierung
    - Parallel.For, Parallel.Invoke
    - Asynchrone Ausführungen
  - Actor Model
- Deskriptiv
  - Datenflüsse
    - PLINQ
    - Reactive Programming ← Heutiger Fokus

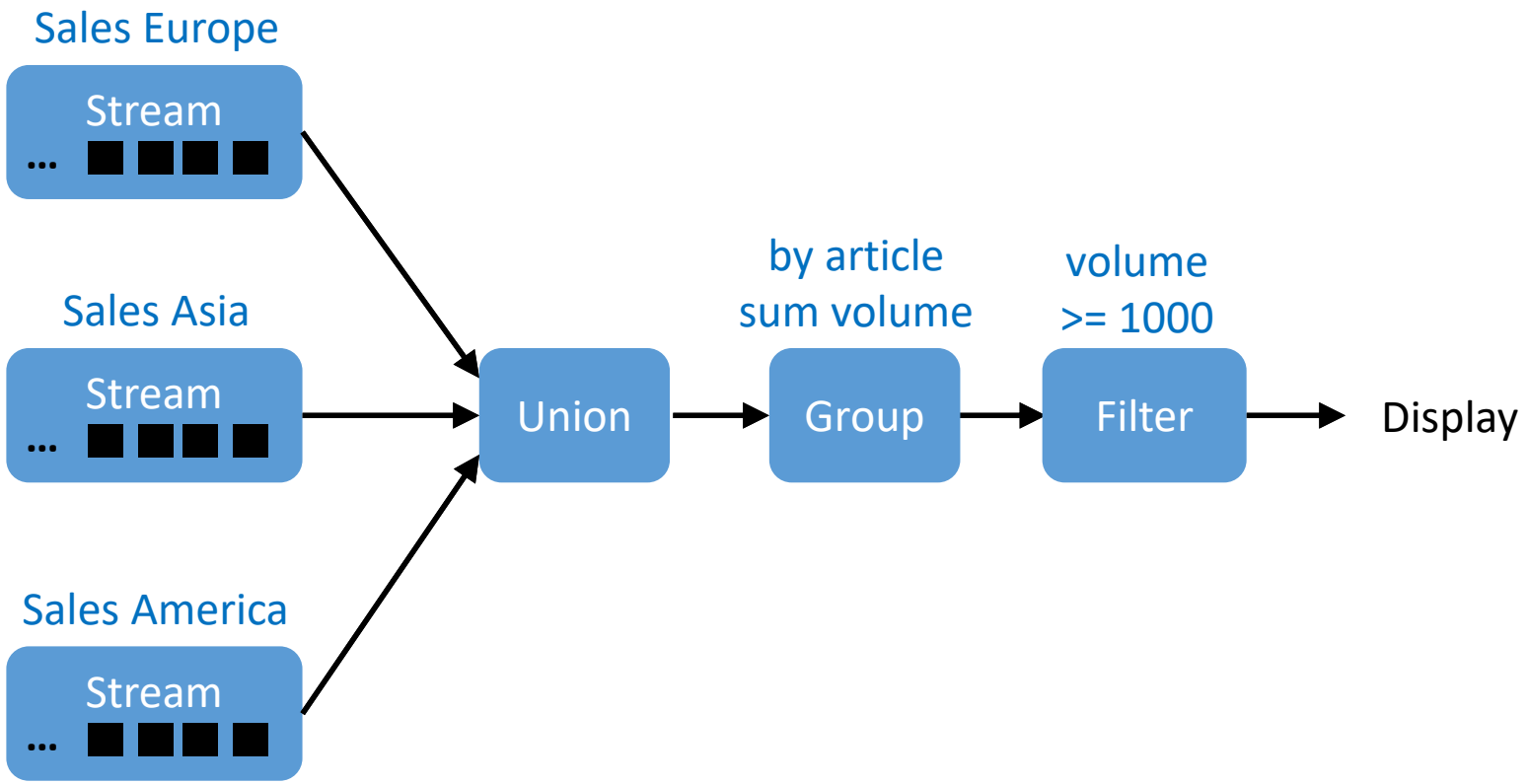
Inhalt Heute

- Reactive Programming
  - Modell und Paradigma
  - Concurrency Aspekte
  - Nutzen und Limitationen

Lernziele

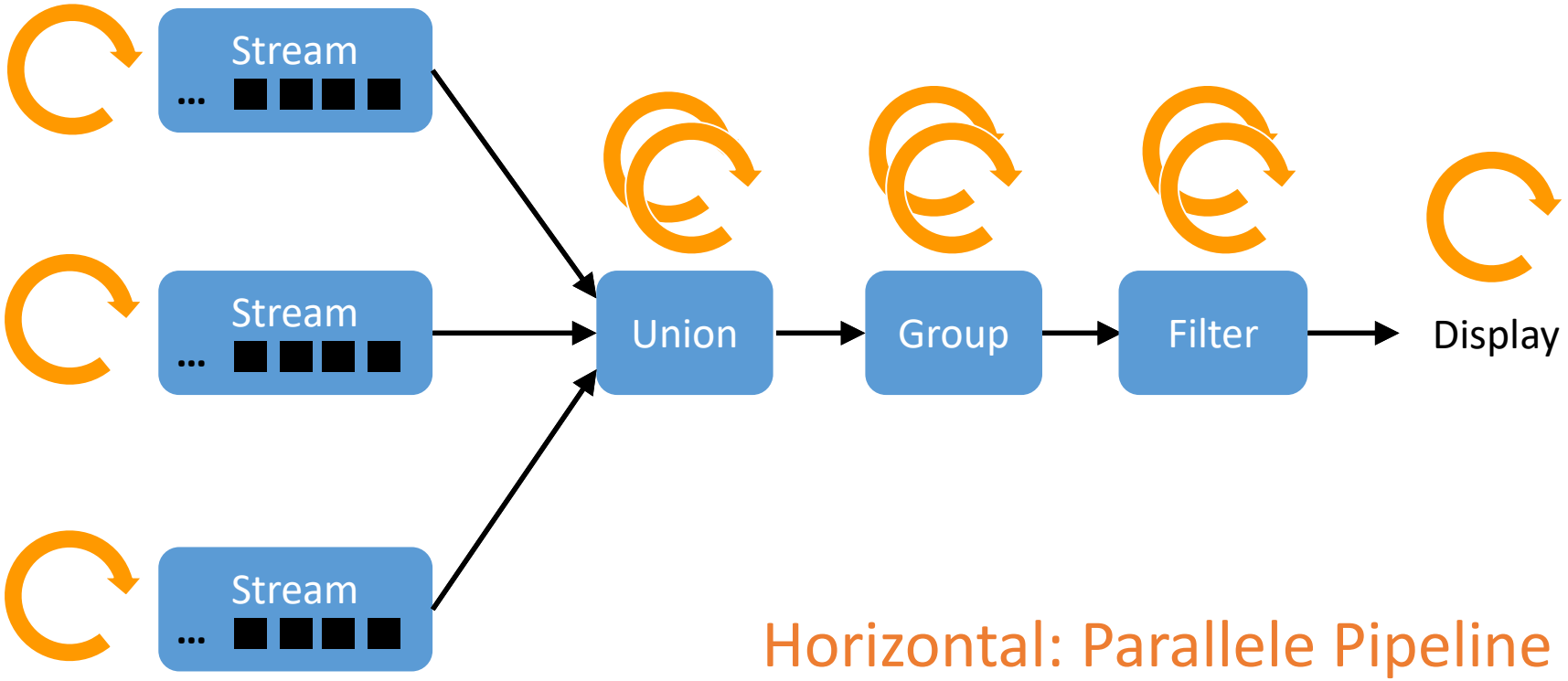
- Reactive Programming verstehen
- Am Beispiel von Reactor anwenden können
- Nutzen und Einschränkungen beurteilen

Programm-Datenflüsse



Auf welchen Stufen ist dies parallelisierbar?

Parallele Datenflüsse



Horizontal: Parallele Pipeline
Vertikal: Datenmenge teilen

Mainstream Datenfluss-Modelle

- .NET LINQ (Language-Integrated Query)
  - Eingebettete C# Syntax im Stil von SQL
  - Oder explizit über Methoden («Extension Methods»)
  - Parallelisierung auf .NET Task Parallel Library (TPL)
- Java 8 Stream API als Nachahmer
  - Keine eigene Syntax, via Methoden («Default Methods»)
  - Eingeschränkter, z.B. Gruppierung verlässt Stream API
  - Parallelisierung auf Fork Join Thread Pool

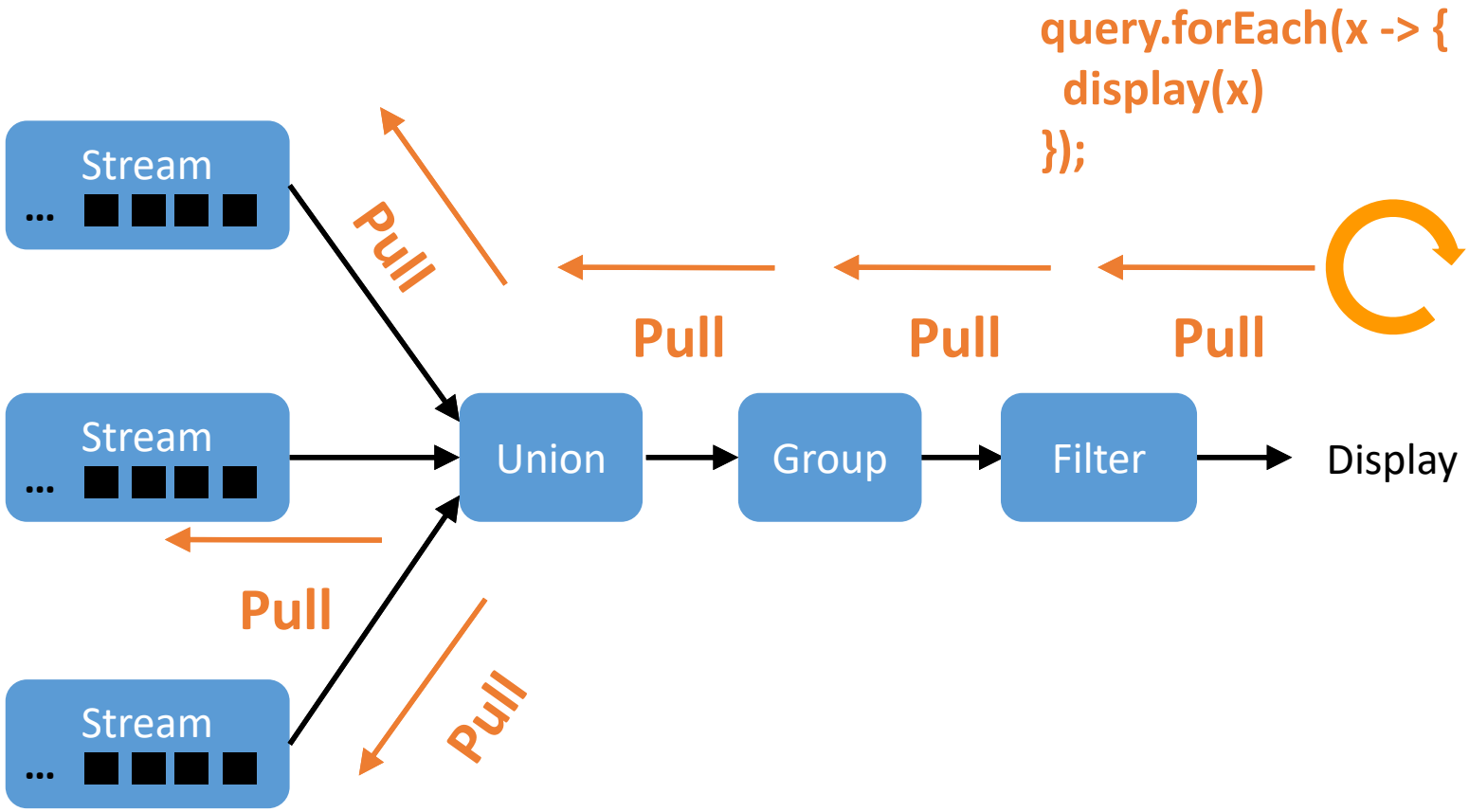
Java Stream API

```
concat(
  concat(
    salesEurope.stream(),
    salesEurope.stream()
  ),
  salesAmerica.stream()
)
.collect(
  groupingBy(
    SalesEntry::getArticle,
    summingInt(SalesEntry::getVolume)
  )
)
.entrySet().stream()
.filter(category -> category.getValue() >= 1000)
```

Parallele Streams

```
concat(
  concat(
    salesEurope.parallelStream(),
    salesEurope.parallelStream()
  ),
  salesAmerica.parallelStream()
)
.collect(
  groupingBy(
    SalesEntry::getArticle,
    summingInt(SalesEntry::getVolume)
  )
)
.entrySet().parallelStream()
.filter(category -> category.getValue() >= 1000)
```

Java Stream = Pull-Mechanismus



Pull Mechanismus

Auswertung beginnt durch Iterieren der Abfrage

- Pipeline-Schritte werden rückwärts ausgelöst

Input-Quelle ist passiv

- Input muss vollständig vorliegen
- Wird durch Query ausgelesen und verarbeitet

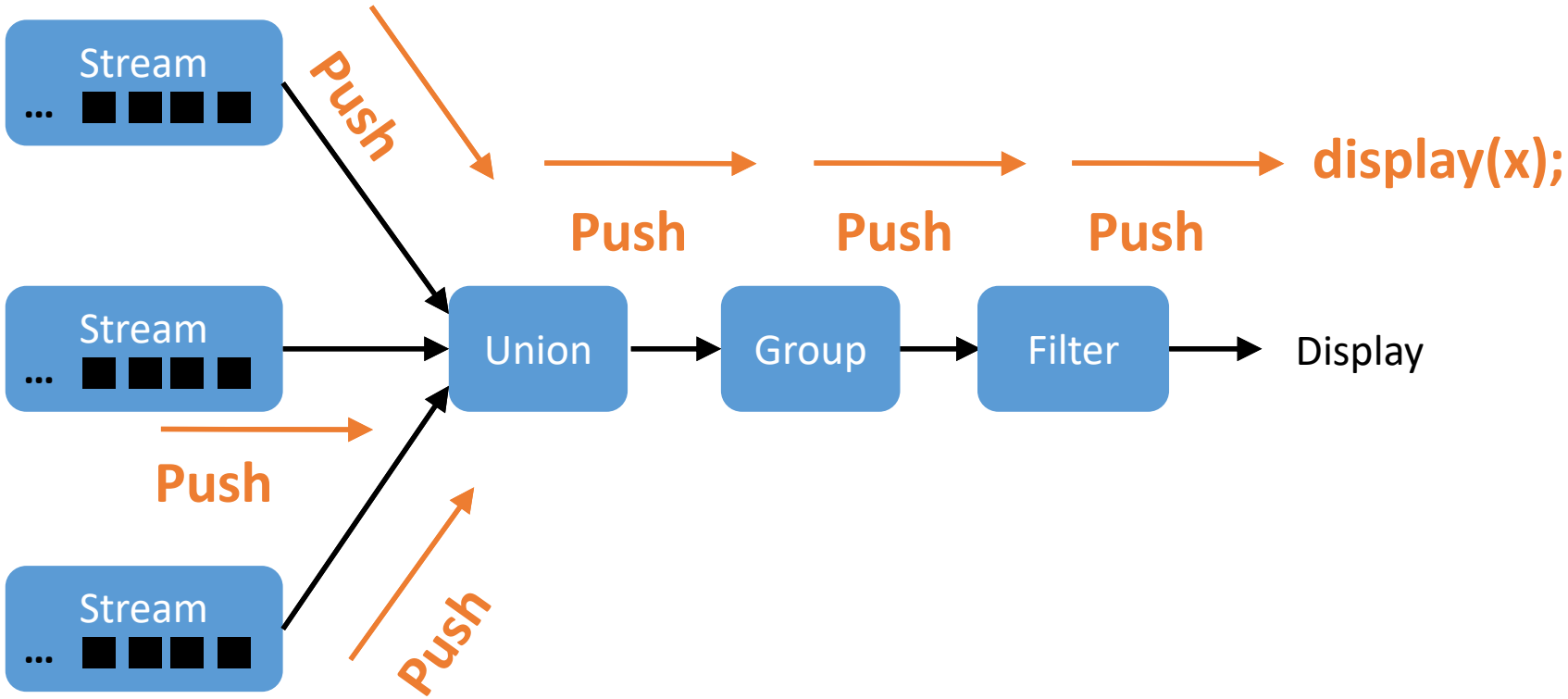
Problem: Aktive Input-Quellen

Pull-Modell funktioniert nicht, falls

- Input sukzessive mit Pausen ankommt
  - User Input, Netzwerk etc.
- Länge des Streams unbekannt bzw. unendlich ist

=> Push Modell (aka Reactive)

Reactive = Push-Mechanismus



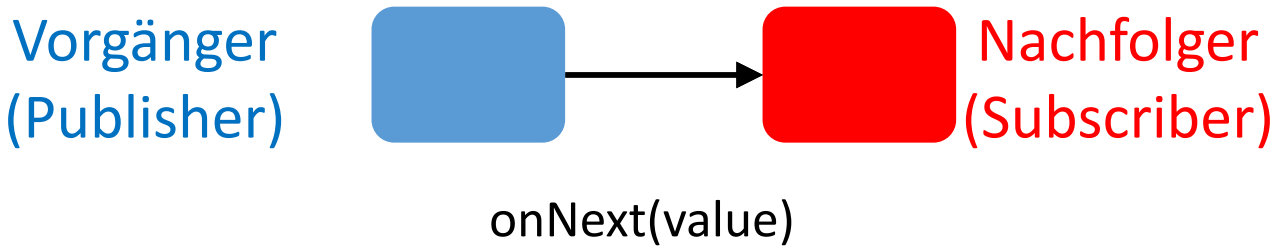
Reactive Programming

Input und Arbeitsschritte sind aktiv

- Lösen pro Wert ein Ereignis aus (onNext)

Verkettung der Arbeitsschritte

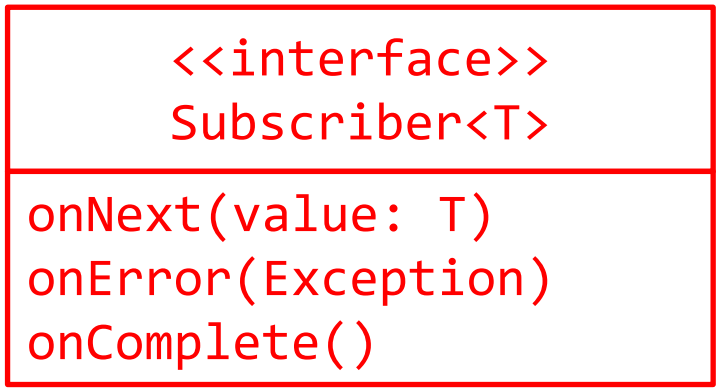
- Nachfolgeschritt abonniert Events des Vorgängers

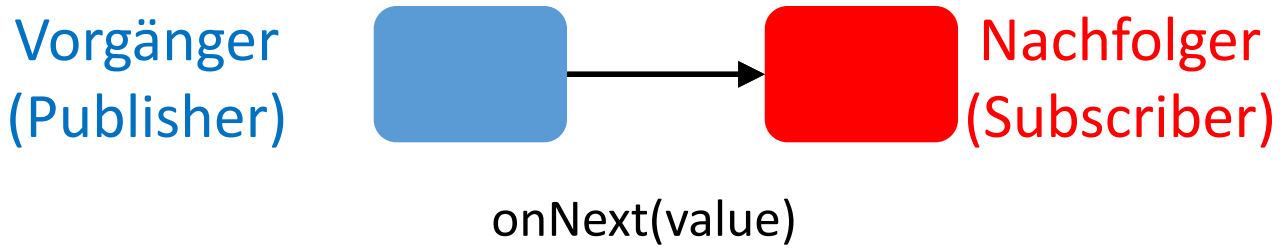


Reactive Programming

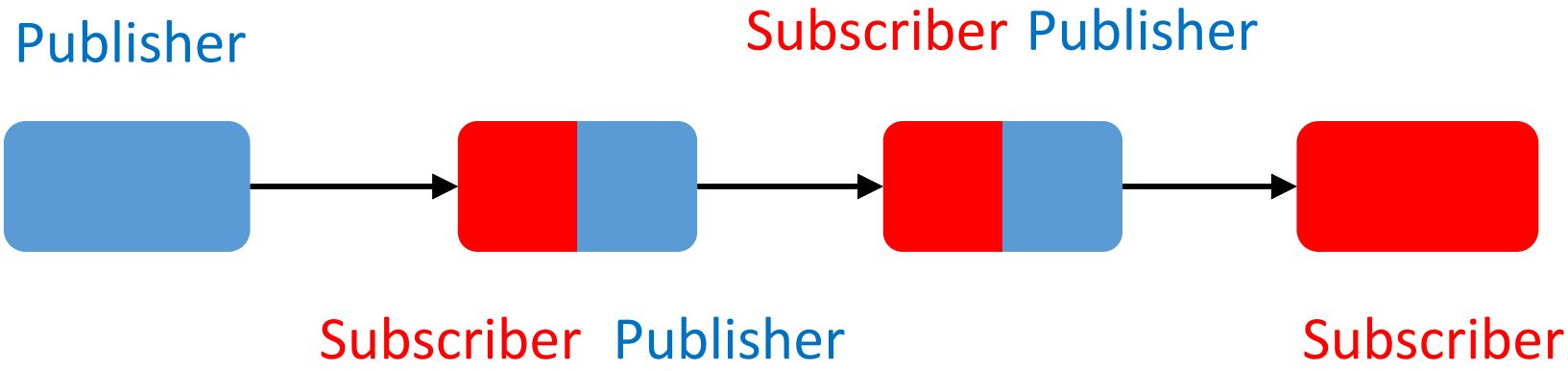
- Asynchrone Events
  - Inputs können asynchron als Events ankommen
  - Outputs sind asynchrone Events (ideal für GUI)
- Rx.NET: Reactive Programming für .NET
  - Erfinder: Erik Meijer, 2009
  - Basiert auf Datenfluss-Paradigma (SISAL 1983, VHDL etc.)
  - Verarbeitungsschritte als LINQ beschreibbar
- Project Reactor
  - Java Framework (nebst RxJava)
  - Primäre Wahl für Spring Projekte

Reactor Publisher/Subscriber Pattern





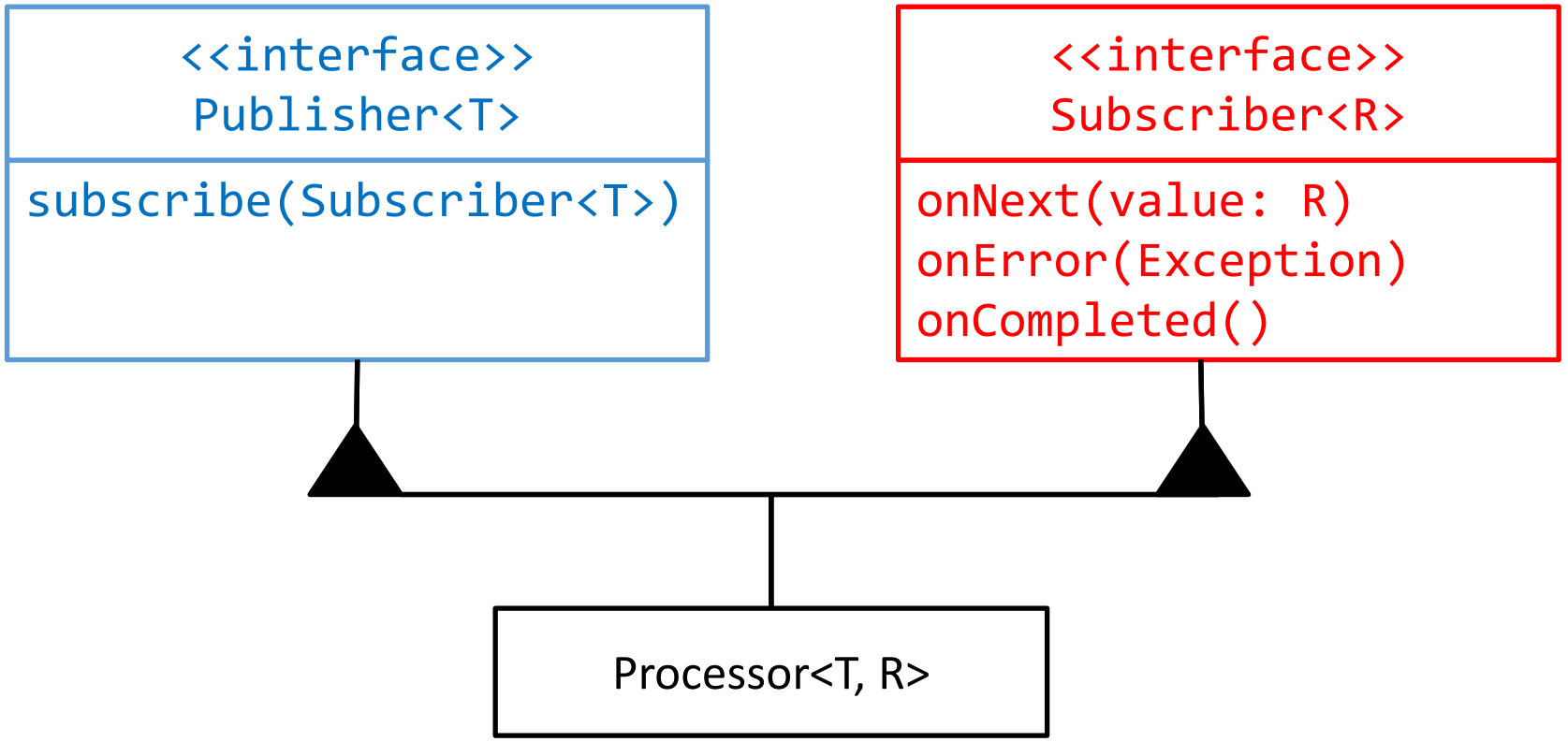
Reactive Pipelining



Zwischenschritte haben zwei Rollen (Interfaces)

- Subscriber des Vorgängers
- Publisher des Nachfolgers

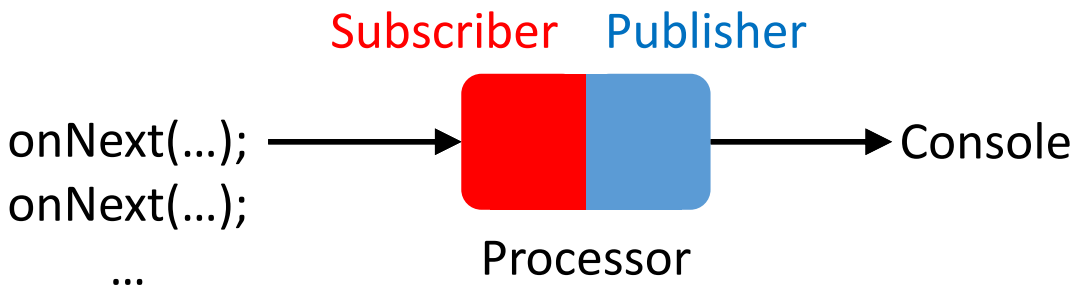
Processor = Subscriber + Publisher



Auch Promise genannt

Einfaches Reactor Beispiel

```
var processor = DirectProcessor.<String>create();

processor.subscribe(System.out::println);

processor.onNext("A");
processor.onNext("B");
processor.onNext("C");

processor.onComplete();
```



Sequenzende

Subscriber kann beliebig viele Werte erhalten

- Beliebige Verzögerungen zwischen onNext()

Ende der Sequenz

- Erfolgreich mit onComplete()
- Fehlerhaft mit onError()

Nach Ende ignorieren Prozessoren weitere Aufrufe

Ad-Hoc Subscriber Erzeugung

```
processor.subscribe(
  ConsumerForOnNext,
  ConsumerForOnError,
  RunnableForOnCompleted
);
```

onError und onCompleted optional

Beispiel:

```
processor.subscribe(
  value -> System.out.println("received: " + value),
  exception -> System.out.println("thrown: " + exception),
  () -> System.out.println("completed")
);
```

Processor Varianten

DirectProcessor

- Erlaubt max. 1 Subscriber
- Kein Buffer
- Subscriber erhält nur zukünftige Werte

ReplayProcessor

- Erlaubt mehrere Subscriber
- (Un)beschränkter Buffer
- Subscriber erhält alte Werte

ReplayProcessor

```
var processor = ReplayProcessor.<String>create();

processor.onNext("A");
processor.onNext("B");

processor.subscribe(System.out::println);

processor.onNext("C");
```

Welche Ausgabe wird erwartet?
Vorteil/Nachteil von ReplayProcessor?

Reactive Streams Specification

- Keine parallelen onNext() Aufrufe
- Keine null Werte
- Aufrufe von onNext() passieren auf dem selben Thread, ausser anders angegeben
- …

Paralleles onNext() über Sink

- Eine Sink synchronisiert die onNext() Aufrufe
- onNext() darf nicht mehr direkt verwendet werden

```
var processor = DirectProcessor.<String>create();
processor.subscribe(System.out::println);

var sink = processor.sink();
CompletableFuture.runAsync(() -> sink.next("A"));
CompletableFuture.runAsync(() -> sink.next("B"));
```

Flux und Mono

- Direkte Verwendung von Prozessoren eher unüblich
- Fluent Interface für die Definition einer Pipeline
  - Flux: 0..N Elemente
  - Mono: 0..1 Elemente

```
var flux = source
  .filter(x -> x > 1000)
  .distinct()
  .map(x -> x * x);
```

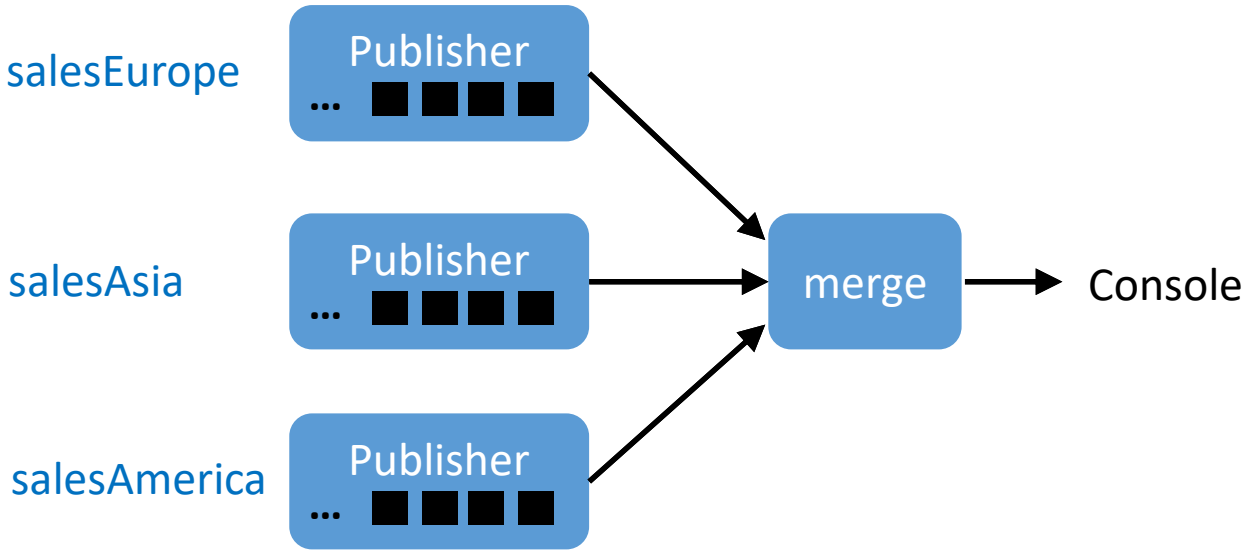
Collections zu Flux

Passives Iterable oder Stream zu aktivem Publisher umwandeln

```
Flux.fromIterable(salesEurope)

Flux.fromIterable(salesAsia)

Flux.fromStream(salesAmerica.stream())
```

Flux kombinieren

```
var combinedSales = Flux.merge(
  Flux.fromIterable(salesEurope),
  Flux.fromIterable(salesAsia),
  Flux.fromIterable(salesAmerica)
);
combinedSales.subscribe(System.out::println);
```



Reactor und Concurrency

- Default: Alles sequentiell
  - Jedoch asynchron
- Concurrency einfach einstellbar
  - Scheduler mit publishOn() angeben

```
publisher.publishOn(Schedulers.parallel())
  .subscribe(…)
```

Publisher-Aufrufe werden in Thread-Pool ausgeführt

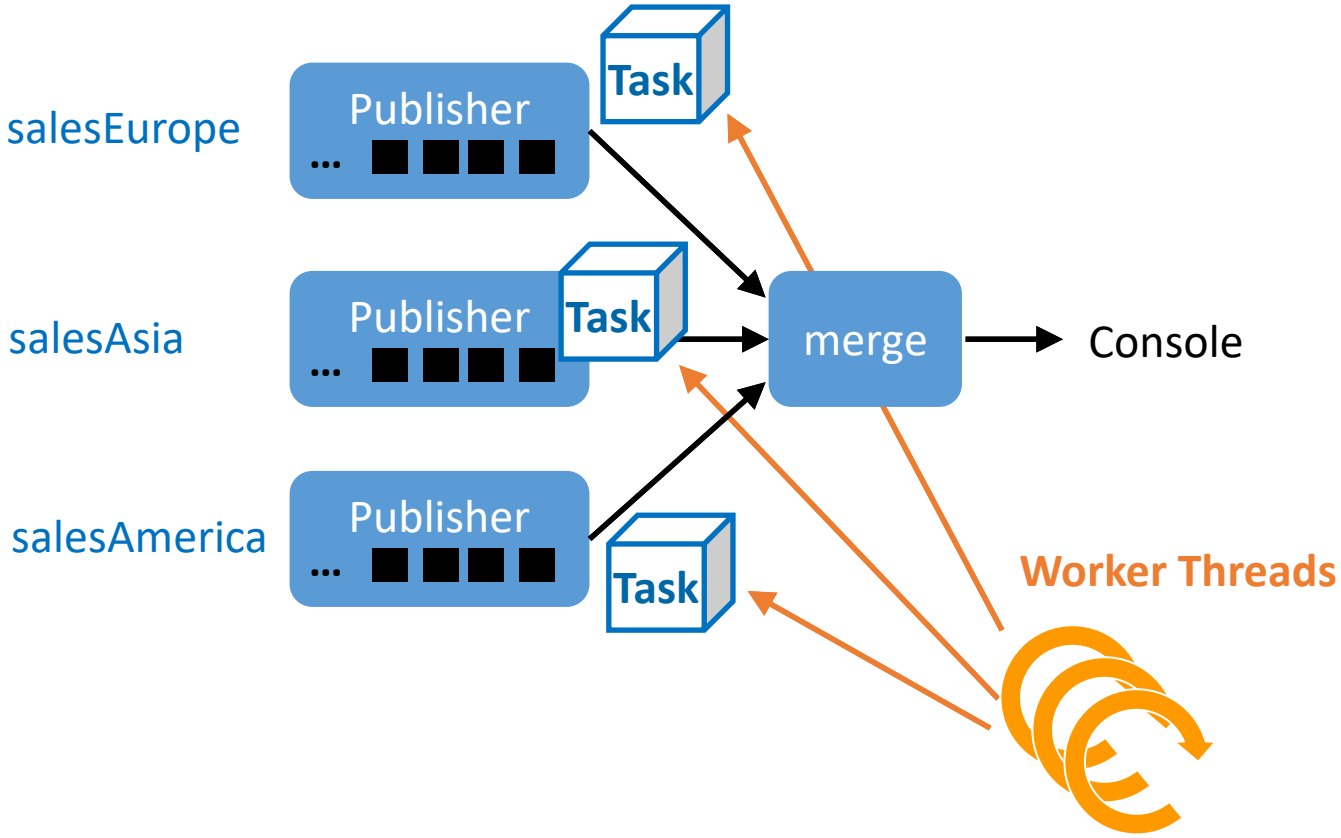
Parallele Verarbeitung

```
var sales1 = Flux.fromIterable(salesEurope)
  .publishOn(Schedulers.parallel());
var sales2 = Flux.fromIterable(salesAsia)
  .publishOn(Schedulers.parallel());
var sales3 = Flux.fromIterable(salesAmerica)
  .publishOn(Schedulers.parallel());

var combinedSales = Flux.merge(
  sales1, sales2, sales3
);

combinedSales.subscribe(System.out::println);
```

Wieso ist die Verarbeitung nach merge() wieder sequentiell?

Parallele Ausführung

```
publishOn(Schedulers.parallel())
```



Reactor Scheduler

- Default
  - Synchrone Ausführung (gleicher Thread wie Subscriber)
- `Schedulers.parallel()`
  - Parallele Ausführung in Thread Pool fixer Grösse
- `Schedulers.elastic()`
  - Parallele Ausführung in Thread Pool variabler Grösse
- `SwingScheduler.create()`
  - Erzeugt Scheduler für Swing Dispatching
  - Benötigt «reactor-extra» Dependency

Interaktion mit GUI

```
combinedSales.publishOn(SwingScheduler.create()).subscribe(
  entry -> {
    var newText = textBlock.getText()
        + entry.Article + " " + entry.Volume;
    textBlock.setText(newText);
  }
);
```

Ausführung auf GUI Thread

Mögliche Concurrency Fehler

Race Conditions

- Mit Seiteneffekten in Subscriber möglich
- Vermeiden oder wenn nötig, synchronisieren

Deadlocks

- Bei Warteabhängigkeiten in Subscriber
- Blockierende Aufrufe wie blockFirst(), blockLast() vermeiden

Vorgefertigte Flux (1)

`Flux.just("Value", ...)`	Liefert die Werte, dann Completed
`Flux.empty()`	Liefert sofort Completed
`Flux.never()`	Liefert nie etwas (auch nicht Completed)
`Flux.error(exception)`	Liefert sofort Error

Vorgefertigte Flux (2)

```
Flux.range(-10, 20)
```

Zahlen von -10 bis 9

```
Flux.<Long, Long>generate(
  () -> 0L,
  (state, sink) -> {
    sink.next(state);
    if(state == 98) {
      sink.complete()
    }
    return state + 2;
  }
)
```

Liefert 0, 2, 4, …, 98

```
source.take(10)
```

Nächste 10 Werte von source, danach Completed

Vorgefertigte Flux (3)

```
Flux.interval(
  Duration.ofMillis(250)
)
```

Liefert 0, 1, 2, .. im Zeitintervall von 250 ms

```
source.delayElements(
  Duration.ofSeconds(1)
)
```

Verzögert alle Werte von source einmal um 1 Sekunde

Events zu Flux

Event Quellen zu Flux umwandeln

```
Flux.<ActionEvent>create(sink -> {
  ActionListener listener = sink::next;
  button.addActionListener(listener);
  sink.onDispose(() -> {
    button.removeActionListener(listener);
  });
});
```

An- und Abmelden beim Event spezifizieren

Hot & Cold Flux

Cold = Passiv

- Nur aktiv mit Subscriber
- Startet mit jedem neuen Subscriber von vorne

```
var source = Flux.interval(…)

var source = Flux.create(…)
```

Hot = Aktiv

- Aktiv auch ohne Subscriber
- Neue Subscriber erhalten nur zukünftige Werte

```
// Erzeugt ConnectableFlux<T>
var hotFlux = source.publish();
// Hot Flux starten
var d = hotFlux.connect();
// Hot Flux stoppen
d.dispose();
```

Reactive Programming: Fazit

Vorteile

- Aktive Datenflüsse statt nur passive Streams
- Skalierbare Parallelität durch Wahl der Scheduler
- Durchgängig asynchron

Nachteile

- Zerstückelung komplexer Logiken in Handler
- Allfälliger Kontext muss durchgeschleust werden
- Komplizierte Aggregation (Observable statt Skalar)

Rückblick: Lernziele

- Reactive Programming verstehen
- Am Beispiel von Reactor anwenden können
- Nutzen und Einschränkungen beurteilen

Reactor Zusatzmaterial (bei Interesse)

- Reference Guid: https://projectreactor.io/docs/core/release/reference/
- JavaDoc: https://projectreactor.io/docs/core/release/api/
- Sonstiges: https://projectreactor.io/learn

Weitere Reactive Technologien

.NET TPL Dataflow

- https://msdn.microsoft.com/en-us/library/hh228603(v=vs.110).aspx
- Auch nicht Teil der Default TPL

ReactiveX RxJava (ehemals Netflix)

- Library: https://github.com/ReactiveX/RxJava
- Einführung: http://techblog.netflix.com/2013/02/rxjava-netflix-api.html
- http://www.introtorx.com/