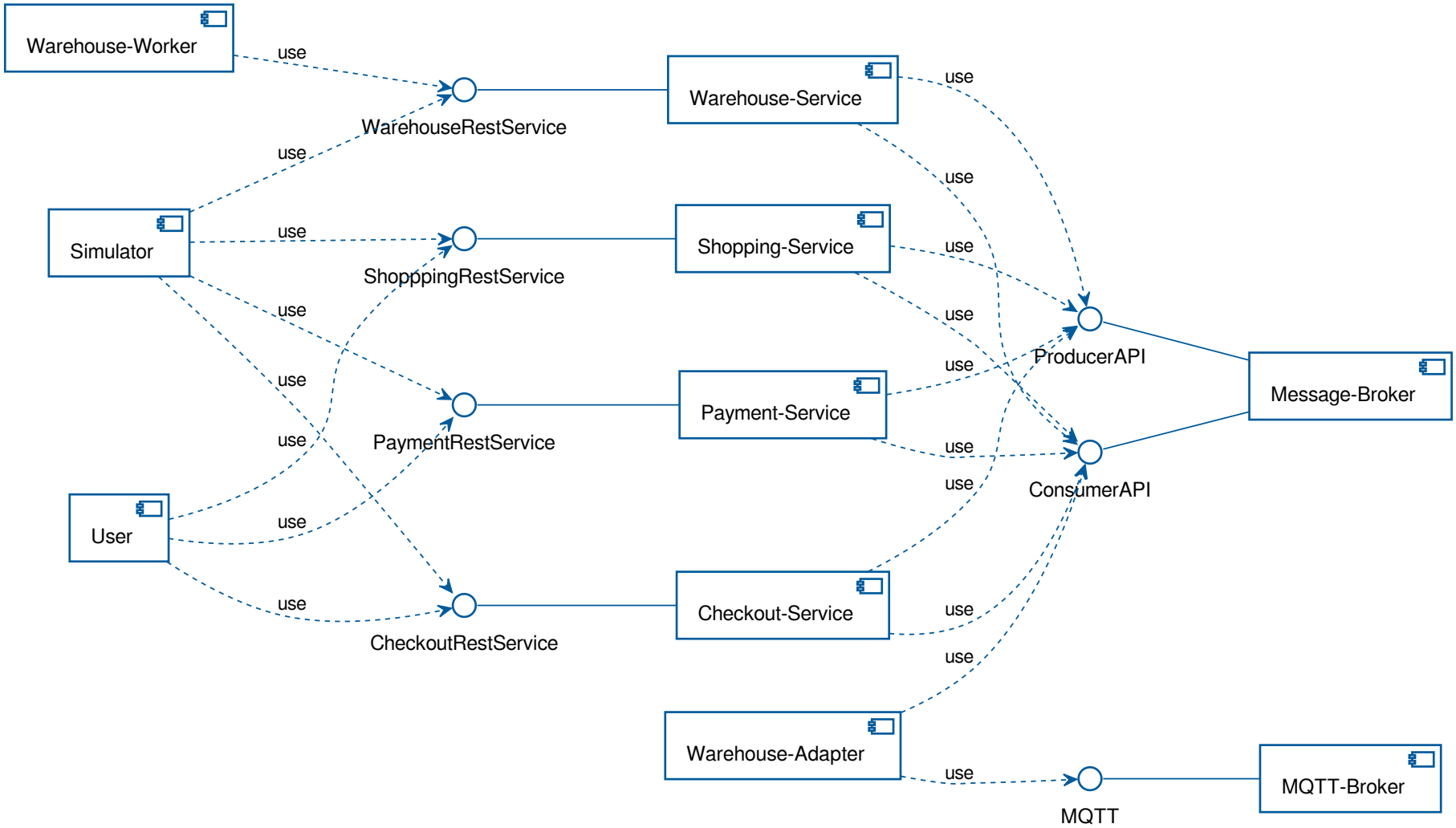
KAPITEL 3: STREAMING-APIS UND REACTIVE PROGRAMMING

LERNZIELE

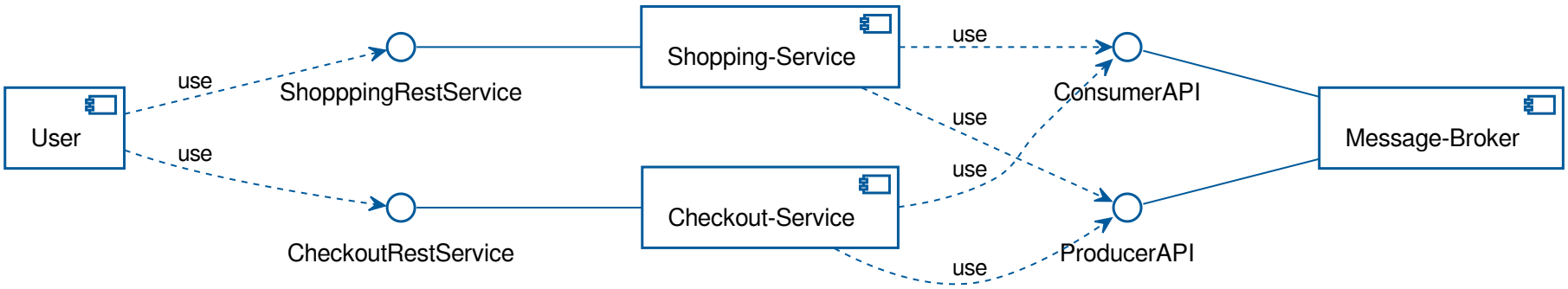
- Reactive Programming beschreiben und von klassischen imperativen Vorgehen abgrenzen
- Reactive Manifesto und die vier Kernaussagen kurz erklären
- Konzepte des Reactive Programming erklären
- *Marble Diagrams* verwenden, um Verarbeitungsvorgänge zu visualisieren
- Typische Operatoren des Reactive Programmings anwenden, um Problemstellungen zur Datenstromverarbeitung funktional zu lösen

3.1 MOTIVATION

Rückblick Shop-Beispiel



Die verschiedenen Dienste zur Bereitstellung von Geschäftslogik (Shopping, Checkout, Payment, Warehouse) sowie der Warehouse-Adapter, um die Sensoren zu integrieren, sind über Broker entkoppelt.



Die Kommunikation zwischen allen Diensten funktioniert über Nachrichten oder Ereignissen.

Aktuell sind Sequenzen von Anweisungen zur Verarbeitung platziert:

```
CartMessage message = ...
if(message instanceof ArticleAddedToCartMessage) {
    this.orders.addItemToOrderByCartRef(
            ((ArticleAddedToCartMessage) message).getId(),
            ((ArticleAddedToCartMessage) message).getArticle(),
            ((ArticleAddedToCartMessage) message).getName(),
            ((ArticleAddedToCartMessage) message).getPrice(),
            ((ArticleAddedToCartMessage) message).getCount());
} else if(message instanceof DeleteArticleFromCartMessage) {
    this.orders.deleteItemFromOrderByCartRef(
            ((DeleteArticleFromCartMessage) message).getId(),
            ((DeleteArticleFromCartMessage) message).getArticle());
} else if(message instanceof CreatedCartMessage) {
    this.orders.createOrderWithCartRef(
        ((CreatedCartMessage) message).getId());
}
```

Hier je nach eingegangener Nachricht die entsprechenden Bedingungen zur Verarbeitung des Ereignisses ausgewertet, z.B. wenn ein Elemente vom Cart gelöscht wird.

Fragestellung: Wie kann dieser imperative Ansatz zu einem ereignisbasierten Ansatz, bzw. funktionalen Ansatz weiterentwickelt werden?

Mit dem Fokus auf Reactive Programming und Streaming-APIs soll ein Blick in die Realisierungsmöglichkeiten für die Integration von Diensten wie auch die Realisierung von Verarbeitungslogik auf Basis von Ereignissen bzw. Nachrichten geworfen werden.

3.2 BASICS

Imperative vs. Functional Programming

Java ist eine imperative Programmiersprache.

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> output = new ArrayList<>();
for (Integer x : input) {
  if (x % 2 == 0) {
    output.add(x);
  }
}
```

Normalerweise besteht ein Java-Programm aus einer Folge von Anweisungen. Jede dieser Anweisungen wird in der gleichen Reihenfolge ausgeführt, in der Sie sie schreiben, und die Ausführung führt zu Änderungen im Zustand des Programms.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

Alternative: Funktionale Programmierung

Bei der funktionalen Programmierung ergibt sich das Ergebnis des Programms aus der Auswertung mathematischer Funktionen, ohne dass der interne Programmzustand verändert wird.

```
var output = input.where( x -> x % 2 == 0);
```

- Bei jeder Funktion `$f(x)$` hängt das Ergebnis der Funktion von den Argumenten ab, die der Funktion übergeben werden.
- Jedes Mal, wenn `$f(x)$` mit dem gleichen Parameter x aufgerufen wird, erhält man immer das gleiche Ergebnis.
- Einfacher ausgedrückt: In der funktionalen Programmierung sind die Bausteine, mit denen Sie das Programm aufbauen, keine Objekte, sondern Funktionen und Prozeduren.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

Eine funktionale Sprache zeichnet sich durch die folgenden Merkmale aus:

- **Funktionen höherer Ordnung** sind Funktionen, die andere Funktionen als Argumente annehmen.

```
public static void doSomething(Runnable r) {
    r.run();
}
public static void main(String[] args) {
    doSomething(() -> System.out.println("Hello"));
}
```

doSomething ist eine Funktion, die eine andere Funktion als Argument annimmt, auch wenn dies in Java mit einem Objekt in Verbindung steht.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

- **Unveränderliche Daten**; anstatt bestehende Werte zu ändern, arbeiten funktionale Sprachen (oft) mit Kopien der ursprünglichen Werte, um sie zu erhalten. In Java sind z.B. primitive Typen bereits unveränderlich, ein Objekt jedoch nicht, so dass seine Implementierung nicht zulassen darf, dass der Zustand des Objekts nach der Erstellung geändert wird, vgl. Clone, Immutable, usw.
- **Concurrency**: Gleichzeitigkeit wird unterstützt und ist sicherer zu zu implementieren, auch dank der standardmäßigen Unveränderlichkeit.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

- **Referentielle Transparenz**: Berechnungen können jederzeit durchgeführt werden und liefern immer das gleiche Ergebnis. Mit referentiell transparenten Methoden / Funktionen, wie z.B. statische Methoden in Java, gibt es keinen Bezug zu außerhalb verwalteten Zuständen.

```
static int plusOne(int x) {
  return x + 1;
}
```

- **Lazy Evaluation**: Werte können nur bei Bedarf (Layz) berechnet werden, da Funktionen jederzeit ausgewertet werden können und immer das gleiche Ergebnis liefern (diese Funktionen sind nicht vom internen Zustand des Programms abhängig).

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

LAMBDA-AUSDRÜCKE

Lambda-Ausdrücke sind anonyme Funktionen

Der Lambda-Operator wird z.B. in Java durch ein nach rechts weisendes Pfeilsymbol (->) gekennzeichnet. Eingaben werden links vom Operator platziert, der Funktionskörper rechts.

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);
input.forEach(i -> System.out.println(i));
```

In Java können Lambda-Ausdrücke verwendet werden, um anonyme innere Klassen zu ersetzen, die eine Schnittstelle mit nur einer Methode implementieren.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

Beispiel Button in JavaFX: Anonyme innere Klasse

```
Button button = ...
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Hello, World!");
    }
});
```

Eine anonyme innere Klasse beschreibt die Funktionalität, die auszuführen ist, wenn der Button geklickt wird.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

Beispiel Button in JavaFX: Lambda-Ausdruck

```
Button button = ...
button.setOnAction((event) -> System.out.println("Hello, World!"));
```

Die selbe Funktionalität lässt sich ebenfalls mit einem Lambda-Ausdruck hinterlegen, welche beim Klick des Buttons genutzt wird.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

In Java kommen hierfür sogenannte funktionale Schnittstellen zum Einsatz.

```
public interface Calculate {
    int doIt(int a, int b);
}
```

```
Calculate foo = (a, b) -> a + b;
System.out.println(foo.doIt(2, 3));
```

Ein **Functional Interface** ist ein Interface mit genau einer Methode, etwa das Interface `Comparator<T>`, welches die Methode *compare* fordert. Mit Java 8 werden Functional Interfaces als eigener Datentyp eingeführt, um Lambda Ausdrücke als Instanzen von Functional Interfaces (=Objekte, die ein Functional Interface implementieren) definieren zu können.

Eta-Konvertierung

Häufig findet man Situationen, in denen mittels Lambda-Ausdrücken eine Methode an einer Objektinstanz in Java mit den Parametern des Lambda-Ausdrucks als Argumente ausgeführt werden soll. Beispiel:

```
List<String> list = Arrays.asList("Apfel", "Kiwi", "Birne");
list.forEach((str) -> System.out.println(str));
```

Das Beispiel gibt den Parameter mittels eines Aufrufs von `System.out.println` als Argument weiter.

Alternativ kann mit der Eta-Konvertierung die Objektinstanz und der zu tätigende Methodenaufruf in einer Kurzform übergeben wird.

```
List<String> list = Arrays.asList("Apfel", "Kiwi", "Birne");
list.forEach(String.out::println);
```

Dabei bezeichnet `String.out` die Referenz auf die `PrintStream`-Instanz und `println` nennt die Methode die für jedes Element in `forEach` an `String.out` aufgerufen werden soll.

Damit Methoden über die η-Konvertierung angegeben werden können, muss nur die Signatur und der Rückgabedatentyp passen.

Viele Sprachen bieten die Möglicheit Lambda-Ausdrücke zu formulieren

- Python: `f = lambda x, y : x + y`
- JavaScript: `var f = (x, y) => x + y`
- C#: `Func<int, int, int> square = (x, y) => x + y;`

REACTIVE PROGRAMMING

Was ist reaktive Programmierung?

- Grundsätzlich befasst sich die reaktive Programmierung mit Ereignisströmen: einer Abfolge von Ereignissen, die im Laufe der Zeit eintreten
- Wann immer ein Ereignis eintritt, wird darauf reagiert, indem etwas getan wird
- Auf Ereignisse könnte imperativ reagiert werden, z.B. indem `for`-Schleifen verwendent werden
- ... in der funktionalen Programmierung werden die Transformationen jedoch über map(), filter() und andere Rx-Operatoren durchgeführt.

Quelle: https://medium.com/@jshvarts/read-marble-diagrams-like-a-pro-3d72934d3ef5#8a02

Reactive programming takes functional programming a little bit further, by adding the concept of data flows and propagation of data changes.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

Beispiel einer Formel:

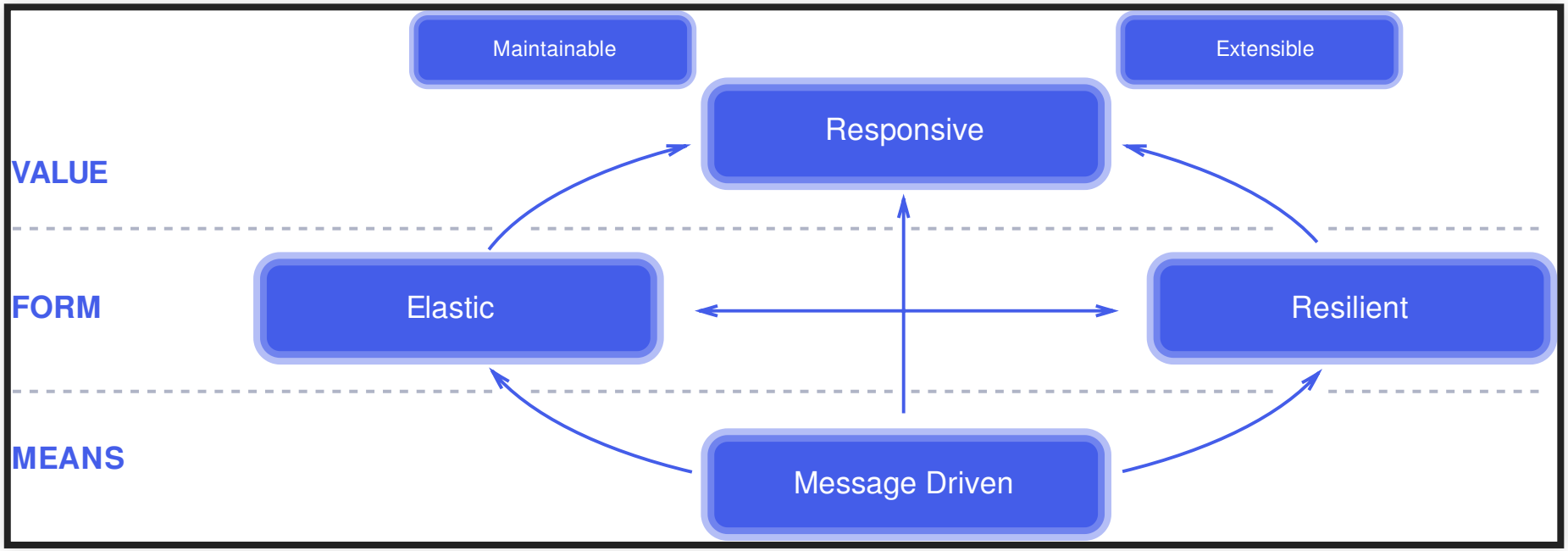
$$x = y + z$$

In der imperativen Programmierung, würde das Ergebnis zum Zeitpunkt der Ausführung der Anweisung die Summe aus dem aktuell in y und z gespeicherten Wert in x speichern.

In der reaktiven Programmierung würde sich x ändern, wann immer sich y oder z ändern. Die Implementierung reagiert basierend auf Ereignissen.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

REACTIVE MANIFESTO



Quelle: https://www.reactivemanifesto.org

*Reaktive Systeme sind **flexibler**, **lose gekoppelt** und **skalierbar**. Dadurch sind sie einfacher zu entwickeln und leichter zu ändern. Sie sind fehlertoleranter, und wenn ein Fehler auftritt, begegnen sie diesem entsprechend. Reaktive Systeme sind reaktionsschneller und geben ein effektives interaktives Feedback.*

Quelle: https://www.reactivemanifesto.org

Reaktionsfähig

Das System reagiert, wenn möglich, zeitnah

- Reaktionsfähigkeit ist der Eckpfeiler für Usability und Utility
- Reaktionsfähigkeit bedeutet darüber hinaus, dass Probleme schnell erkannt und effektiv behandelt werden können
- Reaktionsfähige Systeme konzentrieren sich auf schnelle und konsistente Reaktionszeiten und legen zuverlässige Obergrenzen fest, so dass sie eine konsistente Dienstqualität bieten
- Konsistentes Verhalten vereinfacht die Fehlerbehandlung, stärkt das Vertrauen der Endbenutzer und ermutigt zu weiterer Interaktion

Quelle: https://www.reactivemanifesto.org

Widerstandsfähig

Das System bleibt auch im Falle eines Ausfalls reaktionsfähig

- Die Ausfallsicherheit wird durch Replikation, Eingrenzung, Isolierung und Delegation erreicht.
- Ausfälle werden innerhalb jeder Komponente eingegrenzt, wodurch die Komponenten voneinander isoliert werden und somit sichergestellt wird, dass Teile des Systems ausfallen und wiederhergestellt werden können, ohne das System als Ganzes zu gefährden.
- Die Wiederherstellung jeder Komponente wird an eine andere (externe) Komponente delegiert, und die Hochverfügbarkeit wird erforderlichenfalls durch Replikation sichergestellt.
- Der Client einer Komponente ist nicht mit der Behandlung ihrer Ausfälle belastet.

Quelle: https://www.reactivemanifesto.org

Elastisch

Das System bleibt auch bei wechselnder Arbeitslast reaktionsfähig.

- Reaktive Systeme können auf Änderungen in der Inputrate reagieren, indem sie die zur Behandlung dieser Inputs zugewiesenen Ressourcen erhöhen oder verringern
- Dies setzt Entwürfe voraus, die keine Konfliktpunkte oder zentralen Engpässe aufweisen, was dazu führt, dass die Komponenten aufgeteilt oder repliziert und die Eingaben auf sie verteilt werden können.
- Reaktive Systeme unterstützen sowohl prädiktive als auch reaktive Skalierungsalgorithmen, indem sie relevante Live-Leistungsmessungen bereitstellen.
- Sie erreichen Elastizität auf kostengünstige Weise auf Standard-Hardware- und Software-Plattformen.

Quelle: https://www.reactivemanifesto.org

Message-Driven

Reaktive Systeme stützen sich auf asynchrones Message-Passing, um eine Grenze zwischen den Komponenten zu schaffen, die eine lose Kopplung, Isolierung und Standorttransparenz gewährleistet.

- Diese Grenze bietet die Möglichkeit, Fehler als Nachrichten zu delegieren.
- Der Einsatz von Message-Passing ermöglicht Lastmanagement, Elastizität und Flusskontrolle, indem die Nachrichtenwarteschlangen im System geformt und überwacht werden und bei Bedarf Backpressure ausgeübt wird

Quelle: https://www.reactivemanifesto.org

- Messaging als Kommunikationsmittel macht es möglich, dass das Fehlermanagement in einem Cluster oder innerhalb eines einzelnen Hosts mit denselben Konstrukten und derselben Semantik arbeitet
- Nicht blockierende Kommunikation ermöglicht es den Empfängern, nur dann Ressourcen zu verbrauchen, wenn sie aktiv sind

Quelle: https://www.reactivemanifesto.org

Vorteile

- eine bessere Kontrolle über die Reaktionszeiten bei der Verarbeitung von Ereignissen
- Konsistenz im Softwaredesign für Echtzeitsysteme, um Entwicklungs- und Wartungskosten und -aufwand zu reduzieren
- Lastausgleich und Ausfallsicherheit, um die Qualität der Erfahrung zu verbessern, und
- Verdeutlichung des Konzepts eines Datenstroms oder Ereignisflusses, wodurch die Verwaltung von Rechenelementen und Verarbeitungsressourcen insgesamt verbessert werden kann, indem sie "anschaulicher" gemacht werden

Quelle: https://www.techtarget.com/searchapparchitecture/definition/reactive-programming

Herausforderungen

- Hinzufügen von Beobachterprozessen zu aktueller Software kann schwierig oder unmöglich sein, abhängig von der Verfügbarkeit von Quellcode und den Programmierkenntnissen der Mitarbeiter
- Reaktives Design bedeutet für die Entwickler einen Bewusstseinswandel
- es gibt eine Lernphase, bei der mehr Validierung und Überwachung von Design und Implementierungen erforderlich sein kann
- Reaktive Systeme können durch eine übermäßige Anzahl von Prozessen, die mit dem Datenstrom verbunden sind, zu Verzögerungen führen

Quelle: https://www.techtarget.com/searchapparchitecture/definition/reactive-programming

REACTIVE STREAMS

- Reactive Streams ist ein Standard für asynchrone Stream-Verarbeitung mit nicht-blockierendem Backpressure
- Diese Spezifikation ist im Reactive Manifesto definiert, und es gibt verschiedene Implementierungen davon, zum Beispiel RxJava oder Akka-Streams

Um einen Flow zu erstellen, können drei Hauptabstraktionen verwendet und sie zu einer asynchronen Verarbeitungslogik zusammengesetzt werden: Publisher, Subscriber und Processor

Publisher

Jeder Flow muss Ereignisse verarbeiten, die ihm von einer Publisher-Instanz veröffentlicht werden; der Publisher hat eine Methode - `subscribe()`.

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> subscriber);
}
```

Wenn einer der Abonnenten die von ihm veröffentlichten Ereignisse empfangen möchte, muss er sich bei dem betreffenden Publisher anmelden.

Subscriber

Der Empfänger von Nachrichten muss die Subscriber-Schnittstelle implementieren. Normalerweise ist dies das Ende für jede Flow-Verarbeitung, da die Instanz keine weiteren Nachrichten sendet.

```
public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T value);
    void onError(Throwable t);
    void onComplete();
}
```

Der Subscriber kann als eine Senke beschrieben werden. Diese hat vier Methoden, die überschrieben werden müssen - `onSubscribe()`, `onNext()`, `onError()` und `onComplete()`.

Subscription

Eine Instanz der Subscription wird als Argument übergeben.

```
public interface Subscription {
    void request(long n);
    void cancel();
}
```

Es handelt sich um eine Klasse, die zur Steuerung des Nachrichtenflusses zwischen Subscriber und Publisher verwendet wird.

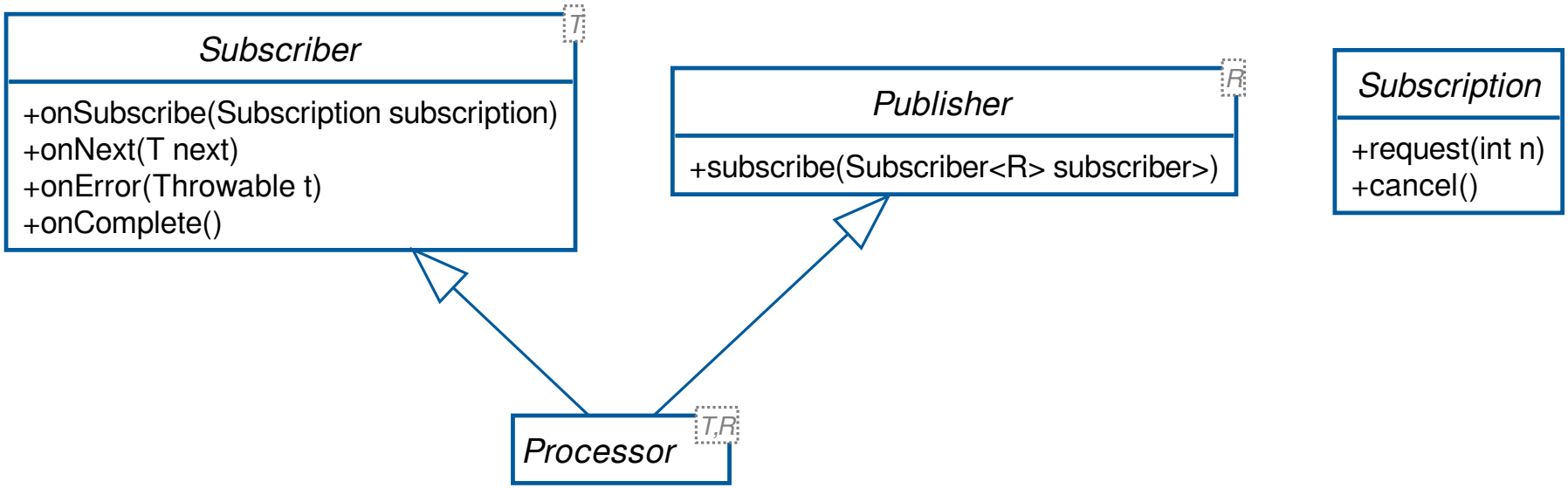
Processor

Wenn eine eingehende Nachricht umgewandelt und an den nächsten Subscriber weitergeleitet werden soll, muss die Schnittstelle Processor implementiert werden.

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Dieser fungiert sowohl als Subscriber, weil er Nachrichten empfängt, als auch als Publisher, weil er diese Nachrichten verarbeitet und zur weiteren Verarbeitung weiterleitet.

Die vier Schnittstellen sind im folgenden als Klassendiagram dargestellt und stellen die Basis für Reactive Streams dar.



Ein Transformationsbeispiel kann im folgenden Paket betrachtet werden.

Nutzen Sie das bereitgestellte Projekt-Archiv reactive-streams.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Beispiel-Verwendung von Reactive Streams

```
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
TransformProcessor<String, Integer> transformProcessor =
    new TransformProcessor<>(Integer::parseInt);
EndSubscriber<Integer> subscriber = new EndSubscriber<>();
publisher.subscribe(transformProcessor);
transformProcessor.subscribe(subscriber);
List.of("1", "21", "2", "4", "3", "42").forEach(publisher::submit);
publisher.close();
```

- Erstellt einen Publisher
- Erstellt einen Processor (welcher Subscriber und Publisher ist)
- Erstellt einen Subscriber
- Verknüpft Publisher, Processor und Subscriber: SubmissionPublisher > TransformerPublisher > EndSubscriber
- Als Beispiel werden Einträge eines Arrays an den Publisher übergeben

BIBLIOTHEKEN

Es gibt einige Bibliotheken um Reactive Programming in verschiedenen Sprachen zu ermöglichen

- ReactiveX
- WebFlux

Was ist ReactiveX

> *ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.*

Es handelt sich um eine Bibliothek, die funktionale reaktive Programmierung in vielen Sprachen implementiert. Sie verwendet "Observables", um asynchrone Datenströme darzustellen, und abstrahiert alle Details im Zusammenhang mit Threading, Gleichzeitigkeit und Synchronisierung.

Quelle: A. Maglie, Reactive Java Programming, DOI 10.1007/978-1-4842-1428-2_1

Implementierungen von ReactiveX sind:

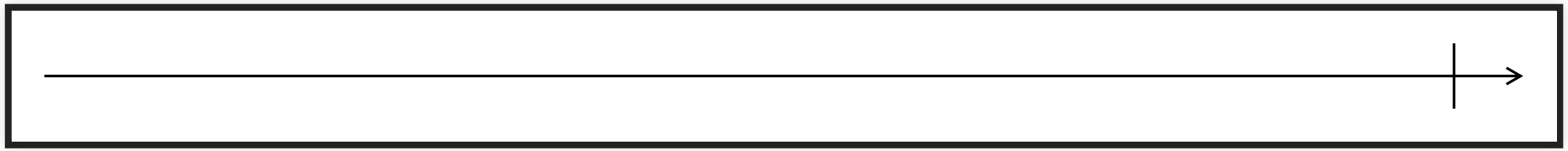
- RxJava
- RxJs
- Rx.NET
- RxScala
- RxClojure
- RxSwift

Was ist WebFlux

Eine Alternative ist WebFlux, welche im Spring-Kontext zu finden ist. WebFlux kann direkt in Spring-Anwendungen verwendet werden, zum Beispiel die Rückgabewerte / -typen der Rest-Controller könnte direkt auf Ereignisketten basieren.

MARBLE DIAGRAMS

Für die Darstellung und Erklärung von Rx-Operatoren werden häufig Marble-Diagramme verwendet.

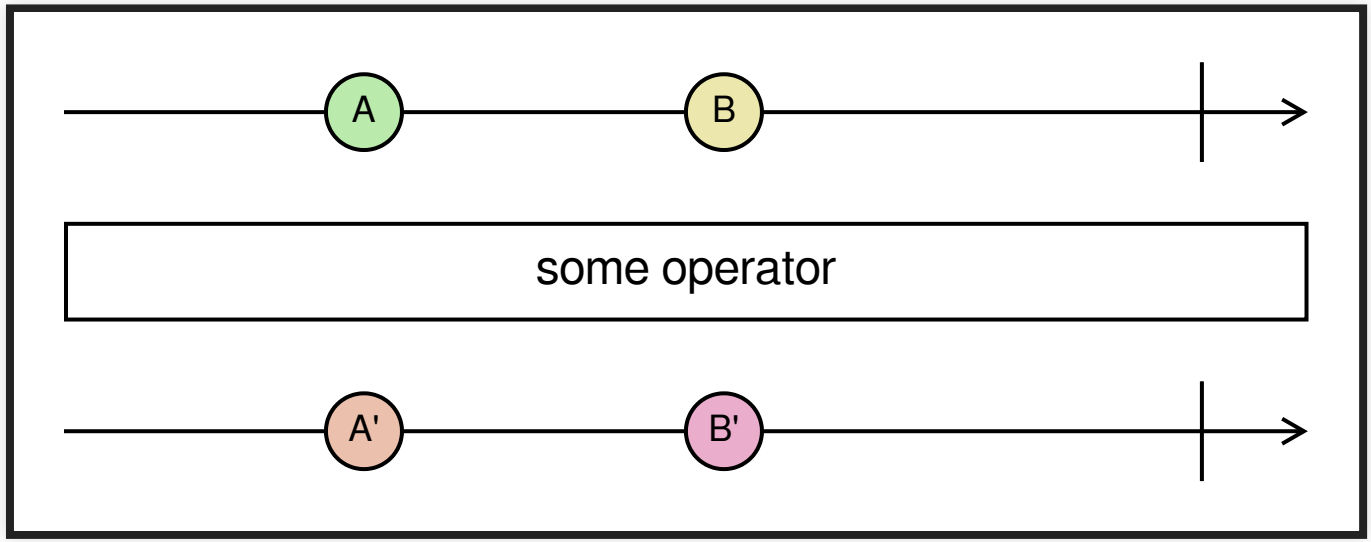


In der einfachsten Darstellung wird eine Zeitleiste verwendet, in der nichst passiert.

Auf dieser Zeitleiste können Ereignisse platziert werden

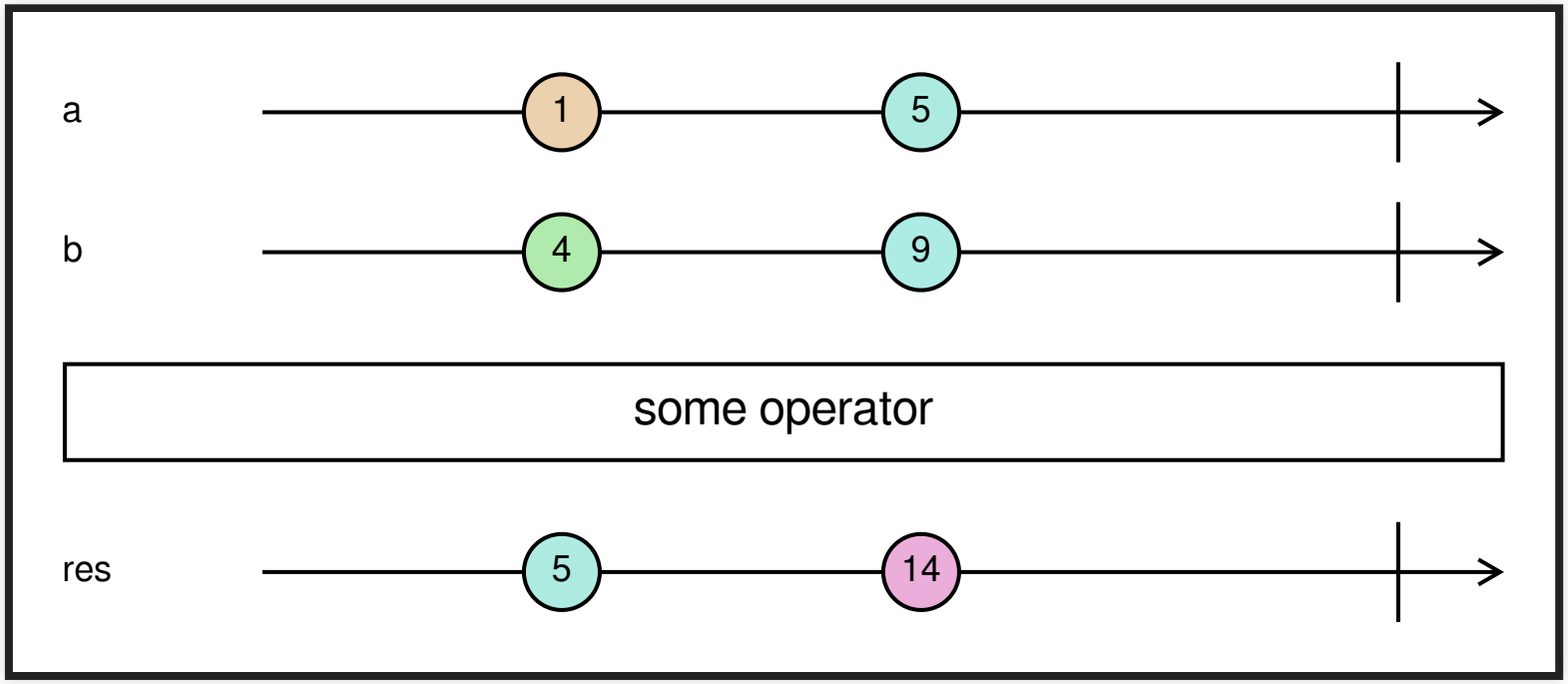
Dabei können diese beliebig beschrift werden und deren Auftreten ist sortiert nach dem Eintrittszeitpunkt. Unterschiedliche Abstände deuten unterschiedliche Zeitabstände an.

Operatoren werden in weißen Rechtecken dargestellt



Über dem Rechteck befinden sich Inputs, unter dem Rechteck die Outputs.

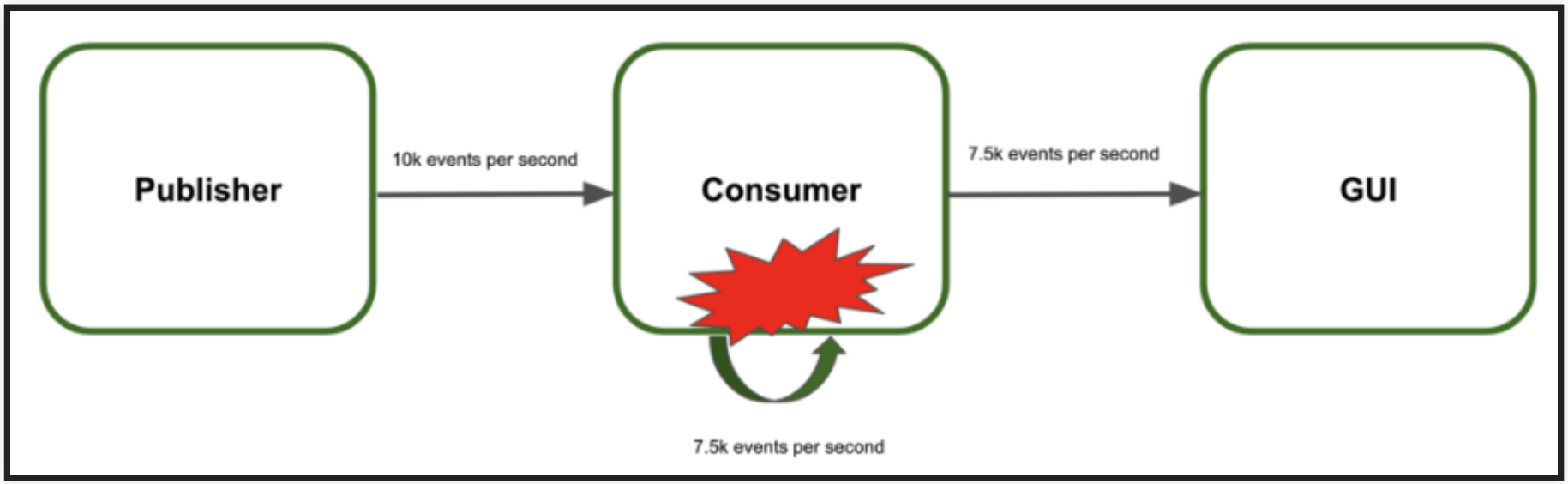
Mehrere Inputs möglich



Mehrere eingehende Datenreihen werden über unabhängige Datenreihen dargestellt, zum besseren Verständnis können diese benannt werden.

BACKPRESSURE

In Reactive Streams definiert Backpressure, wie die Übertragung von Stream-Elementen reguliert werden kann. Mit anderen Worten, es wird gesteuert, wie viele Elemente der Empfänger konsumieren kann.



Quelle: https://www.baeldung.com/spring-webflux-backpressure

Mittels Backpressure kann gesteuert werden, wieviel ein Subscriber verarbeiten möchte

```
Flux.range(1, 50).subscribe(
    System.out::println,
    err -> System.out.println(err),
    () -> System.out.println("All 50 items have been successfully processed!
    subscription -> {
        for (int i = 0; i < 5; i++) {
            System.out.println("Requesting the next 10 elements!!!");
            subscription.request(10);
        }
    }
);
```

Hierfür wird mit der Subscription über die request-Methode gesteuert, wieviele Ereignisse als nächstes verarbeitet werden sollen.

Quelle: https://www.baeldung.com/spring-webflux-backpressure

3.3 STREAM API

Sprachen bieten Konzepte zur Verarbeitung von Datenstrukturen, welche mit den Ansätzen im Reactive Programming vergleichbar sind, jedoch nicht gleichgesetzt werden können.

Zum Beispiel könnte in Java eine Collection (z.B. eine Personenliste) dazu verwendet werden, Teile der Collection mit bestimmten Eigenschaften herauszufiltern (z.B. alle weiblichen Personen) und das Ergebnis zu verändern bzw. als Eingabe für weiterverarbeitende Methoden zu verwenden (z.B. schicke eine E-Mail an alle Personen der Liste, die älter als 18 sind).

Das Stream-Konzept in Java erlaubt eine Art Datastream-Processing, auf dem Operationen auf alle Elemente einer Datenstruktur angewendet werden. Dabei werden drei Typen von Operationen auf Streams unterschieden:

Daten → Stream → Operation_1 → ... → Operation_n → Ergebnis

Daten → Stream → Operation$_1$ → ... → Operation$_n$ → Ergebnis

Daten und **Stream** entsprechen den Create-Operation

Operation$_1$, ..., Operation$_n$ entsprechen den Intermediate-Operation

Ereignis entspricht der Terminal-Operation

Beispiel

```
List<Person> personen = …;
List<Person> erwachsene = personen.stream() // create
  .filter(p -> p.getAlter() >= 18) // intermediate
  .collect(Collectors.toList());   // terminal
```

- Mit der `stream`-Methode als Create-Operation wird ein Stream-Objekt erzeugt
- Die `filter`-Methode als Intermediate-Operation reduziert die Menge der Personen-Objekte auf die Erwachsenen (Alter >= 18!) ...
- ... während die vorgegebene toList-Methode als Terminal-Operation den resultierenden gefilterten Stream wieder in eine Liste umwandelt.

CREATE-OPERATIONEN

Umwandlung Array → Stream:

```
Person[] personen = …;
Stream<Person> stream = Arrays.stream(personen);
```

Umwandlung Collection → Stream:

```
Collection<Person> personen = new ArrayList<Person>();
Stream<Person> stream = personen.stream();
```

INTERMEDIATE-OPERATIONEN

- `Stream<T> filter(Predicate<T> p):`
  Liefert einen Stream von T-Objekten, die p erfüllen
- `Stream<R> map(Function<T, R> f):`
  Liefert einen Stream von R-Objekten, die durch f aus dem Ausgangs-Stream gebildet werden
- `Stream<T> sorted(Comparator<T> c):`
  Liefert Stream mit Sortierung gemäß c

TERMINAL-OPERATIONEN

- `void forEach(Consumer<T> action):`
  Führt action auf jedem Element des Streams aus
- `Object[] toArray():`
  Liefert ein Array mit den Objekten des Streams
- `T reduce(T identity, BinaryOperator<T> op):`
  Reduktionsoperation, siehe Beispiel Folgeseite

- `T min(Comparator<T> c):`
  Liefert das Minimum des Streams gemäß c
- `T max(Comparator<T> c):`
  Liefert das Maximum des Streams gemäß c
- `long count():`
  Liefert die Anzahl Elemente des Streams
- `T findFirst():`
  Liefert das erste Element des Streams

BEISPIEL: PERSONEN-STREAM

```
public static void main(String[] args) {
  List<Person> personen = Arrays.asList(
    new Person("Anna", 19), new Person("Hugo", 24),
    new Person("Codie", 18), new Person("Susi", 14));
  int summeAlter = personen.stream()
    .mapToInt(p -> p.getAlter())
    .reduce(0, (x, y) -> x + y);
  double durchschnittsAlter = (double) summeAlter / personen.size();
  System.out.printf("Anzahl Personen: %d, Durchschnittsalter: %f\n",
    personen.size(), durchschnittsAlter);
}
```

- Die Personenliste wird zunächst in einen Stream umgewandelt.
- Die Methode mapToInt (s. Java API) wandelt den Person-Stream in einen int-Stream um, indem von jeder Person das Alter verwendet wird.
- Dann führt reduce eine Summation der Alterswerte durch, beginnend bei 0.
- Abschließend wird das Durchschnittsalter berechnet und ausgegeben.

BEISPIEL: DATENTYP-TRANSFORMATION VON LISTEN

```
class SomeEntity {
    private UUID id;
    public UUID getId() {
        return id;
    }
}
class SomeEntityDto {
    private UUID id;
    public SomeEntityDto(UUID id) {
        this.id = id;
    }
    public static SomeEntityDto fromSomeEntity(SomeEntity entity) {
        return new SomeEntityDto(entity.getId());
    }
}
```

```
List<SomeEntity> entities = loadFromDb();
List<SomeEntityDto> result = entities.stream()
    .map(SomeEntityDto::fromSomeEntity)
    .collect(Collectors.toList());
```

ÜBUNG: FILTER MITARBEITER

Gegeben ist eine Liste von Mitarbeitern

```
List<Employee> employees = Arrays.asList(
    new Employee("Alice", 25, "Marketing", 55000.0),
    new Employee("Bob", 30, "Sales", 60000.0),
    new Employee("Charlie", 35, "Engineering", 75000.0),
    new Employee("Dave", 40, "Engineering", 80000.0),
    new Employee("Emily", 28, "Marketing", 45000.0),
    new Employee("Frank", 33, "Sales", 65000.0)
);
```

Verwenden Sie die Stream-API, um die Standorte aller Mitarbeiter die älter als 30 und weniger als 50.000 verdienen zu finden.

```
Set<String> departments = employees.stream()
    .filter(e -> e.getAge() < 30 && e.getSalary() < 50000)
    .map(Employee::getDepartment)
    .collect(Collectors.toSet());

System.out.println(departments);
```

ABGRENZUNG VON REACTIVE PROGRAMMING

Reactive Programming bietet vergleichbare Ansätze wie bei der Stream-API in Java

- Stream-API ist blockierend
- Verarbeitung erfolgt synchron
- Ausführung ist auf **eine** Verarbeitungskette beschränkt, keine Aufteilung
- *Stream-API basiert nicht auf Reactive Streams*
- Hier ist kein Observerieren von Änderungen im Vordergrund (bzw. Pub/Sub), Daten werden an die nachfolgende Verarbeitung weiter "gepusht"

3.4 WEBFLUX

Spring WebFlux ist ein Framework für die Entwicklung reaktiver und nicht blockierender Webanwendungen. Es unterstützt reaktive Streams und arbeitet gut mit Java Streams und dem Paradigma der funktionalen Java-Programmierung zusammen.

WebFlux-Anwendungen verwenden Reactive Streams, um die nicht blockierende Kommunikation zwischen einem Publisher und einem Subscriber zu erleichtern.

Quelle: https://www.baeldung.com/java-reactor-flux-vs-mono

Mono

Mono ist ein spezieller Typ von Publisher. Ein Mono-Objekt repräsentiert einen einzelnen oder leeren Wert. Das bedeutet, dass es höchstens einen Wert für den `onNext`-Aufruf ausgeben kann und dann mit dem `onComplete`-Signal beendet wird. Im Falle eines Fehlers gibt es nur ein einziges `onError`-Signal aus.

```
Mono<String> colorPublisher = Mono.just("RED");
```

```
Mono<String> colorPublisher = Mono.empty();
```

Quelle: https://www.baeldung.com/java-reactor-flux-vs-mono

Flux

Flux ist ein Publisher, der 0 bis N asynchrone Sequenzwerte darstellt. Das bedeutet, dass er 0 bis viele Werte ausgeben kann, möglicherweise unendliche Werte für `onNext`-Aufforderungen, und dann entweder mit einem Abschluss- oder einem Fehlersignal endet.

```
Flux<String> colorsPublisher = Flux.just("RED", "BLUE", "ORANGE");
```

Quelle: https://www.baeldung.com/java-reactor-flux-vs-mono

Terminal-Operationen

Damit auf Ereignisse gelauscht werden kann sind Terminal-Operationen notwendig.
Zum Beispiel hierfür steht `subscribe` zur Verfügung.

```
Flux<String> colors = Flux.just("RED", "BLUE", "ORANGE");
colors.subscribe(el -> System.out.println(el));
```

```
Flux<String> colors = Flux.just("RED", "BLUE", "ORANGE");
colors.subscribe(System.out::println);
```

Alternative Terminal-Operation: Ergebnis zurückgeben

```
String test = Mono.just("Test").block();
```

```
List<String> list = Flux.just("RED", "BLUE", "ORANGE").collectList().block()
```

Integration in Spring

```
@RestController
@RequestMapping("/test")
public class TestController {
    @GetMapping
    public Flux<Integer> getAllEmployees() {
        return Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4});
    }
}
```

Quelle: https://www.baeldung.com/spring-webflux

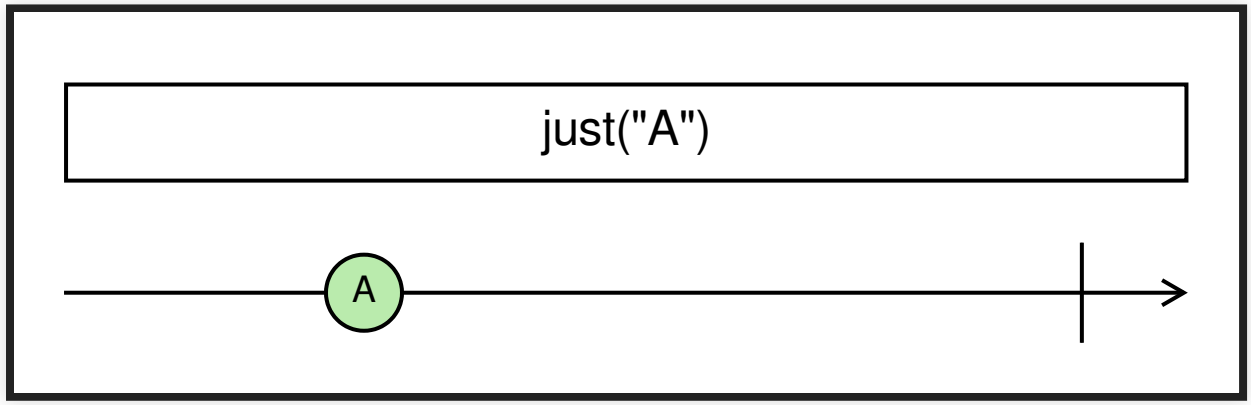
TYPISCHE OPERATOREN

Rx kennt verschiedene Operatoren die zur Verarbeitung und Manipulation von Datenströmen genutzt werden kann

- Creation: Erzeugt neue Publisher, z.B. über ein Array
- Filtering: Verändert die Anzahl auftretender Ereignisse
- Math: Ermittelt Ergebnisse durch Betrachtung von Datenströmen
- Combination: Kombiniert verschiedene Datenströme
- Transformation: Verändert aufgetretene Ereignisse in Datenströmen

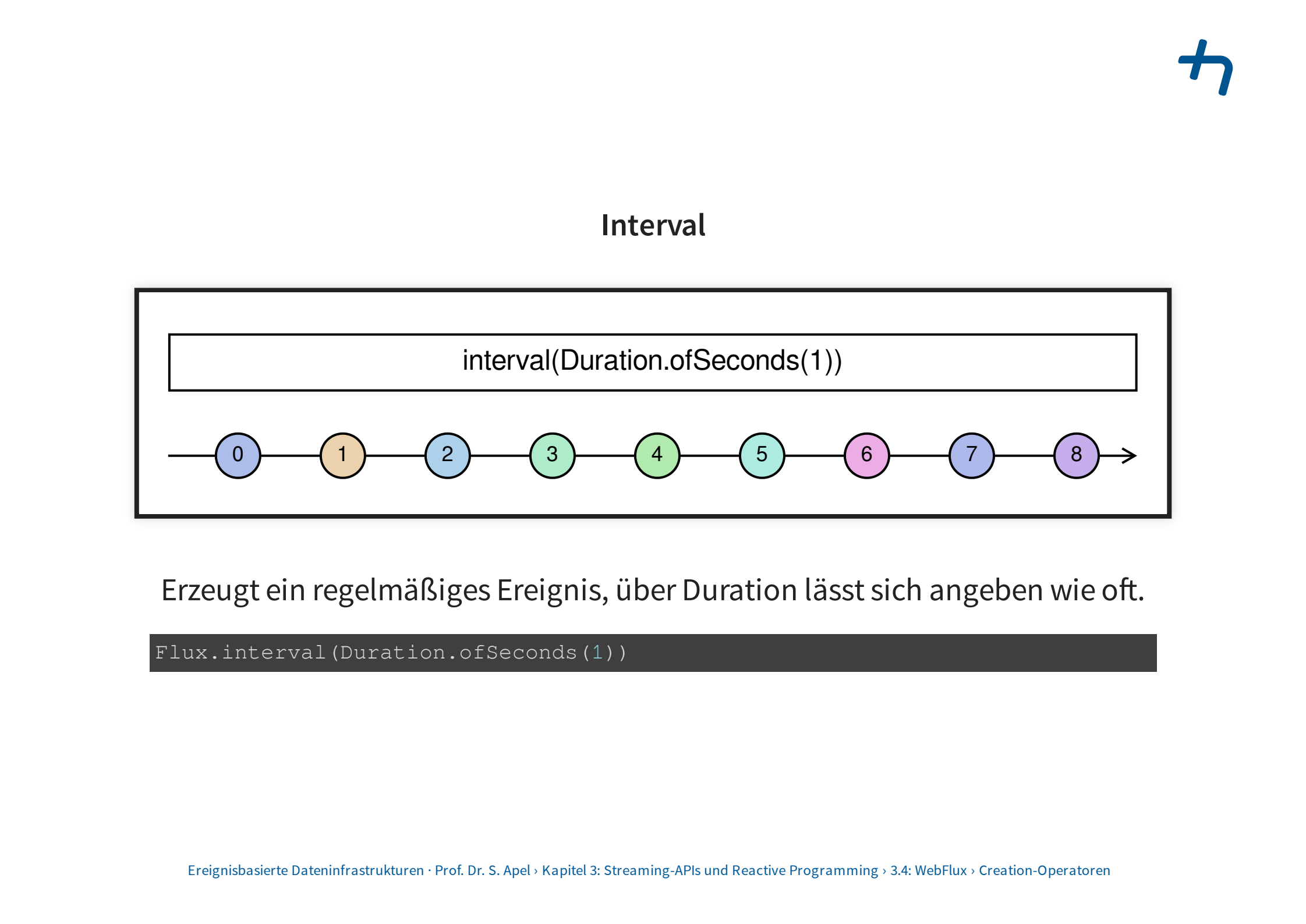
CREATION-OPERATOREN

Just



Erzeugt ein einzelnes Ereignis.

```
Mono.just("A")
```

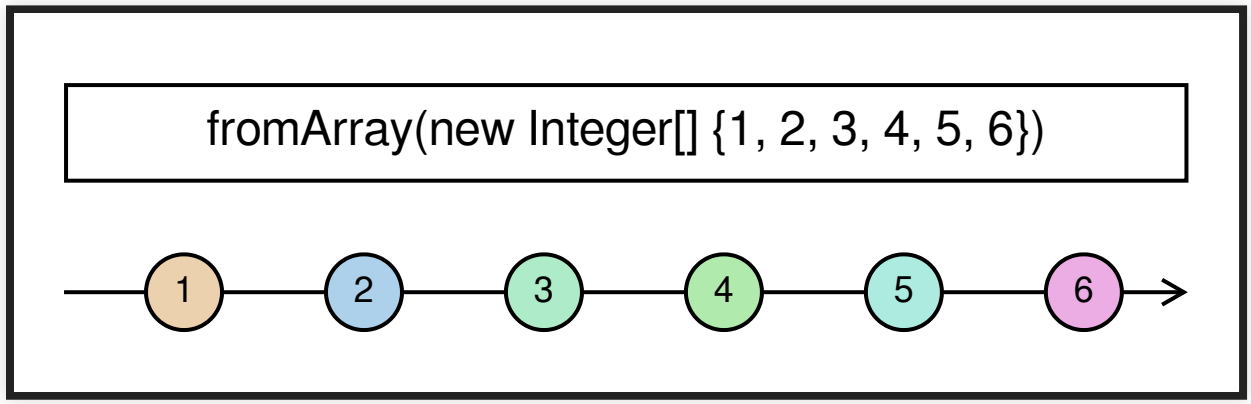


Interval

Erzeugt ein regelmäßiges Ereignis, über Duration lässt sich angeben wie oft.

```
Flux.interval(Duration.ofSeconds(1))
```

From



Erzeugt eine Ereigniskette unter Verwendung eines Arrays, alternativ sind hier auch Iteratoren oder Streams möglich.

```
Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4});
```

Create und **Generate**

Mittels `create` und `generate` lassen sich Datenströme erzeugen, die programatisch gefüllt werden können.

```
Flux.<String>create(sink -> {
    sink.next('A');
    sink.next('B');
    sink.next('C');
}).subscribe(System.out::println);
```

Hier werden drei Ereignisse in die Senke gegeben, welche dann verarbeitet werden können.

```
Flux<String> flux = Flux.create(sink -> {
    System.out.println('Created');
    sink.next('A');
});
flux.subscribe(System.out::println);
flux.subscribe(System.out::println);
```

Wichtig ist, dass hier mit jeder Subscription eine neue Senke entsteht.

Sink

```
Many<Object> many = Sinks.many().multicast().onBackpressureBuffer();
Flux<Object> flux = many.asFlux();
flux.subscribe(System.out::println);
flux.subscribe(System.out::println);
// ... later ...
many.tryEmitNext(1.0);
```

Wenn Datenströme notwendig sind, die einmalig befüllt werden und durch mehrere Subscriptions konsumiert werden sollen, bietet sich der Weg über `Sinks.many()` an.

```
Many<Object> many = Sinks.many().multicast().onBackpressureBuffer();
```

- Alternativen sind `Sinks.empty()`, `Sinks.one()` oder `Sinks.unsafe()`.
- Mittels `multicast()` und `unicast` erlauben zu steuern, ob ein oder mehrere Subscriber erlaubt sind
- Mithilfe von `onBackpressureBuffer()`, `directAllOrNothing()` oder `directBestEffort()` kann gesteuert werden, wie die Sink damit umgeht, wenn Ereignisse emittiert werden, aber noch keine Subscriptions vorliegen

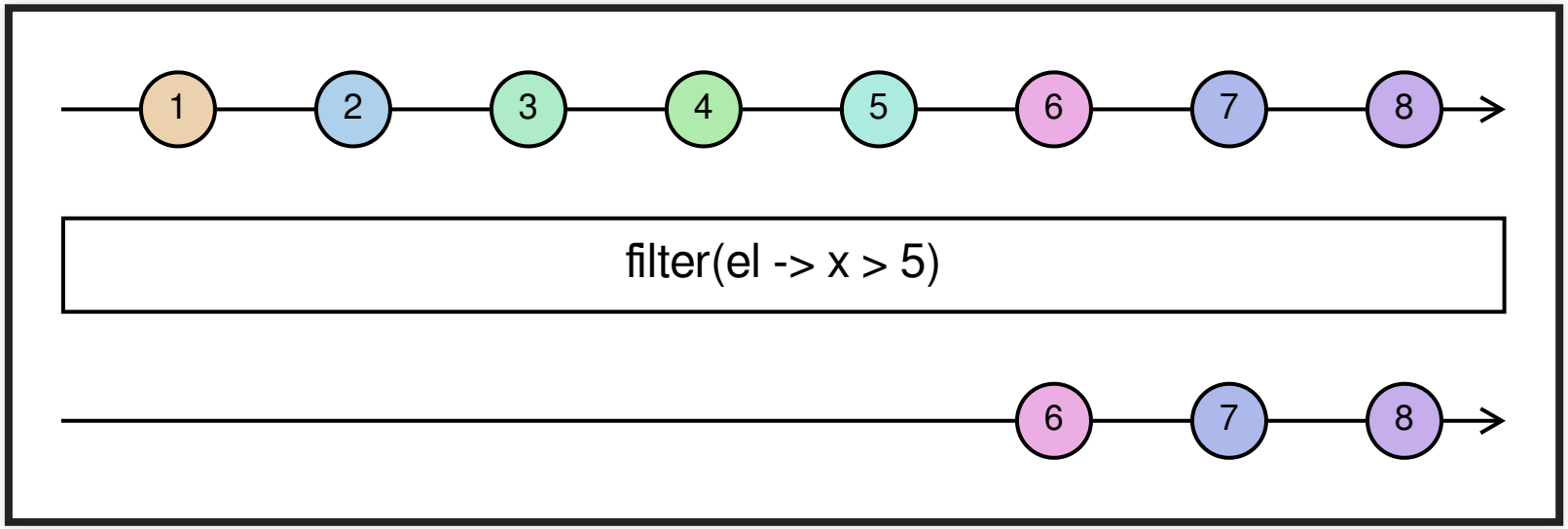
***Cold** vs **Hot** Publisher*

Wenn man `Flux.create()` und die `Sinks.many().multicast().onBackpressureBuffer().asFlux()` vergleicht kann man hier **Cold** und **Hot** Publisher differenzieren.

- Publisher können als **Hot** bezeichnet werden, wenn diese Ereignisse emittieren, bevor es einen Subscriber gibt, mit `Sinks.many()` entsteht ein Hot-Publisher
- Publisher können dagegen als **Cold** bezeichnet werden, wenn diese erst Ereignisse emittieren, wenn es einen Subscriber gibt; mittels `Flux.create()` entsteht ein Cold-Publisher.

FILTERING-OPERATOREN

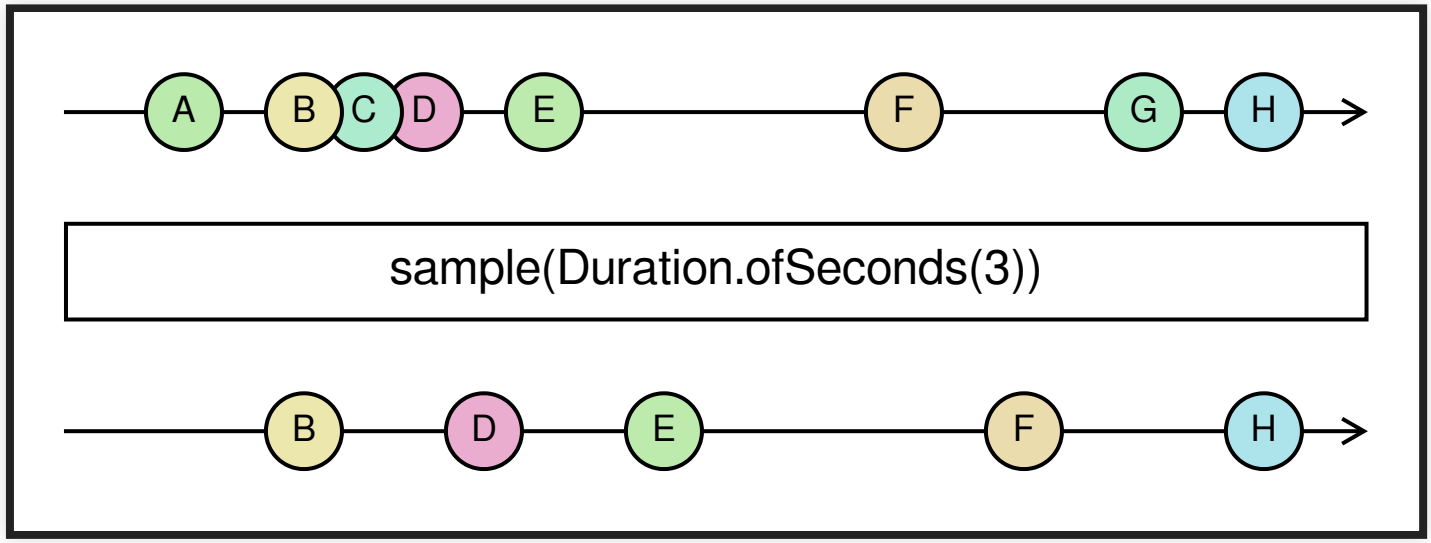
Filter



Filtert Elemente basierend auf der angegebenen Regel, z.B. nur gerade Zahlen:

```
Flux.interval(Duration.ofSeconds(1))
    .filter(el -> el % 2 == 0)
    .subscribe(System.out::println);
```

Sample und **SampleFirst**



Liefert das erste Element in einem Zeitfenster (`sampleFirst`) oder das letzte Element (`sample`).

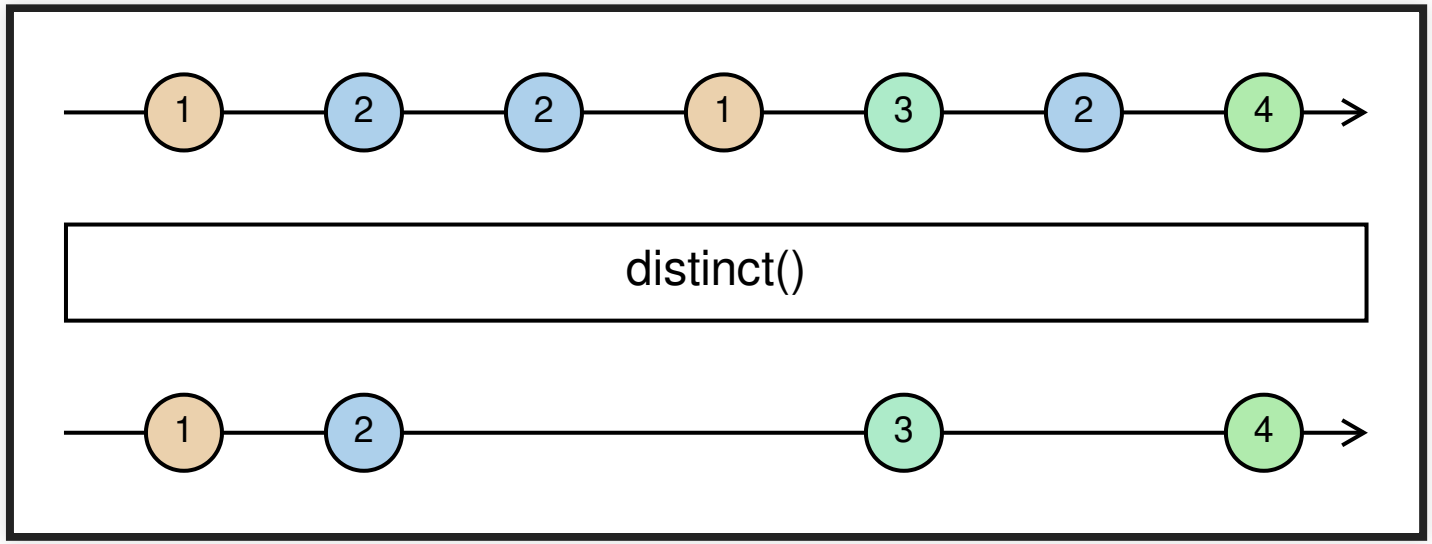
Beispiel in WebFlux

```
Flux.interval(Duration.ofSeconds(1))
    .sample(Duration.ofSeconds(4))
    .subscribe(System.out::println);
```

Es wird jede Sekunde ein Ereignis ausgelöst, aber nur alle 4 Sekunden wird das letzte sichtbar.

```
Flux.interval(Duration.ofSeconds(1))
    .sampleFirst(Duration.ofSeconds(4))
    .subscribe(System.out::println);
```

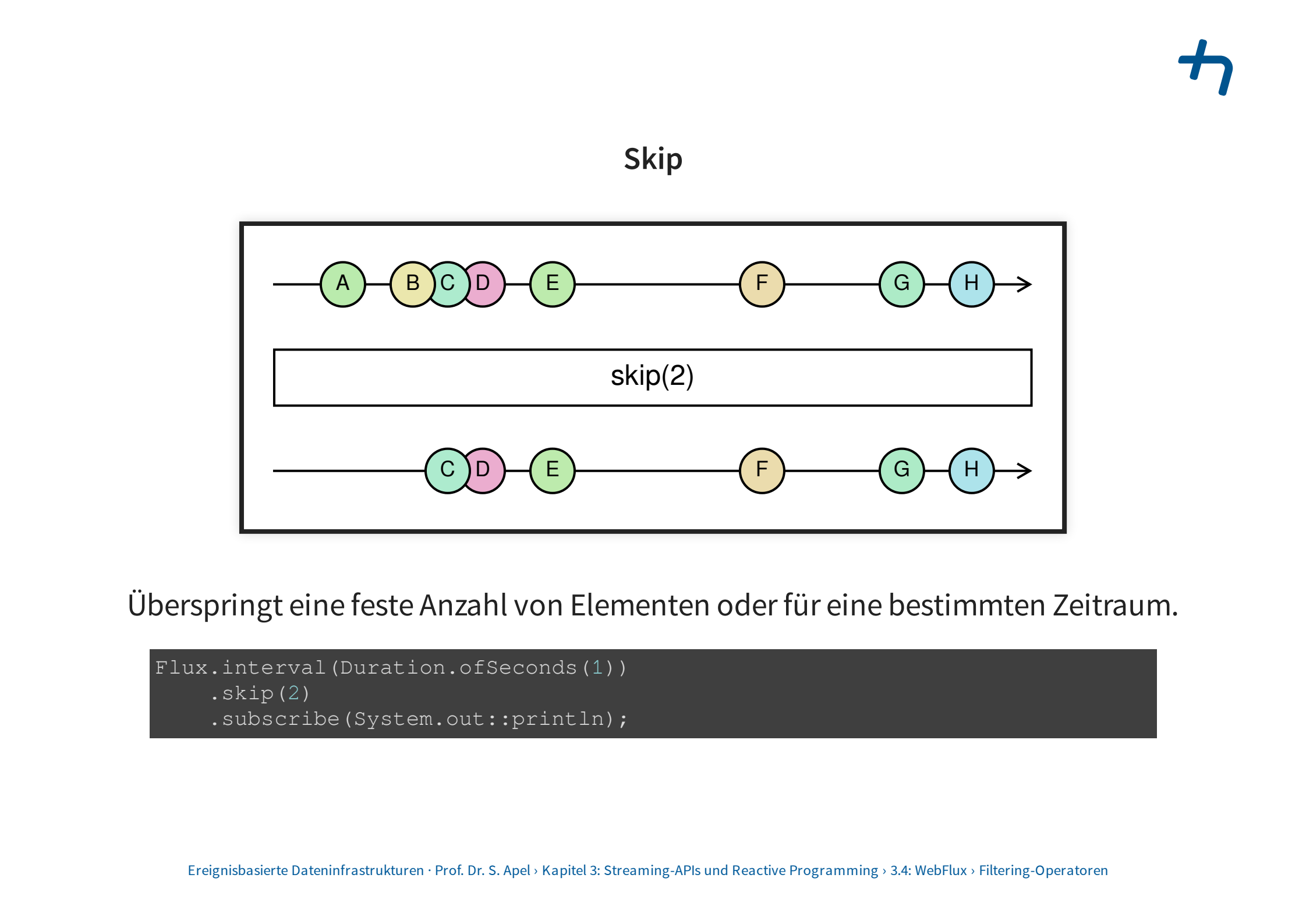
Es wird jede Sekunde ein Ereignis ausgelöst, aber nur alle 4 Sekunden wird das erste sichtbar.



Distinct

Filtert duplikate aus dem Datenstrom.

```
Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4})
    .distinct()
    .subscribe(System.out::println);
```



Skip

Überspringt eine feste Anzahl von Elementen oder für eine bestimmten Zeitraum.

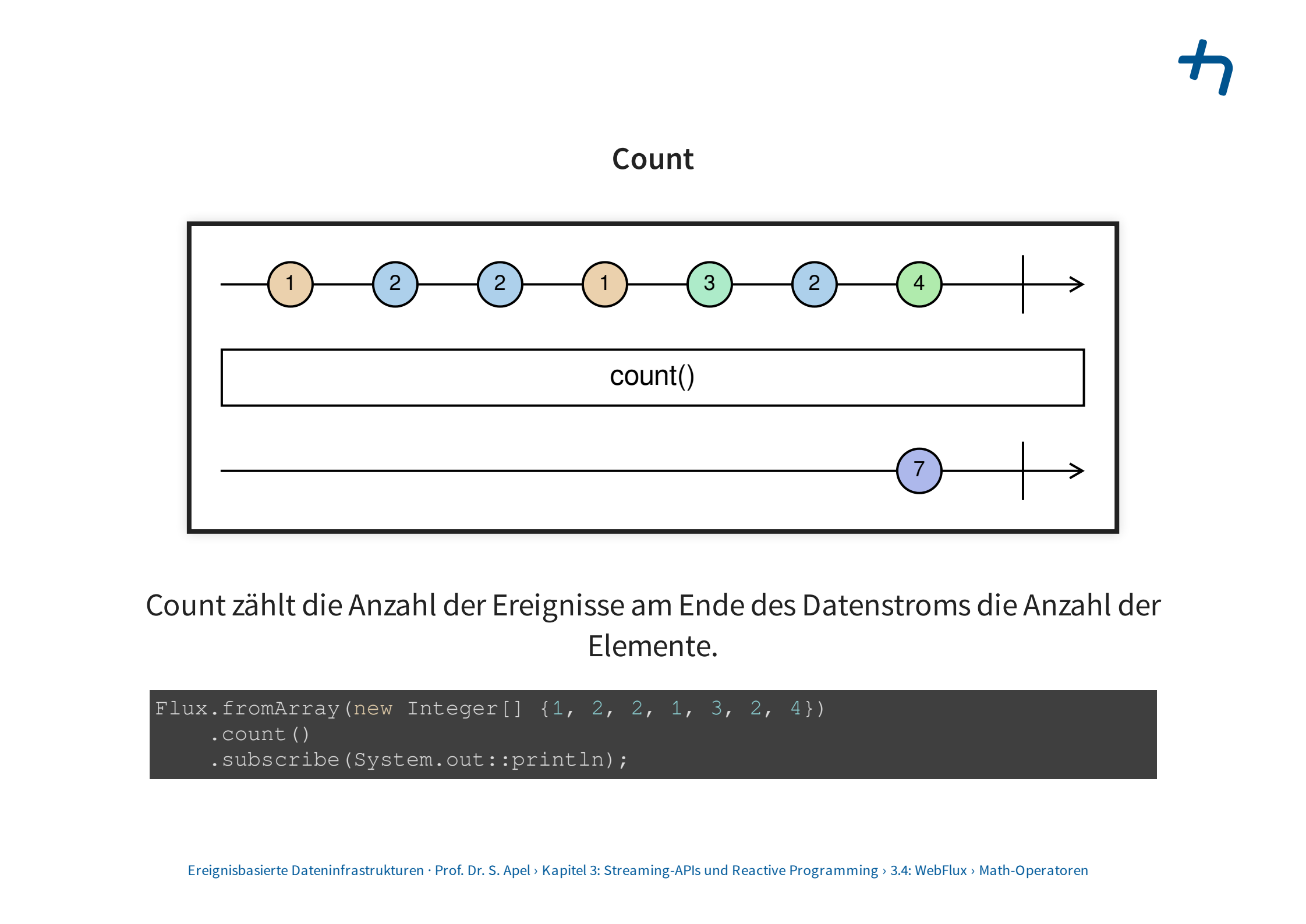
```
Flux.interval(Duration.ofSeconds(1))
    .skip(2)
    .subscribe(System.out::println);
```

Für das Überspringen eines Zeitraums ist eine Duration anzugeben.

```
Flux.interval(Duration.ofSeconds(1))
    .skip(Duration.ofSeconds(3))
    .subscribe(System.out::println);
```

Hier werden Ereignisse in den ersten drei Sekunden übersprungen.

MATH-OPERATOREN



Count

Count zählt die Anzahl der Ereignisse am Ende des Datenstroms die Anzahl der Elemente.

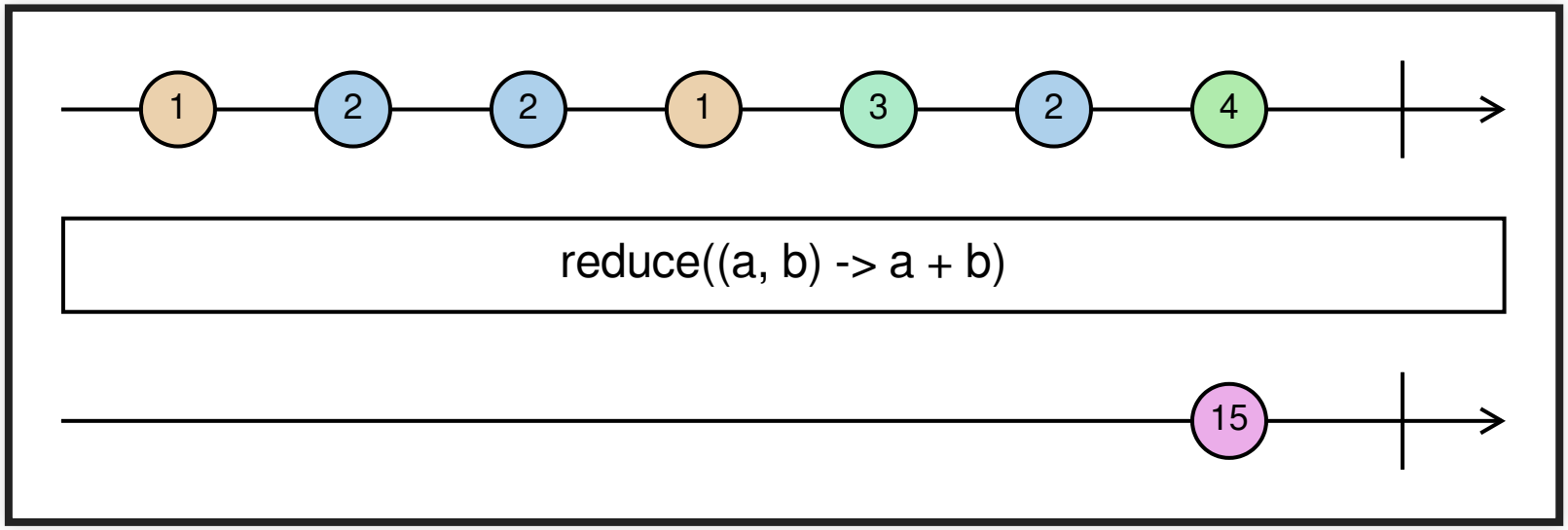
```
Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4})
    .count()
    .subscribe(System.out::println);
```

Was passiert mit folgender Verarbeitung?

```
Flux.interval(Duration.ofSeconds(1))
    .count()
    .subscribe(System.out::println);
```

Hier wird ebenfalls am Ende des Datenstroms die Anzahl der Ereignisse gezählt, dieser Datenstrom kommt jedoch nicht zum Ende.

Reduce



Reduce erlaubt es Rechen-Operationen auf alle auftretenden Ereignisse zu realisieren.

```
Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4})
    .reduce((a, b) -> a + b)
    .subscribe(System.out::println);
```

Zum Einsatz kommt ein Lambda-Ausdruck, welcher sich auf das vorherige Ergebnis und den aktuellen Wert bezieht. Bei Sequenzen mit weniger als 2 Elementen wird reduce nicht ausgelöst.

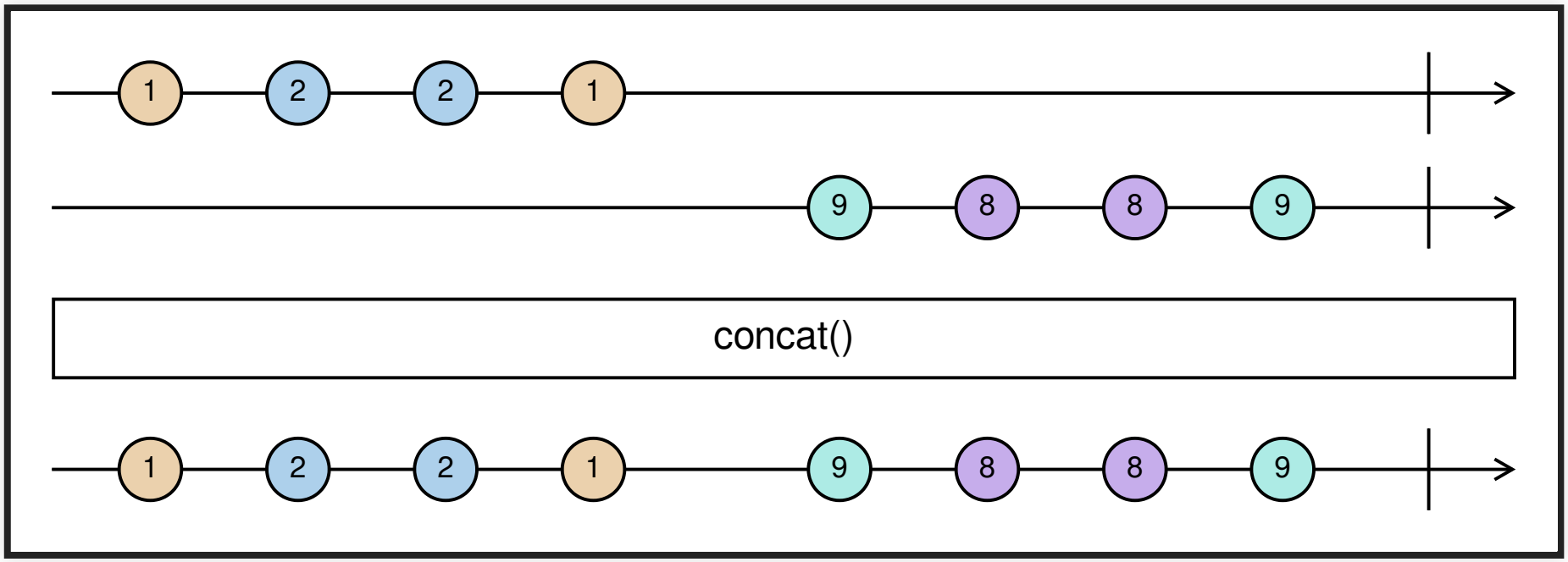
Wie kann ein Min bzw. Max auf allen Ereignissen realisiert werden?

```
Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4})
    .reduce((a, b) -> Math.max(a, b))
    .subscribe(System.out::println);
```

```
Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4})
    .reduce((a, b) -> Math.min(a, b))
    .subscribe(System.out::println);
```

COMBINATION-OPERATOREN

Concat



Kombiniert mehrere Datenströme in einen Datenström aus Ereignissen.

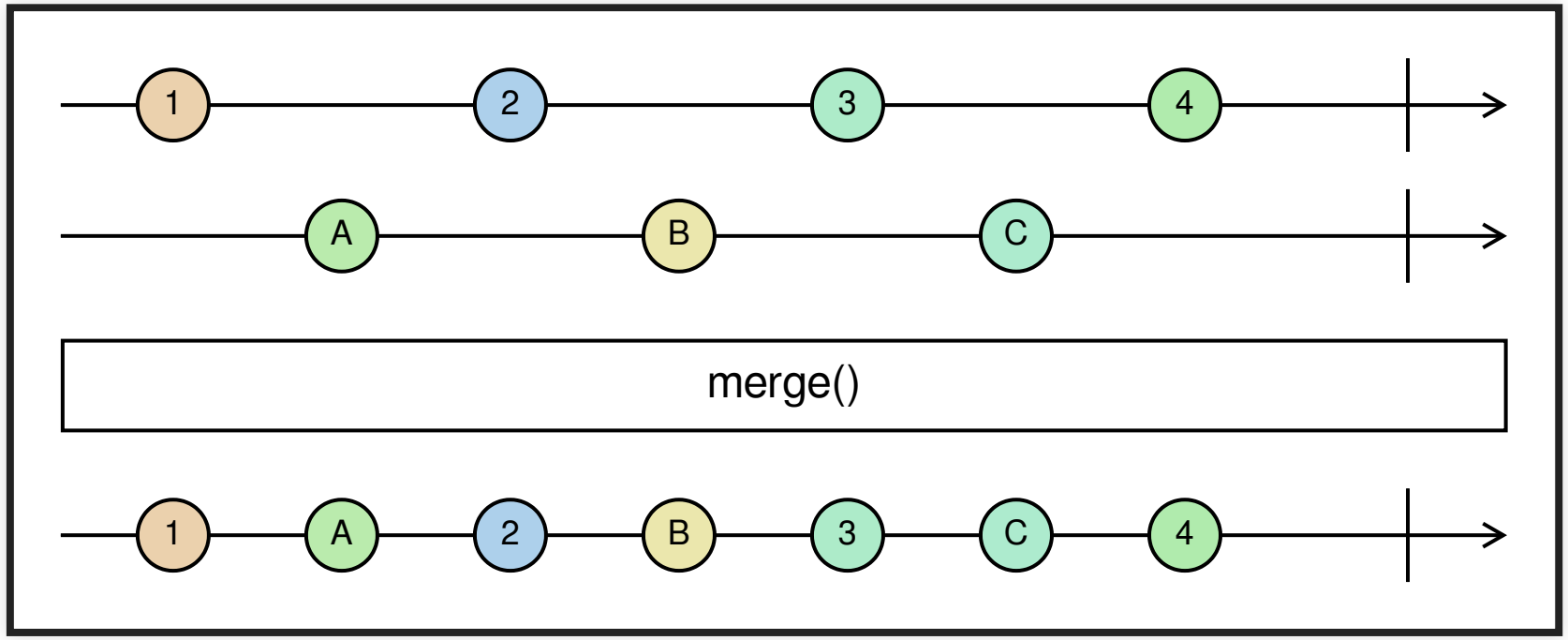
```
Flux.concat(
        Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4}),
        Flux.fromArray(new Integer[] {9, 8, 8, 9, 7, 8, 6})
).subscribe(System.out::print);
```

Ergebnis wäre hier die Kombination beider Zahlenreihen, also `122132498897 86`.

```
Flux.concat(
        Flux.interval(Duration.ofSeconds(1)).map(el -> 'A'),
        Flux.interval(Duration.ofSeconds(1)).map(el -> 'B')
).subscribe(System.out::print);
```

Ereignisse werden erst nach Terminierung zusammengefügt, kommt ein Datenstrom nicht zum Ende, wird der folgende nicht angefügt, bzw. verwendet.

Merge



Kombiniert mehrere Datenströme in einen Datenström aus Ereignissen. Dabei werden anstehende Ereignisse in den Datenstrom vollständig verarbeitet.

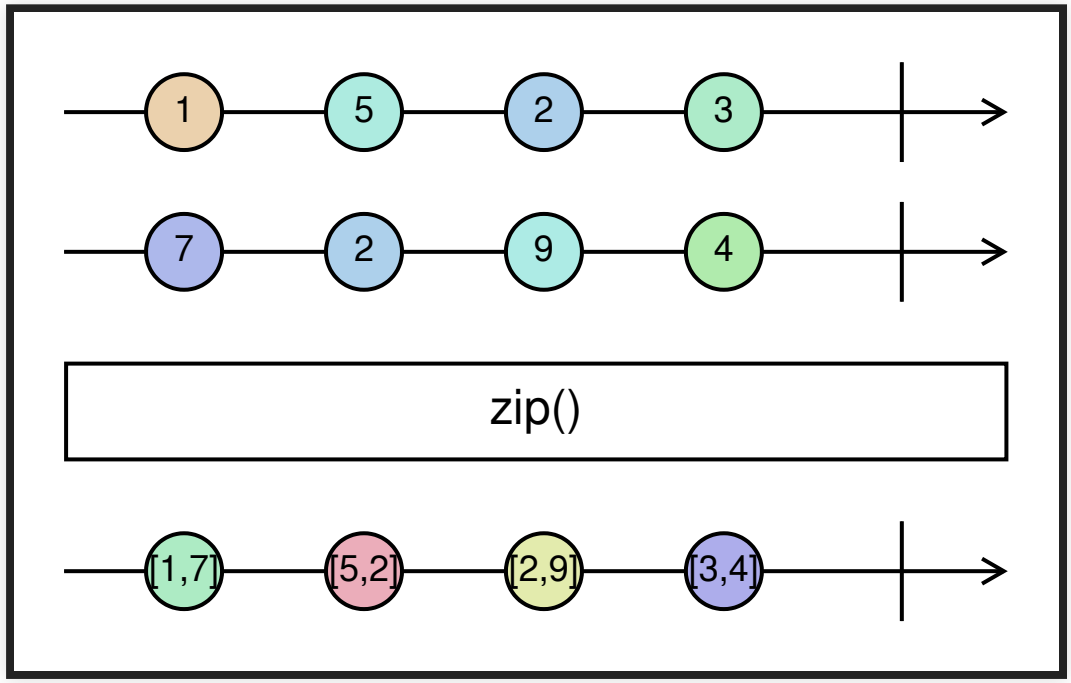
```
Flux.merge(
        Flux.interval(Duration.ofSeconds(1)).map(el -> 'A'),
        Flux.interval(Duration.ofSeconds(1)).map(el -> 'B')
).subscribe(System.out::println);
```

Bei Datenströmen, die über Zeit Ereignisse erzeugen erzeugt `merge` ein Vermischen der Ströme.

```
Flux.merge(
        Flux.fromArray(new Integer[] {1, 2, 2, 1, 3, 2, 4}),
        Flux.fromArray(new Integer[] {9, 8, 8, 9, 7, 8, 6})
).subscribe(System.out::print);
```

Grundsätzlich mischt `merge` die verfügbaren Ereignis-Sequenzen nach Verfügbarkeit zusammen. Entsprechend entsteht hier das selbe Ergebnis wie bei `concat`, also `122132498897 86`.

Zip

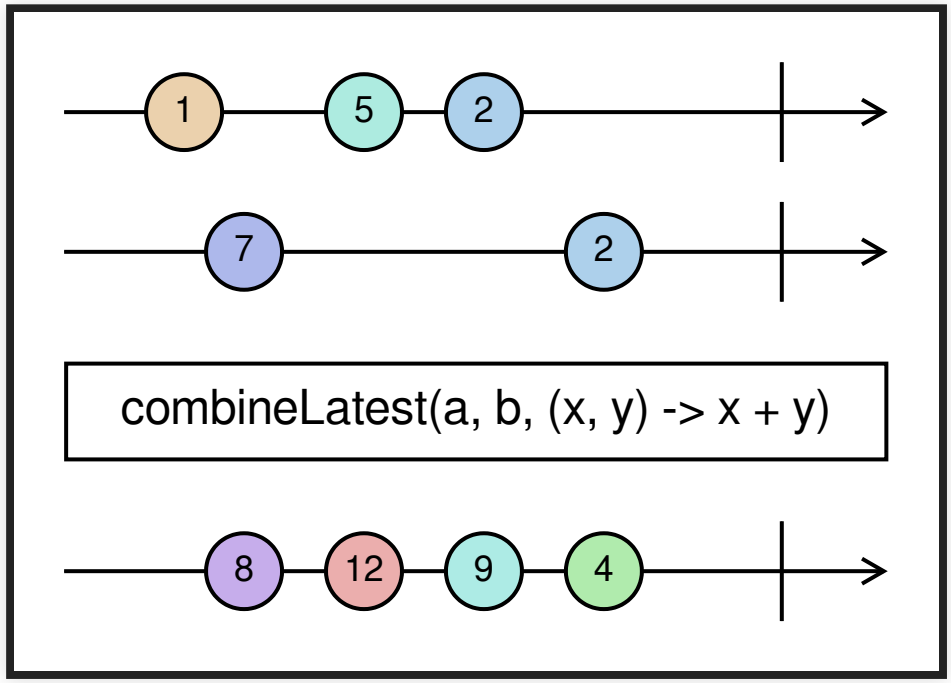


Verknüpft zwei Quellen miteinander, um ein Element als Tupel auszugeben. Der Operator arbeitet solange, bis eine Quelle abgeschlossen ist.

```
Flux<Integer> a = Flux.just(1.0, 5.0, 2.0, 3.0);
Flux<Integer> b = Flux.just(7.0, 2.0, 4.0, 9.0);
Flux.zip(a, b).subscribe(System.out::println);
```

Quelle `a` and `b` werden mittels Zip zusammengeführt, das Ergebnis sind Tupel aus den entsprechenden Werten je Datenstrom.

Combine Latest



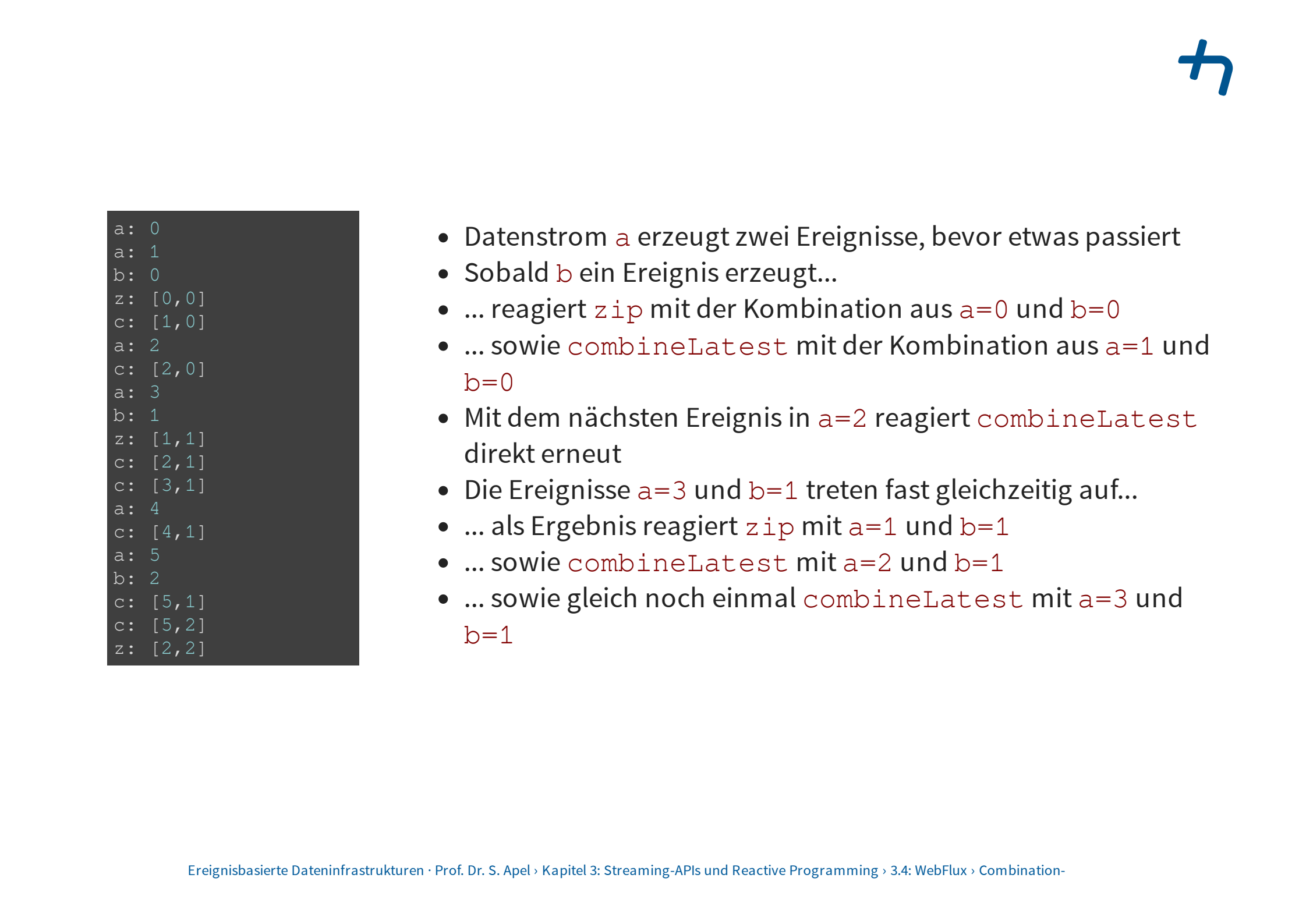
Kombiniert die zuletzt aufgetretenen Ereignisse und erlaubt es eine Kombinationslogik zu ergänzen, die beschreibt, wie mit diesen umzugehen ist.

```
Flux<Long> a = Flux.interval(Duration.ofSeconds(1));
Flux<Long> b = Flux.interval(Duration.ofSeconds(2));
Flux.combineLatest(a, b, (x, y) -> List.of(x, y))
    .subscribe(System.out::println);
```

Vergleich zwischen Zip und Combine Latest

```
Flux<Long> a = Flux.interval(Duration.ofSeconds(1));
Flux<Long> b = Flux.interval(Duration.ofSeconds(2));
a.map(el -> "a: " + el.toString()).subscribe(System.out::println);
b.map(el -> "b: " + el.toString()).subscribe(System.out::println);
Flux.zip(a, b).map(el -> "z: " + el.toString())
    .subscribe(System.out::println);
Flux.combineLatest(a, b, (x, y) -> List.of(x, y))
    .map(el -> "c: " + el.toString()).subscribe(System.out::println);
```

- Gegeben ist ein Beispiel mit zwei Datenreihen, die in einem abweichenden Takt Ereignisse erzeugen
- Zur Kontrolle werden diese direkt ausgegeben
- Für den Vergleich wird anschließend `zip` und `combineLatest` verwendet



```
a: 0
a: 1
b: 0
z: [0,0]
c: [1,0]
a: 2
c: [2,0]
a: 3
b: 1
z: [1,1]
c: [2,1]
c: [3,1]
a: 4
c: [4,1]
a: 5
b: 2
c: [5,1]
c: [5,2]
z: [2,2]
```

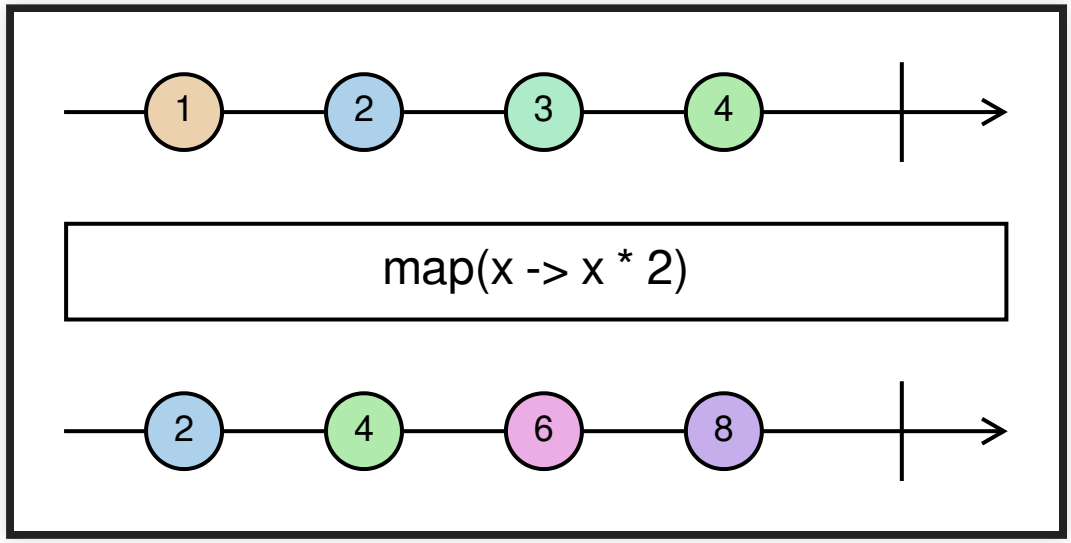
- Datenstrom `a` erzeugt zwei Ereignisse, bevor etwas passiert
- Sobald `b` ein Ereignis erzeugt...
- ... reagiert `zip` mit der Kombination aus `a=0` und `b=0`
- ... sowie `combineLatest` mit der Kombination aus `a=1` und `b=0`
- Mit dem nächsten Ereignis in `a=2` reagiert `combineLatest` direkt erneut
- Die Ereignisse `a=3` und `b=1` treten fast gleichzeitig auf...
- ... als Ergebnis reagiert `zip` mit `a=1` und `b=1`
- ... sowie `combineLatest` mit `a=2` und `b=1`
- ... sowie gleich noch einmal `combineLatest` mit `a=3` und `b=1`

Hinweise zu den Operatoren

- Ein Operator wie `zip` kann bei zeitlich asynchronen Reihen dafür sorgen, dass Ereignisse in einem Kanal auflaufen und nicht weiter betrachtet werden
- Ein Operator wie `combineLatest` wird häufiger mit dem selben Ereignis aus einer Datenreihe aufgerufen, bei der weniger Ereignisse vorliegen
- Operatoren wie `merge` und `concat` dagegen erzeugen aus mehreren Datenreihen eine kombinierte Reihe

TRANSFORMATION-OPERATOREN

Map



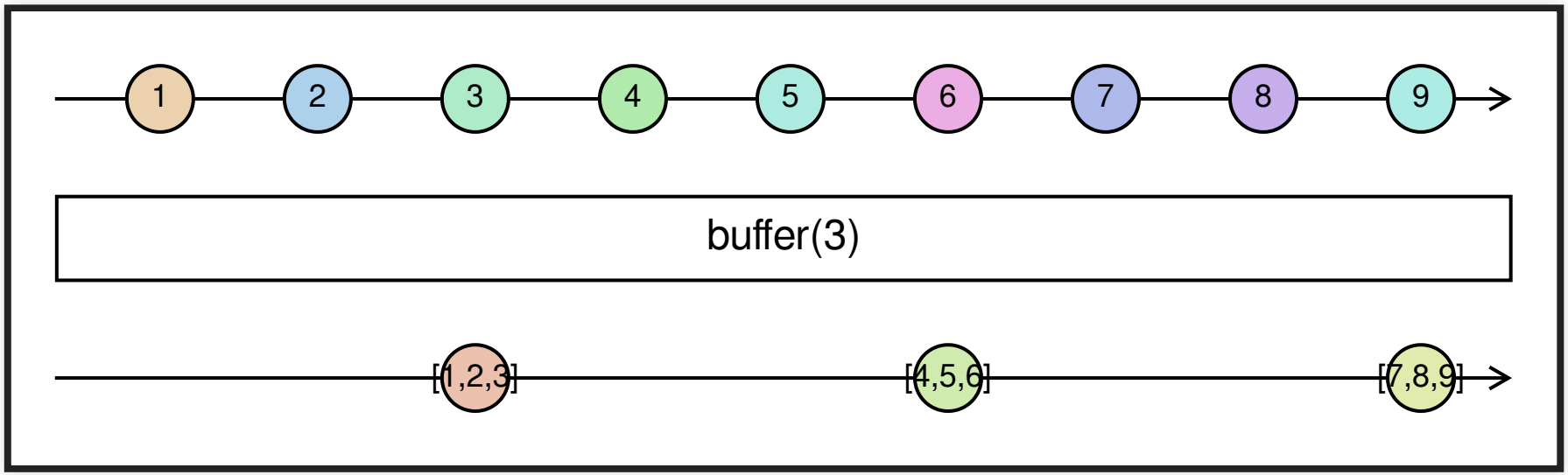
Map wird verwendet, um Ereignisse zu Transformieren. Hier wird jeder eingehende Wert mit zwei multipliziert.

```
Flux.fromArray(new Integer[] {1, 2, 3, 4})
    .map(x -> x * 2)
    .subscribe(System.out::println);
```

Für die Transformation hat der Parameter den Datentyp des Ereignisses und über den Rückgabe Datentyp des Lambda-Ausdrucks ergibt sich der Datentyp des folgenden Datenstroms.

```
Flux.fromArray(new Integer[] {1, 2, 3, 4})
    .map(x -> x * 2.0)
    .subscribe(System.out::println);
```

In diesem Fall ändert sich der Integer-Datentyp auf einen Double-Datentyp für jedes Ereignis.

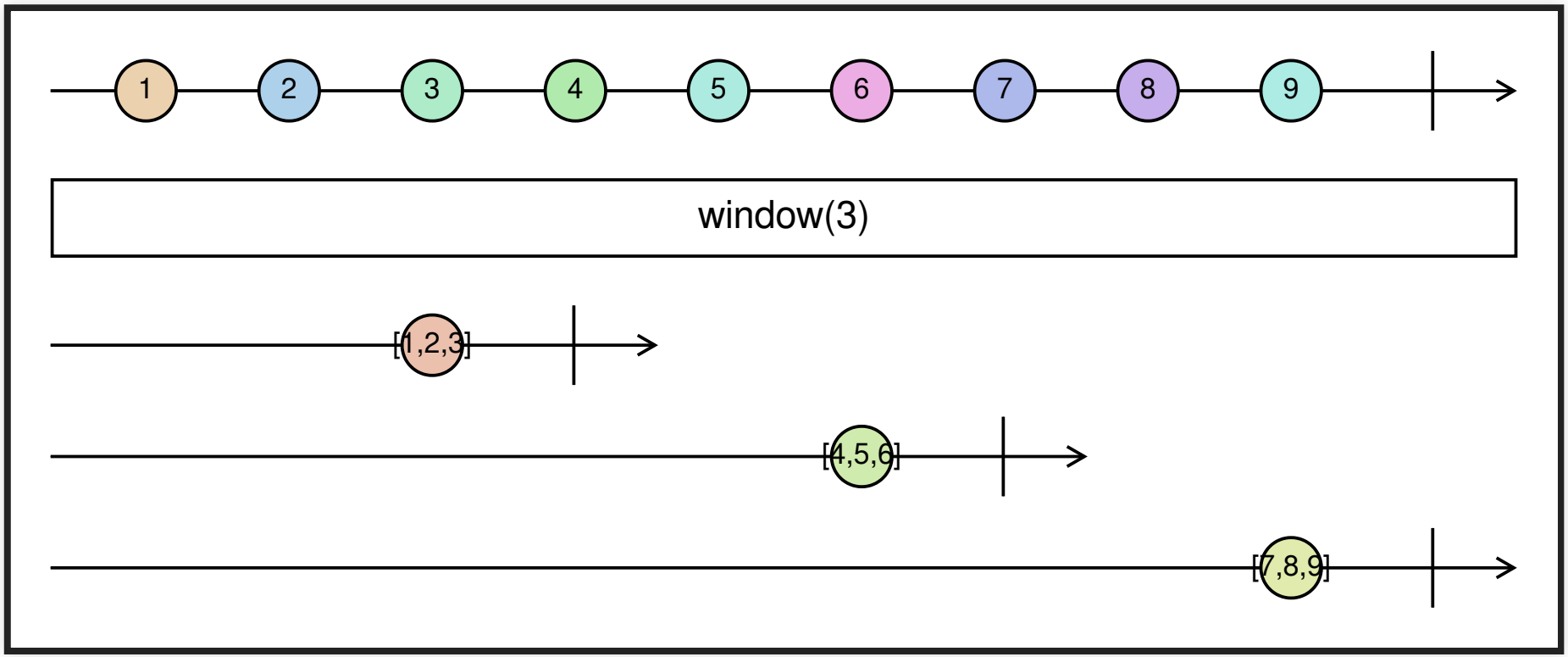


Buffer

Sammelt Ereignisse und gibt diese, wenn die Anzahl erreicht oder z.B. das Zeitfenster vorbei ist, weiter. Ergebnis ist eine Liste.

```
Flux.interval(Duration.ofSeconds(1))
    .buffer(3)
    .subscribe(System.out::println);
```

Window



Realisiert ebenfalls eine Art Buffer von Ereignissen, die Ereignisse werden jedoch als Datenstrom weitergegeben.

```
Flux.interval(Duration.ofSeconds(1))
    .window(3)
    .subscribe(win -> win
        .reduce((a, b) -> a + b)
        .subscribe(System.out::println));
```

Erzeugt als Ergebnis im Datenstrom ein eigenständige Ereigniskette. Zur Verarbeitung ist daher die Verarbeitung des Ereignisses selbst zusätzlich notwendig.

FlatMap

Erlaubt es Daten entgegen zu nehmen und auf kein, ein oder mehrere neue Ereignisse zu verteilen.

```
Flux.<List<String>>create(sink -> sink.next(Arrays.asList(
        "https://www.google.com/",
        "https://www.yahoo.com/",
        "https://www.microsoft.com/en-us/")))
    .flatMap(list -> Flux.fromIterable(list))
    .subscribe(System.out::println);
```

Mittels `map` lassen sich keine null-Ereignisse mappen, mit `flatMap` kann ein leeres Ergebnis erzeugt werden

```
Flux.interval(Duration.ofSeconds(1))
    .flatMap(el -> Math.random() > 0.5 ?
        Mono.just(el) :
        Mono.empty())
    .subscribe(System.out::println);
```

Im Beispiel wird zu 50% Wahrscheinlichkeit das Ereignis weitergeleitet oder nicht.

Weitere

- `concatMap`: Asynchron ausgegebenen Elemente werden in Publisher umgewandelt, inneren Publisher anschließend in einen einzigen Flux umgewandelt, Elemente sind nacheinander und unter Beibehaltung der Reihenfolge durch Verkettung
- `groupBy`: Teilt in dynamisch erstellte Flux, Zerteilung erfolgt auf Basis eines Key-Mappers

ÜBUNG: ZUFALLSZAHL PRO SEKUNDE

Generieren Sie eine Zufallszahl pro Sekunde in einem Datenstrom unter Verwendung von Rx.

```
Flux.interval(Duration.ofSeconds(1))
    .map(el -> Math.random())
    .subscribe(System.out::println);
```

Hierfür kann mittels Interval ein regelmäßiges Ereignis ausgelöst werden, welches anschließend mittels Map in eine Zufallszahl gewandelt wird.

ÜBUNG: AVERAGE

Ermitteln Sie den Durchschnitt einer Zahlenreihe für alle Ereignisse innerhalb eines Zeitfensters unter Verwendung von Rx.

```
Flux<Long> data = Flux.interval(Duration.ofSeconds(1));
Duration windowSize = Duration.ofSeconds(5);
Flux<Long> averages = data
    .window(windowSize)
    .flatMap(window -> window
        .reduce((sum, value) -> sum + value)
        .map(sum -> sum / windowSize.getSeconds())
    );
averages.subscribe(System.out::println);
```

Für den Durchschnitt auf einer Datenreihe muss bekannt sein, wieviele Ereignisse existieren. Diese Lösung verwendet Zeitfenster von 5 Sekunden und teilt das Ergebnis entsprechend. Nachteil ist hier, dass ggf. mehr als fünf Ereignisse im Zeitfenster vorkommen.

```
Flux<Long> data = Flux.interval(Duration.ofSeconds(1));
Duration windowSize = Duration.ofSeconds(5);
data.window(windowSize).subscribe(win ->
    win.map(el -> Tuples.of(el, 1))
        .reduce((a, b) -> Tuples.of(a.getT1() + b.getT1(), a.getT2() + b.get
        .map(el -> el.getT1() / el.getT2())
        .subscribe(System.out::println));
```

Alternativ muss ein Weg gefunden werden, wie Ereignisse vor dem `reduce` gezählt werden. Über die Klasse `Tuple`, welche mit WebFlux zur Verfügung steht, kann dies realisiert werden.

ÜBUNG: MOVING AVERAGE

Ermitteln Sie den gleitenden Durchschnitt einer Zahlenreihe innerhalb eines Zeitfensters unter Verwendung von Rx.

```
int windowSize = 3;
int overlap = 1;
Flux.just(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
    .buffer(windowSize, overlap)
    .filter(list -> list.size() == windowSize)
    .map(list -> Flux.fromIterable(list)
        .reduce((a, b) -> a + b)
        .map(res -> res / windowSize)
        .subscribe(System.out::println))
    .subscribe();
```

ÜBUNG: MAX ÜBER MEHRERE DATENSTRÖMEN ERMITTELN

Fassen Sie mehrere Datenströme zusammen und ermitteln Sie den jeweiligen Maximalwert.

```
Flux<Double> a = Flux.just(5.0, 1.0, 2.0, 10.0, 3.0, 0.0, 9.0, 4.0);
Flux<Double> b = Flux.just(7.0, 3.0, 1.0, 2.0, 6.0, 5.0, 5.0, 8.0);
```

```
Flux<Double> a = Flux.just(5.0, 1.0, 2.0, 10.0, 3.0, 0.0, 9.0, 4.0);
Flux<Double> b = Flux.just(7.0, 3.0, 1.0, 2.0, 6.0, 5.0, 5.0, 8.0);
Flux.zip(a, b)
    .map(t -> Math.max(t.getT1(), t.getT2()))
    .subscribe(System.out::println);
```

Mittels Zip lassen sich beide Reihen kombinieren, hier ist jedoch entscheidend welche Art von Datenreihen vorliegen. Sensordaten würden sich z.B. besser mit `combineLatest` kombinieren lassen.

ÜBUNG: RX-WITH-MQTT

Realisieren Sie einen Prototypen, welcher Ereignisse aus einem MQTT-Broker mittels Rx verarbeitet. Zur Datenerzeugung können Sie auf das System-Info-Beispiel zurückgreifen.

Nutzen Sie das bereitgestellte Projekt-Archiv mosquitto-system-info.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Ein Beispiel für die Rx-basierte Verarbeitung mit Spring finden Sie in der folgenden Lösung:

Nutzen Sie das bereitgestellte Projekt-Archiv rx-mosquitto-spring.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

```
Many<Tuple2<String, String>> sink = Sinks.many().multicast().onBackpressureB
try {
    client.subscribe(topic, (t, msg) ->
        sink.tryEmitNext(
                Tuples.of(t, new String(msg.getPayload()))));
    System.out.println("Subscribed to: " + topic);
} catch(MqttException me) {
    sink.tryEmitError(me);
}
Flux<Tuple2<String, String>> flux = sink.asFlux();
```

Wenn eine MQTT-Client-Verbindung hergestellt wurde, können mittels Rx-Sink und Subscription am MQTT-Client ereignisse bereitgestellt werden.

```
Pattern p = Pattern.compile("servers/(.*?)/(.*)");
Flux<DataEvent> events = flux.create(topic)
    .map(el -> Tuples.of(p.matcher(el.getT1()), el.getT2()))
    .filter(el -> el.getT1().find())
    .map(el -> new DataEvent(
            el.getT1().group(1),
            el.getT1().group(2),
            Double.valueOf(el.getT2())));
```

Für die Verarbeitung kommt eine einfache `DataEvent`-Klasse (server, type und value) zum Einsatz. Verarbeitet werden die eingehenden Tupel mittels RegEx und anschließenden Mapping.

Durch den RegEx und Map-Schritt, ist anschließend der Datentyp von T1 im Tupel ein Matcher.

```
events.filter(el -> "cpu-usage".equals(el.getType())).subscribe(System.out::
```

Abschließend könnte man z.B. nach `cpu-usage` filtern und weitere Verarbeitungen ansetzen.

BEISPIEL: RX ZUR ANBINDUNG VON HTTP-ENDPUNKTEN

Gegeben sind zwei HTTP-Endpunkte. Mittels Reactive Programming kann die Realisierung von Anfragen und die Verarbeitung zu einem gewünschten Takt und Format realisiert werden.

Im Beispiel werden die Daten über einen Websocket an Clients ausgeliefert.

Nutzen Sie das bereitgestellte Projekt-Archiv rx-handling-remote-data.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

BEISPIEL: RX-REST-CONTROLLER

Wie könnte Reactive Programming helfen, um REST-basierte Anwendungen zu realisieren?

- Datenbank-Anbindung (wenn diese nicht blockierend ist)
- Im REST-Controller als Ergebnis
- Für Geschäftslogik in Diensten
- Anstatt REST-Controller zum Routing von Anfragen

Nutzen Sie das bereitgestellte Projekt-Archiv rx-rest-spring-01.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Datenbank-Anbindung

```
public interface PersonRepository extends ReactiveCrudRepository<Person, UUI
```

Spring bietet als Erweiterung für nicht-blockierende Datenbanken-Anbindungen (wie für die MongoDB) zum `CrudRepository` ein `ReactiveCrudRepository`.

Dieses verändert Rückgabe-Datentypen auf entsprechende reactive Varianten wie `Flux` oder `Mono`. Wichtig ist, dass Ergebnisse im Sinne von Webflux erst arbeitet, wenn eine Subscription vorliegt.

REST-Controller-Beispiel

```
public class PersonEndpoint {
    @Autowired
    private PersonRepository persons;
    @GetMapping
    public Flux<PersonResponse> listAll() {
        return this.persons.listAll()
                .map(PersonResponse::fromPerson);
    }
    @PostMapping()
    public Mono<PersonResponse> getById(@RequestBody PersonRequest person) {
        return this.persons.create(person.getName())
                .map(PersonResponse::fromPerson);
    }
    // ...
```

Service-Beispiel

```
public class PersonService {
    @Autowired
    private PersonRepository persons;
    public Flux<Person> listAll() {
        return this.persons.findAll();
    }
    public Mono<Person> getById(UUID id) {
        return this.persons.findById(id);
    }
    public Mono<Person> create(String name) {
        return this.persons.save(new Person(name));
    }
    // ...
```

Alternative mit RouterFunctions

```
@Configuration
public class PersonRouter {
    @Bean
    public RouterFunction<ServerResponse> compose(PersonService service) {
        return
                route(GET("/api/v1/person"),
                        req -> ok().body(service.listAll()
                            .map(PersonResponse::fromPerson), PersonResponse
                .and(route(GET("/api/v1/person/{id}"),
                        req -> ok().body(service.getById(UUID.fromString(req
                            .map(PersonResponse::fromPerson), PersonResponse
                .and(route(POST("/api/v1/person"),
                        req -> req.body(toMono(PersonRequest.class))
                            .doOnNext(el ->service.create(el.getName())).the
    }
}
```

Nutzen Sie das bereitgestellte Projekt-Archiv rx-rest-spring-02.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

SCHEDULER

Reactor-Operatoren sind in der Regel *concurrent agnostic*: Sie legen kein bestimmtes Threading-Modell fest und laufen nur auf dem Thread, auf dem ihre onNext-Methode aufgerufen wurde.

```
Flux<Integer> flux = Flux.just(1,2,3,4,5);
flux.subscribe(el -> {
    System.out.println("a: " + el + " (start)");
    try { Thread.sleep(1000); } catch(Exception e) { e.printStackTrace(); }
    System.out.println("a: " + el + " (stop)");
});
flux.subscribe(el -> System.out.println("b: " + el));
```

Es wird erst die erste Subscription abgearbeitet , welche mittels Sleep den aktuellen Thread pausiert.

Anschließend wird die zweite Subscription bearbeitet.

Ausnahme hiervon sind Operatoren die in einem Zeit-Kontext arbeiten

```
Flux<Integer> flux = Flux.interval(Duration.ofSeconds(1));
flux.subscribe(el -> {
    System.out.println("a: " + el + " (start)");
    try { Thread.sleep(1000); } catch(Exception e) { e.printStackTrace(); }
    System.out.println("a: " + el + " (stop)");
});
flux.subscribe(el -> System.out.println("b: " + el));
```

Das Beispiel hier würde parallel beide Subscriptions bearbeiten.

In Reactor ist ein Scheduler eine Abstraktion, die dem Benutzer die Kontrolle über das Threading gibt. Ein Scheduler kann Worker erzeugen, die konzeptionell Threads sind, aber nicht notwendigerweise von einem Thread unterstützt werden. Ein Scheduler beinhaltet auch den Begriff einer Uhr, während es beim Worker nur um die Planung von Aufgaben geht.

Quelle: https://spring.io/blog/2019/12/13/flight-of-the-flux-3-hopping-threads-and-schedulers

Reactor bietet mehrere Scheduler Implementierungen

- `Schedulers.immediate()` kann als Null-Objekt verwendet werden, wenn eine API einen Scheduler erfordert, man aber die Threads nicht wechseln möchte.
- `Schedulers.single()` ist für einmalige Aufgaben, die auf einem einzigen ExecutorService ausgeführt werden können.
- `Schedulers.parallel()` ist für CPU-intensive, aber kurzlebige Aufgaben geeignet. Er kann N solche Aufgaben parallel ausführen (standardmäßig N == Anzahl der CPUs)
- `Schedulers.elastic()` und `Schedulers.boundedElastic()` eignen sich für langlebigere Aufgaben (z. B. blockierende IO-Aufgaben). Der `elastic()` erzeugt Threads nach Bedarf ohne Begrenzung, während der `boundedElastic()` dasselbe mit einer Obergrenze für die Anzahl der erzeugten Threads realisiert.

Quelle: https://spring.io/blog/2019/12/13/flight-of-the-flux-3-hopping-threads-and-schedulers

Verwenden von Schedulers

```
Flux<Integer> flux = Flux.just(1,2,3,4,5);
flux.publishOn(Schedulers.parallel()).subscribe(el -> {
    System.out.println("a: " + el + " (start)");
    try { Thread.sleep(1000); } catch(Exception e) { e.printStackTrace(); }
    System.out.println("a: " + el + " (stop)");
});
flux.publishOn(Schedulers.parallel()).subscribe(el -> System.out.println("b:
```

Mithilfe von `publishOn` kann der Scheduler für die folgenden Operatoren bestimmt werden. Im Beispiel werden beide Subscriptions in parallelen Schedulern ausgeführt.

DEBUGGING

Das Beispiel erzeugt einen Fehler bei der Verarbeitung,
warum und wo passiert der Fehler?

```
Flux.just("Orange", "Red", "Yellow")
    .filter(el -> el != null)
    .map(el -> el.substring(0, 4))
    .subscribe(System.out::println);
```

```
10:55:53.267 [main] ERROR reactor.core.publisher.Operators - Operator called default c
reactor.core.Exceptions$ErrorCallbackNotImplemented: java.lang.StringIndexOutOfBoundsE
Caused by: java.lang.StringIndexOutOfBoundsException: begin 0, end 4, length 3
    at java.base/java.lang.String.checkBoundsBeginEnd(String.java:3734)
    at java.base/java.lang.String.substring(String.java:1903)
    at de.thi.informatik.edi.flux.SimpleExample.lambda$1(SimpleExample.java:31)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuse
    at reactor.core.publisher.FluxFilterFuseable$FilterFuseableSubscriber.onNext(FluxF
    at reactor.core.publisher.FluxArray$ArrayConditionalSubscription.fastPath(FluxArra
    at reactor.core.publisher.FluxArray$ArrayConditionalSubscription.request(FluxArray
    at reactor.core.publisher.FluxFilterFuseable$FilterFuseableSubscriber.request(Flux
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFus
    at reactor.core.publisher.LambdaSubscriber.onSubscribe(LambdaSubscriber.java:119)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMa
    at reactor.core.publisher.FluxFilterFuseable$FilterFuseableSubscriber.onSubscribe
    at reactor.core.publisher.FluxArray.subscribe(FluxArray.java:50)
    at reactor.core.publisher.FluxArray.subscribe(FluxArray.java:59)
    at reactor.core.publisher.Flux.subscribe(Flux.java:8235)
    at reactor.core.publisher.Flux.subscribeWith(Flux.java:8408)
    at reactor.core.publisher.Flux.subscribe(Flux.java:8205)
```

Debugging erfordert das Debuggen der Lambda-Ausdrücke

Mit jedem Signal werden die entsprechenden Operatoren ausgeführt. Erzeugt der Operator Exceptions zur Laufzeit, und sollen diese mittels Debugger analysiert werden müssen Breakpoints in den Lambda-Ausdrücken platziert werden. Das ist nicht immer einfach.

```
Flux.just("Orange", "Red", "Yellow")
    .filter(el -> el != null)
    .map(el -> el.substring(0, 4))
    .subscribe(System.out::println);
```

Ein Breakpoint in Zeile 3 würde zur Pausierung führen, wenn die Operatoren konfiguriert werden, aber nicht wenn das Ereignis eintritt.

Breakpoints erfordern Umbau des Quellcodes

```
Flux.just("Orange", "Red", "Yellow")
    .filter(el -> el != null)
    .map(el ->
        el.substring(0, 4))
    .subscribe(System.out::println);
```

Der Ausdruck muss in einer eigenen Zeile stehen, damit der Haltepunkt für die Logik im Ausdruck Anwendung findet.

Verbesserung der Ausgaben

Alternativ kann `Hooks.onOperatorDebug();` verwendet werden, damit Reactor mehr Informationen zur Fehlerursache ausgibt.

```
Hooks.onOperatorDebug();
Flux.just("Orange", "Red", "Yellow")
    .filter(el -> el != null)
    .map(el -> el.substring(0, 4))
    .subscribe(System.out::println);
```

```
11:04:16.373 [main] ERROR reactor.core.publisher.Operators - Operator called default c
reactor.core.Exceptions$ErrorCallbackNotImplemented: java.lang.StringIndexOutOfBoundsI
Caused by: java.lang.StringIndexOutOfBoundsException: begin 0, end 4, length 3
    at java.base/java.lang.String.checkBoundsBeginEnd(String.java:3734)
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Assembly trace from producer [reactor.core.publisher.FluxMapFuseable] :
    reactor.core.publisher.Flux.map(Flux.java:6091)
    de.thi.informatik.edi.flux.SimpleExample.main(SimpleExample.java:31)
Error has been observed at the following site(s):
    |_ Flux.map ⇢ at de.thi.informatik.edi.flux.SimpleExample.main(SimpleExample.java
Stack trace:
        at java.base/java.lang.String.checkBoundsBeginEnd(String.java:3734)
        ...
```

Hiermit wird ersichtlich, mit welchen Operator der Fehler in Verbindung steht.

Aktivieren von Log-Meldungen

```
Hooks.onOperatorDebug();
Flux.just("Orange", "Red", "Yellow")
    .log()
    .filter(el -> el != null)
    .map(el -> el.substring(0, 4))
    .subscribe(System.out::println);
```

Mittels Log-Operator können zusätzlich Log-Meldungen ausgegeben werden, welche den Ablauf und die Fehlersuche ebenfalls vereinfachen können.

```
11:06:31.305 [main] INFO reactor.Flux.OnAssembly.1 - | onSubscribe([Fuseable] FluxOnA
11:06:31.308 [main] INFO reactor.Flux.OnAssembly.1 - | request(unbounded)
11:06:31.308 [main] INFO reactor.Flux.OnAssembly.1 - | onNext(Orange)
Oran
11:06:31.308 [main] INFO reactor.Flux.OnAssembly.1 - | onNext(Red)
11:06:31.311 [main] INFO reactor.Flux.OnAssembly.1 - | cancel()
```

Hieraus wird ersichtlich, dass bei Behandlung von `Red` der Abbruch erfolgte.

3.5 VOR- UND NACHTEILE

VORTEILE

Lazyness

Reactive Programming bedeutet, dass nichts passiert, bevor nicht eine entsprechende Subscription vorliegt! Berechnungen werden standardmäßig lazy und werden nur bei Bedarf ausgeführt.

```
Flux.just("A", "B", "C").map(el -> {
    System.out.println(el);
    return el;
});
```

Prägnanter Code

Implementierungen werden prägnanter und kompakter.

```
List<Integer> list = List.of(1, 5, 4, 9, 2, 8, 7, 6);
List<Integer> evens = new ArrayList<Integer>();
List<Integer> odds = new ArrayList<Integer>();
for(Integer el : list) {
    if(el % 2 == 0) {
        evens.add(el);
    } else {
        odds.add(el);
    }
}
for (Integer el : odds) {
    System.out.println(el);
}
for (Integer el : evens) {
    System.out.println(el);
}
```

Beispiel in Reactor

```
Flux<Integer> just = Flux.just(1, 5, 4, 9, 2, 8, 7, 6);
List<Integer> evens = just.filter(el -> el % 2 == 0)
        .collect(Collectors.toList())
        .block();
List<Integer> odds = just.filter(el -> el % 2 != 0)
        .doOnEach(System.out::println)
        .collect(Collectors.toList())
        .block();
Flux.fromIterable(evens).subscribe(System.out::println);
Flux.fromIterable(odds).subscribe(System.out::println);
```

Backpressure

Mittels Backpressure kann gesteuert werden, wieviele Ereignisse verarbeitet werden sollen.

Reduktion von Callbacks

Asynchrone Programmierung ist vereinfacht und kann mittels reaktiver Programmierung einfach umgesetzt werden. Dabei muss nicht auf Callbacks zurückgegriffen werden, um die in Zukunft auszuführende Logik zu hinterlegen.

Maintainability

Die Wartbarkeit von Quellcode kann sich verbessern, da einzelne Transformations- / Verarbeitungsschritte verwendet werden können, um komplexe Probleme zu lösen. Probleme mit Nachverfolgbarkeit und Möglichkeiten zum Debuggen lassen diesen Punkt aber auch als Nachteil darstellen.

NACHTEILE

Lernkurve

Damit mit Rx-Werkzeugen gearbeitet werden kann, ist ein grundlegendes Verständnis von Operatoren notwendig und ein kreativer Einsatz um Verarbeitungs- und Transformationsprozesse gezielt zu implementieren.

Debugging und **Maintainability**

Fehler in der Operatorenkette sind unter Umständen schwierig zu verfolgen bzw. zu Debuggen. Hier können weitere Werkzeuge helfen, um ein klares Bild zu erhalten, wann wo etwas passiert ist.

Speicher

Durch verschiedene Operatoren und Mechanismen kann es passieren, dass Ereignisse auflaufen und die Verarbeitung nicht im selben Maß realisiert werden kann. Das Ergebnis wäre, dass der Speicher voll läuft.

Fehlerbehandlung

Behandlung von Fehlern, die wärend der Laufzeit auftreten können, spielen ebenfalls eine Rolle.

```
Flux.just("Orange", "Red", "Yellow")
    .flatMap(el -> Mono.just(el).map(t -> t.substring(0, 4)))
    .onErrorContinue((e, msg) -> {
        System.err.println("Error when handling: " + msg);
        e.printStackTrace();
    })
    .subscribe(System.out::println);
```

3.6 PITFALLS

Bei der Arbeit mit Reactive Programming und Datenströmen gibt es mehrere Aspekte zu berücksichtigen. Einige davon werden im folgenden exemplarisch gezeigt, weitere finden sich in der Quelle.

Quelle: https://medium.com/jeroen-rosenberg/10-pitfalls-in-reactive-programming-de5fe042dfc6

Verarbeitung startet mit Subscription

```
interface Service {
    Mono<Void> update(String s);
}
class Foo {
    private final Service service;
    void problem() {
        service.update("foo");
    }
}
```

Verwendung von Reactive-Programming erfordert, dass am Ende der Verarbeitungskette eine Subscription steht.

Quelle: https://medium.com/jeroen-rosenberg/10-pitfalls-in-reactive-programming-de5fe042dfc6

Exceptions

```
try {
    Flux.just("Foo").map(el -> el.substring(0, 4)).subscribe();
} catch (Exception e) {
    System.out.println("Exception!");
}
```

Exceptions werden anders behandelt, als zu erwarten wäre. Hier wird ein Stack-Trace ausgegeben, der Catch-Block wird nicht verwendet.

Verschachtelte Datenreihen

```
interface Service {
    Mono<String> create(String s);
    Mono<Void> update(String s);
}
class Foo {
    private final Service service;
    Mono<Integer> problem() {
        return service.create("foo").map(foo -> {
            service.update(foo).subscribe();
            return foo.length();
        });
    }
}
```

Die innere Ausführung könnte irgendwann ausgelöst werden, wenn der Aufrufer von `problem` eine Subscription auslöst.

Problem ist hier, dass die innere Subscription losgelöst ist und keine Kontrolle diese existiert.

```
interface Service {
    Mono<String> create(String s);
    Mono<Void> update(String s);
}
class Foo {
    private final Service service;
    Mono<Integer> problem() {
        return service.create("foo").flatMap(foo ->
            service.update(foo).thenReturn(foo.length())
        );
    }
}
```

Besser wäre hier die Ausführung zu bringen

State Management

```
interface Service {
    Flux<Integer> findAll();
}
class Foo {
    private final Service service;
    Flux<Integer> problem() {
        AtomicInteger count = new AtomicInteger();
        return service.findAll()
            .doOnNext(count::addAndGet)
            .doOnComplete(() -> System.out.println("Sum: " + count.get()));
    }
}
```

Wenn jemand `problem().subscribe()` aufruft, sollte die Summe ausgegeben werden. Der Zustand, der in `count` verwaltet wird, ist jedoch an die Verarbeitung gebunden. Wird diese mehrmals verwendet, z.B. 2x subscribe, ist die Summe am Ende doppelt so hoch.

Parallism

```
interface Service {
    Flux<String> findAll();
    Mono<Void> operation(String s);
}
class Foo {
    private final Service service;
    Flux<Void> problem() {
        return service.findAll()
            .flatMap(service::operation);
    }
}
```

Abhängig davon, was `findAll` liefert und wie `operation` damit umgeht, kann es hier zu ungewolltet parallilität kommen.