ARS

Motivation

- Ströme von Daten auf übersichtliche Weise bearbeiten
- Vordefinierte Algorithmen einfach auf Events anwenden
- Komplexität asynchroner Programmierung ein Stück weit reduzieren

ARS

Motivation im Beispiel

Hab' was gemacht

Mach 'was

```
buttonElement.addEventListener('click,
    onMachWasButtonClicked);
```

Berg


```
Rx.Observable.fromEvent(inputElement, 'keyup')
  .pipe(
    .map( e => e.target.value )
    .filter( term => term.length > 2)
    .switchMap(searchForImage)
  )
  .subscribe(
    result => { ... },
    err => { ... }
  );
```

ARS

Pull vs. Push Programmierung

	Einzelwert	Mehrere Werte

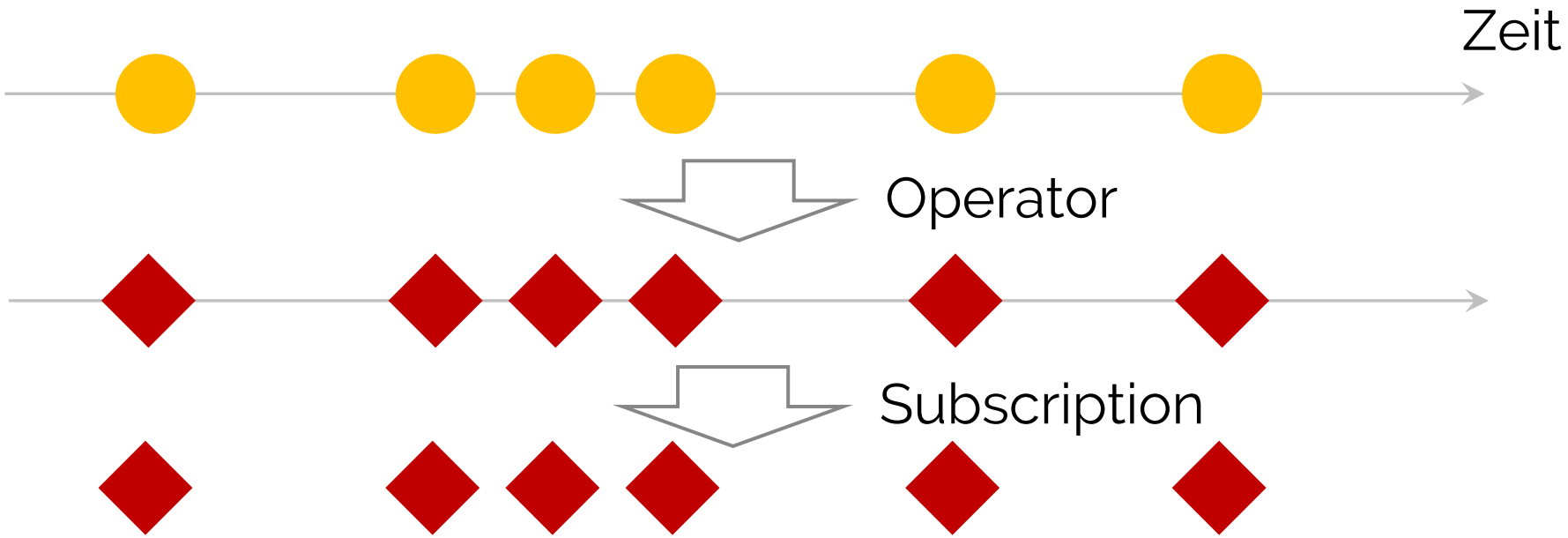
Data Pull	**Funktionsaufruf**
	`let result = getValue();`
	synchron
		Iteratoren/Generatoren
		`let iterator = getValueList();`
		`while (!iterator.done) {`
		`  let result = iterator.next();`
		`}`
		synchron

Data Push	**Promise**
	`getValueAsync()`
	`  .then( result => { ... } )`
	asynchron
		Observables
		`getValueObservable()`
		`  .subscribe( result => { ... } )`
		asynchron synchron

ARS

Observables

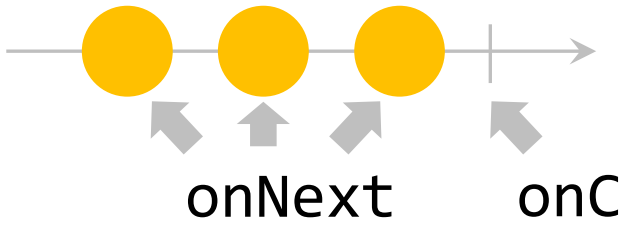
- Zeitfolge von **Events**, auf die programmatisch reagiert werden können (“Strom von Daten”)
- Einzelne Events können mit Hilfe von **Operatoren** bearbeitet werden
- Nach Anwendung von Operatoren, kann das Endresultat aller Operatoren in einer **Subscription** empfangen werden

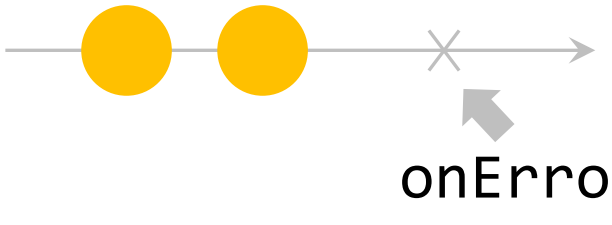


ARS

Observable Events

- Observables stellen drei Events bereit, auf die Operatoren und Subscriptions reagieren können
  - `onNext` : Der nächste Wert der Observable ist bereit zur Bearbeitung
  - `onError:` Die Observable enthält einen Fehler und wird keine weiteren Werte mehr liefern
  - `onCompleted:` Die Observable „ist fertig“ und wird keine weiteren Werte senden.






```
.subscribe(onNext, [onError, [onCompleted]])
```

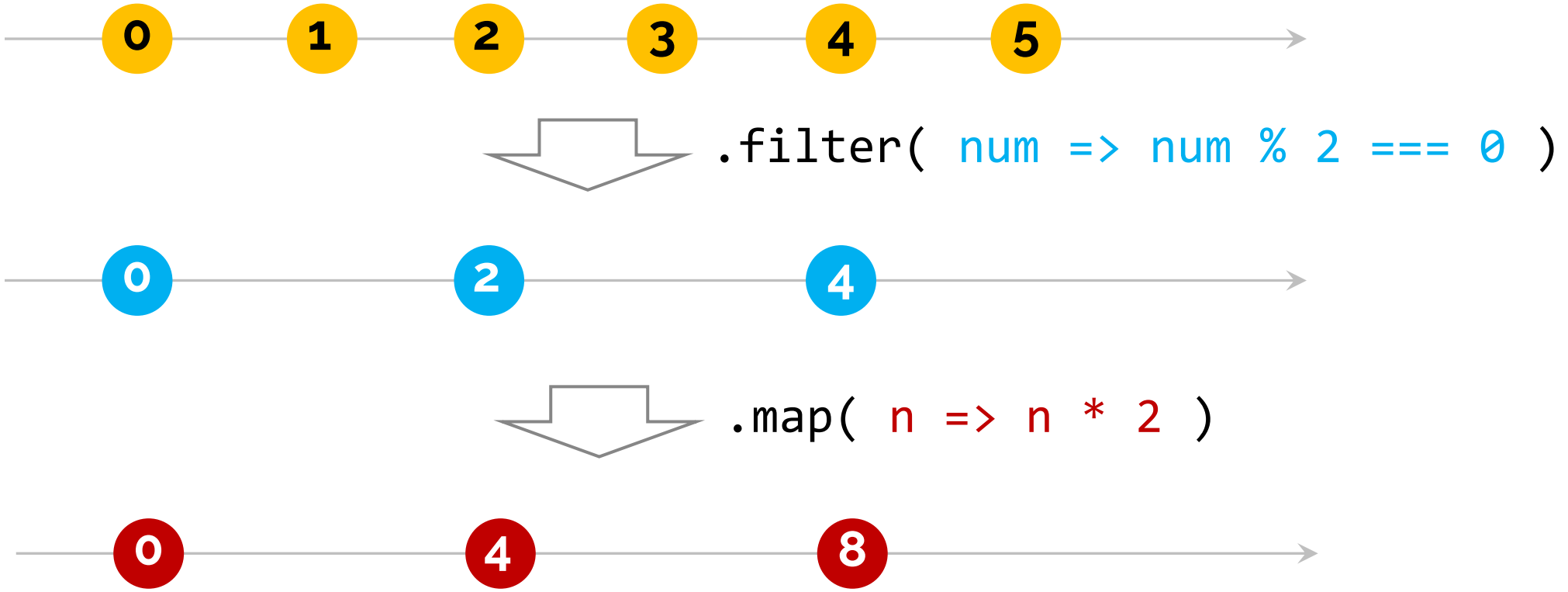
ARS

Funktionale Programmierung Merkmale

1. **Funktionen** sind „First-class citizens“ und **können** sowohl **an andere Funktionen übergeben werden** als auch von diesen zurückgegeben werden.
2. **Funktionen haben keine Seiteneffekte** und liefern daher für gleichen Eingabewerte das gleiche Resultat („pure functions“)
3. Variablen ändern ihren Wert nicht nach der ersten Zuweisung („Immutable Data“)
4. Wegen (3) werden Schleifen mit Rekursion ersetzt

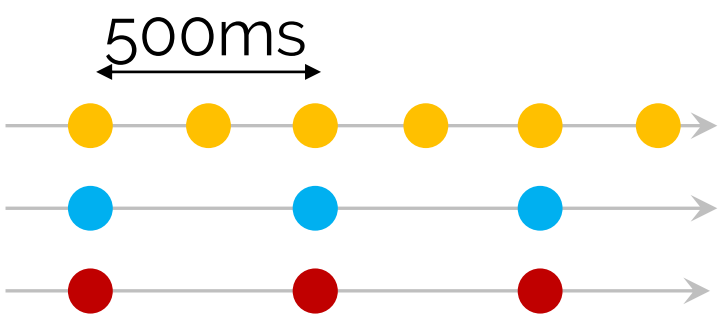
ARS

Operatoren Beispiel



ARS

Operatoren Beispiel

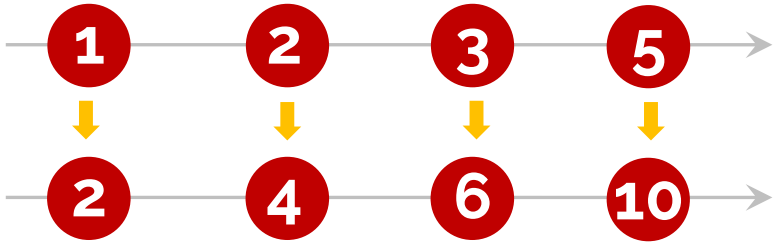



```
Rx.Observable.interval(250)
  .pipe(
    .filter( num => num % 2 === 0 )
    .map( n => 2 * n  )
  )
  .subscribe(
    res => { console.log(res) }
  );
```

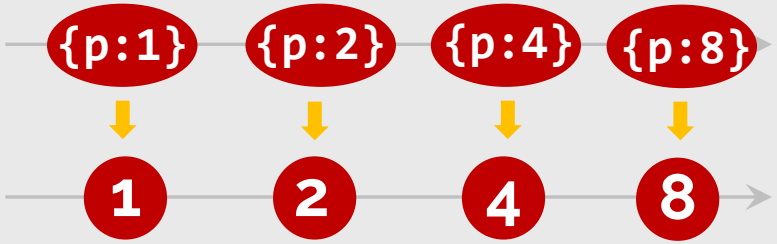
ARS

Transforming Operators

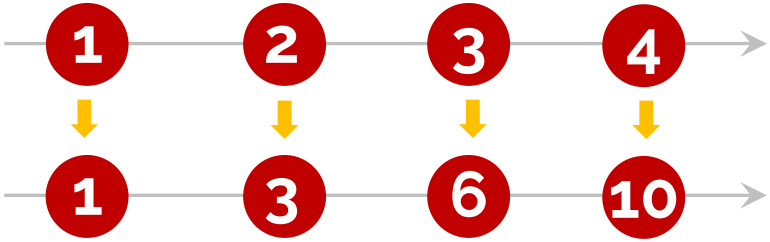
```
.map(x => x * 2)
```



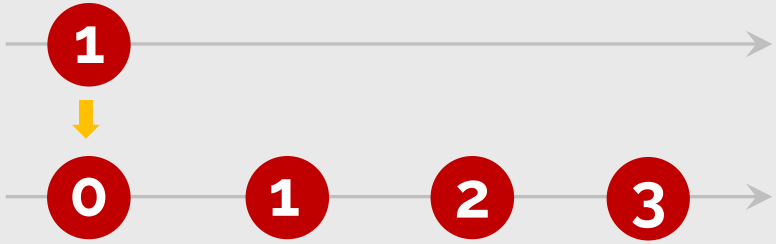

```
.pluck('p')
```




```
.scan((acc, x) => acc + x, 0)
```



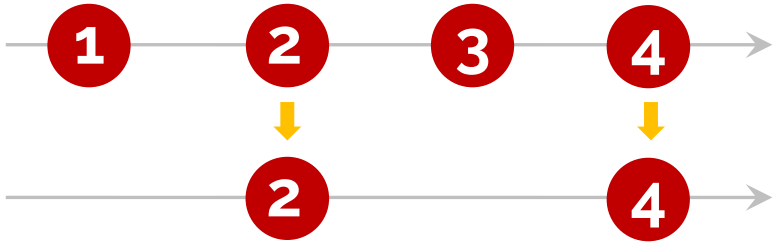

```
.mergeMap(other)

other = Rx.Observable.interval(100)
```



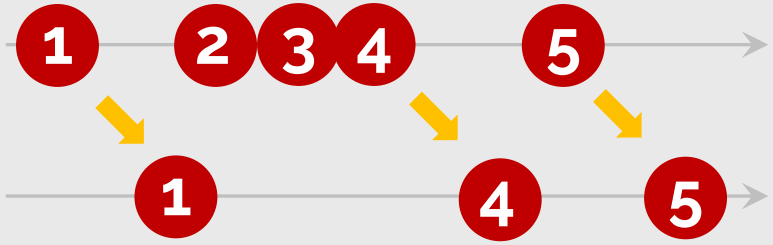
ARS

Filteroperatoren

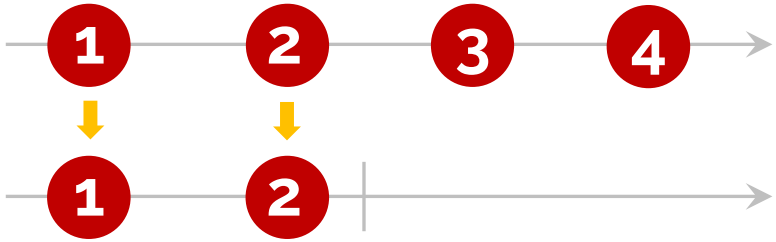
```
.filter(x => x % 2 === 0)
```



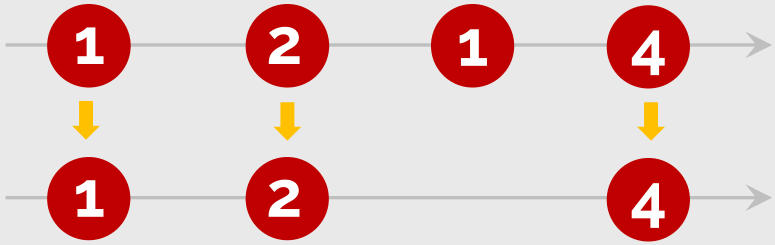

```
.debounce(250)
```




```
.take(2)
```



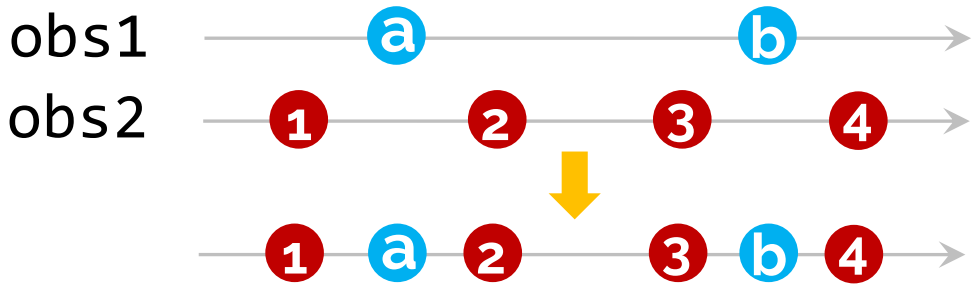

```
.distinct()
```



ARS

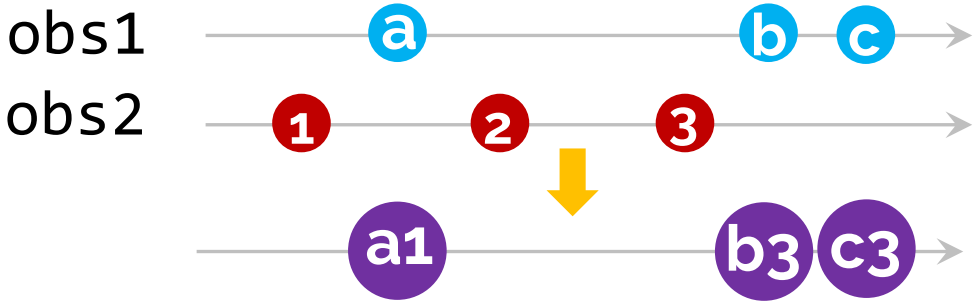
Kombinationsoperatoren

```
obs1.merge(obs2)
```




```
obs1.zip(obs2, (x,y) => x+y)
```


```
obs1.withLatestFrom(obs2)
```



ARS

Asynchronität in Rx.js

Nutzer clickt “Submit”

Sammle Werte

Validiere

Baue Serveranfrage

Sende zum Server

A

Bearbeite die Antwort

Resultat / Fehler Anzeigen

```
Rx.Observable.fromEvent(el, 'click')
  .pipe(
    .withLatestFrom(o1, o2, o3, ...)

    .filter(isFormValid)

    .map(createRequestObject)

    .mergeMap(sendRequestToServer)

    .map(processResponse)
  )
  .subscribe(
      response => { ... },
      error => { ... }
  )
```

ARS

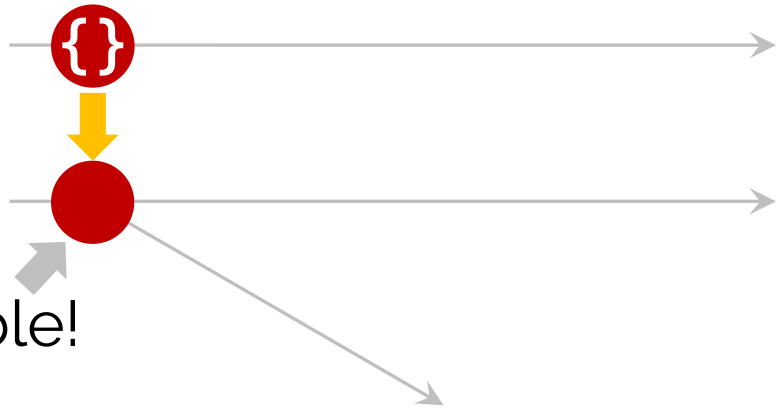
Asynchronität in Rx.js

Sende zum Server (A)

```
.mergeMap(sendRequestToServer)
```

```
let http: HttpClient;
const POST_URL = '/form-prozessor';

sendRequestToServer(requestData: ServerRequestData): Observable<any> {
    let reqOptions = mergeOptions(defaultOptions, requestData.options)
    return http.post(POST_URL, requestData.body, reqOptions)
}
```

```
.map(sendRequestToServer)
```



Event enthält eine Observable!

(nicht die Daten vom Server)

ARS

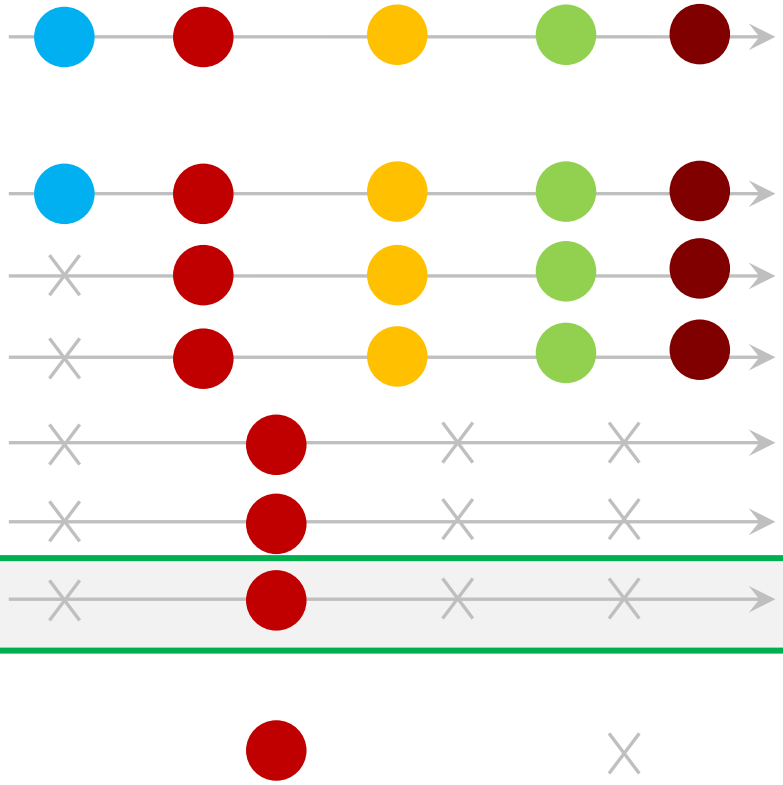
Fehlerverhalten

```
Rx.Observable.fromEvent(el, 'click')
  .pipe(
    .withLatestFrom(o1, o2, o3, ...)
    .filter(isFormValid)
    .map(createRequestObject)
    .mergeMap(sendRequestToServer)
    .map(processResponse)
  )
  .subscribe(
      response => { ... },
      error => { ... }
  )
```

ARS

Fehlerbehandlung mit retry()

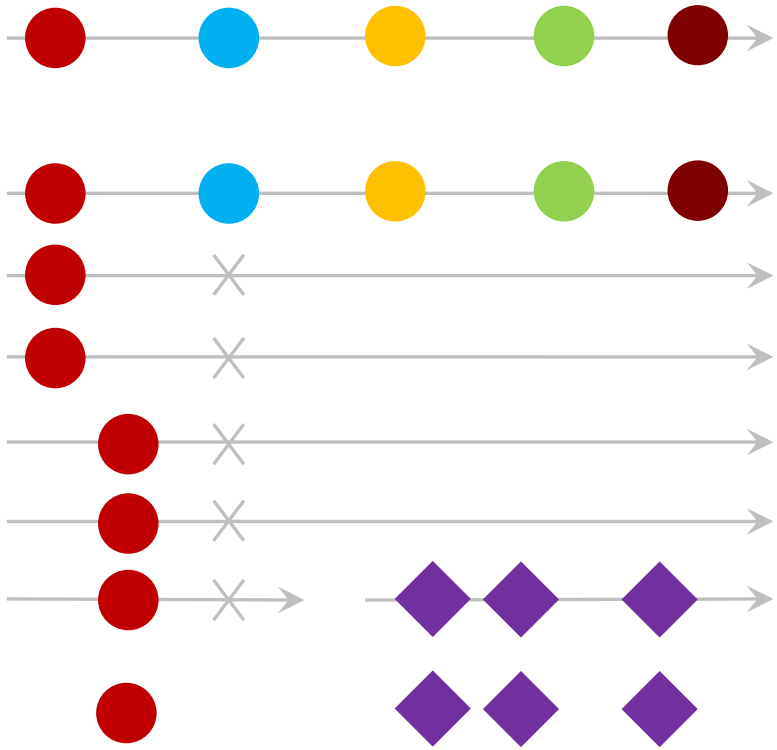
```
Rx.Observable.fromEvent(el, 'click')
  .pipe(
    .withLatestFrom(o1, o2, o3, ...)
    .filter(isFormValid)
    .map(createRequestObject)
    .mergeMap(sendRequestToServer)
    .map(processResponse)
    .retry(1)
  )
  .subscribe(
      response => { ... },
      error => { ... }
  )
```



ARS

Fehlerbehandlung mit catch()

```
Rx.Observable.fromEvent(el, 'click')
  .pipe(
    .withLatestFrom(o1, o2, o3, ...)
    .filter(isFormValid)
    .map(createRequestObject)
    .mergeMap(sendRequestToServer)
    .map(processResponse)
    .catchError(provideBackup)
  )
  .subscribe(
      response => { ... },
      error => { ... }
  )
```



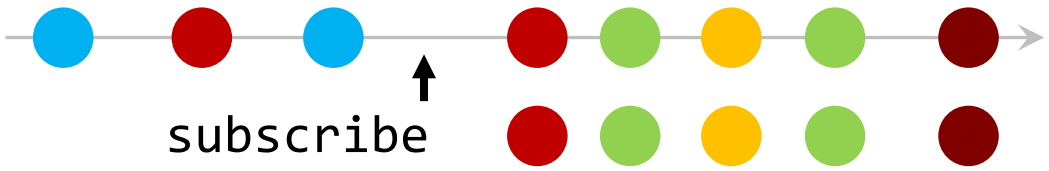
```
provideBackup(error: Error): Observable<any> {
  ...
}
```

ARS

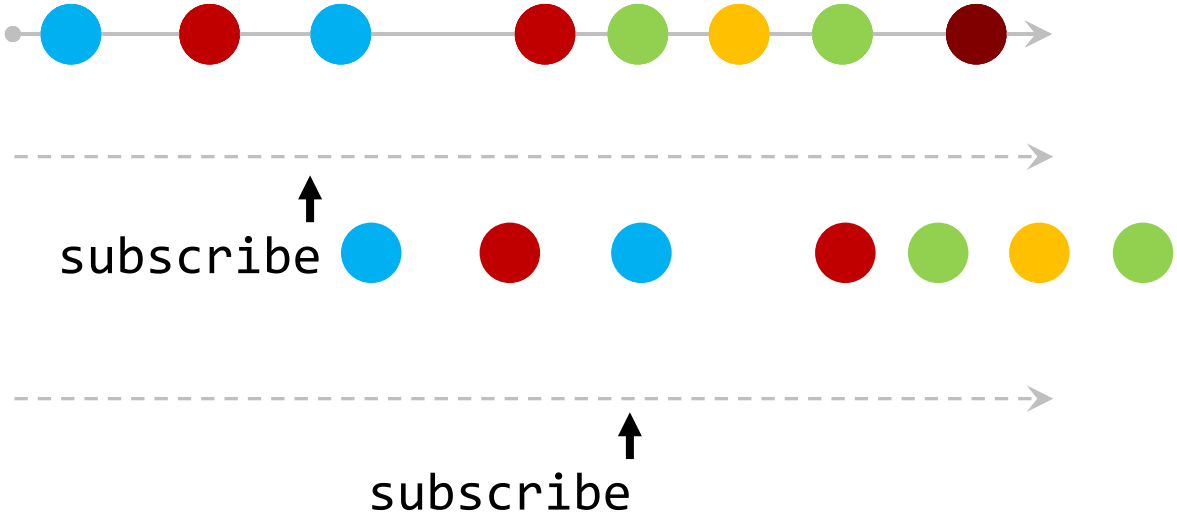
Hot & Cold Observables

Wann fangen Observables an, Events zu generieren?

Hot Observables generieren Events sobald die Observable erzeugt ist, und Subscriber erhalten nur zukünftige Events.

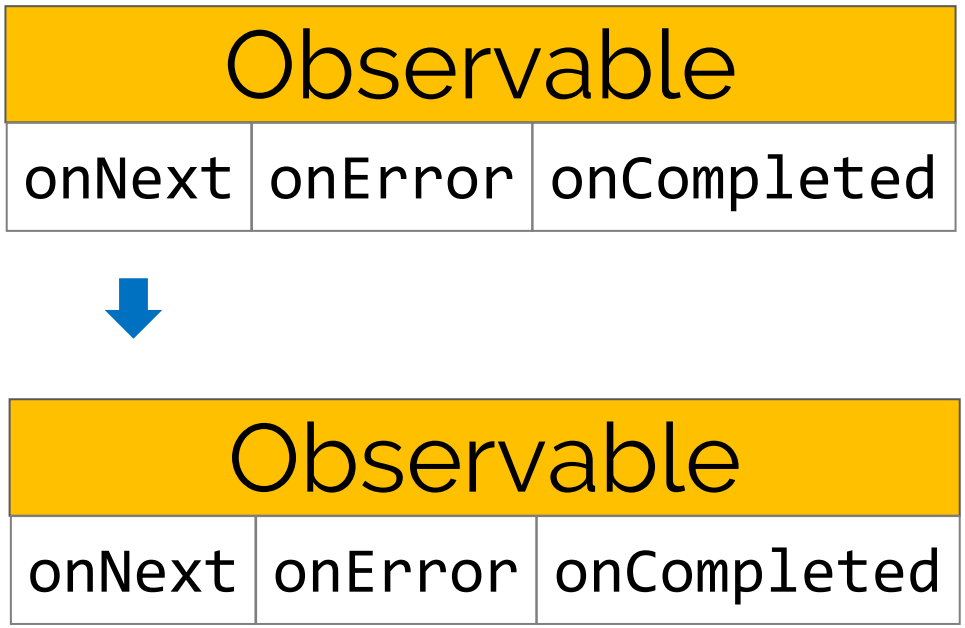


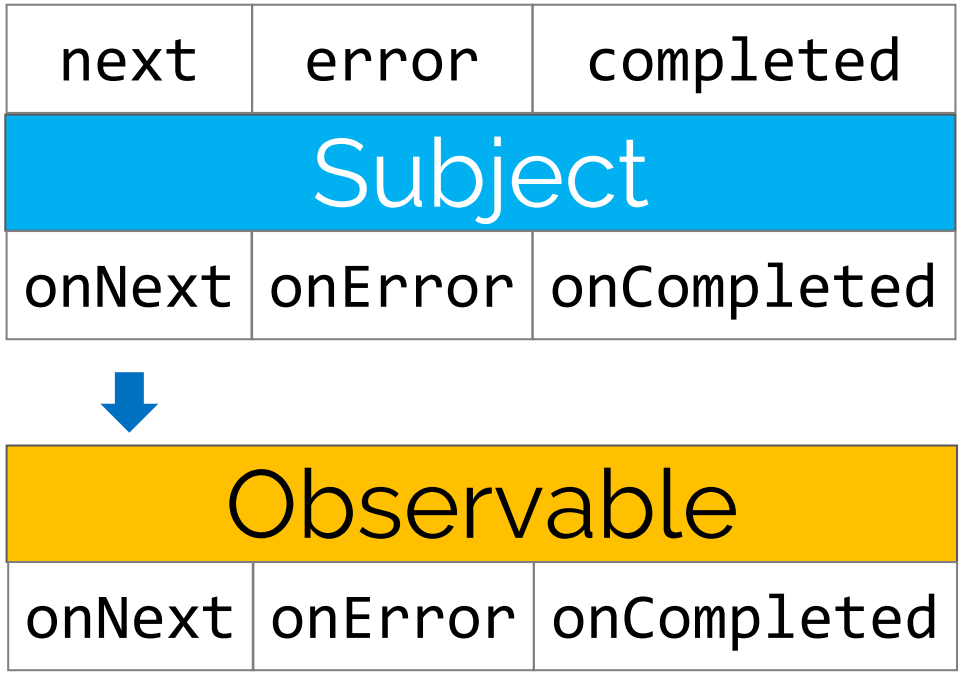
Cold Observables warten auf eine Subscription um die Events zu erzeugen. Daher bekommt jeder Subscriber alle erzeugten Werte.



ARS

Subjects



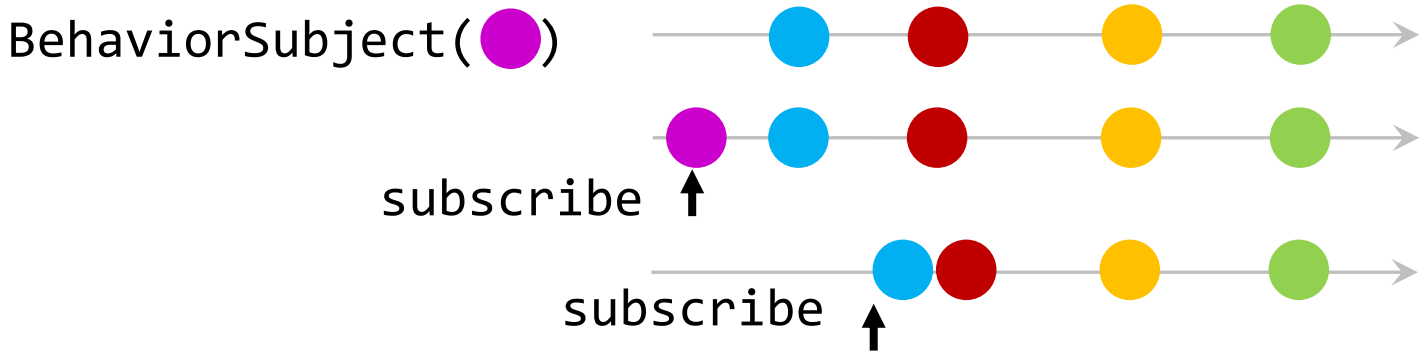


- Subjects haben eine API um Daten zu übergeben
- Dafür stellen Subjects drei Funktionen bereit: next(), error() und completed()
- Subjects bieten die gleichen Operatoren wie Observables an

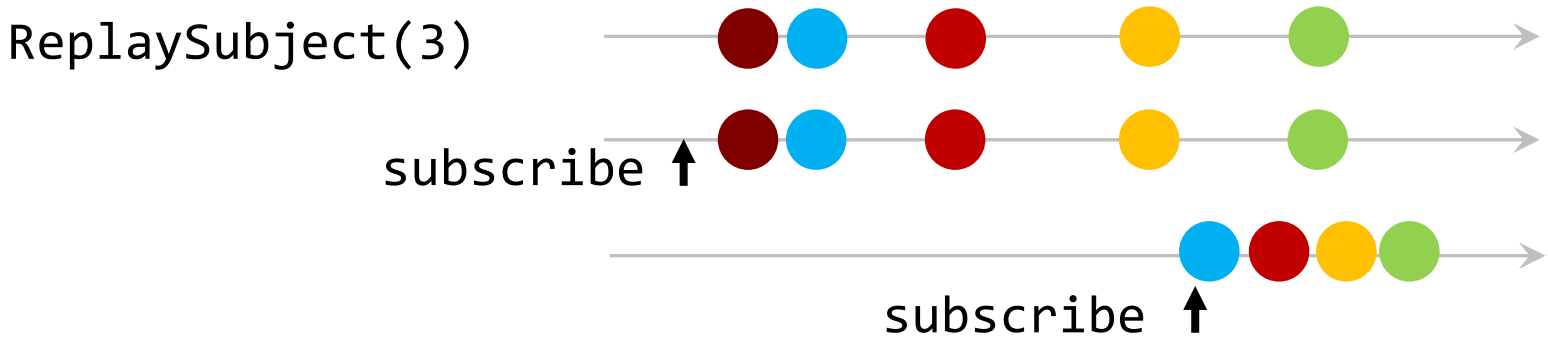
ARS

Arten von Subjects

BehaviorSubject — Puffert den letzten Wert und akzeptiert einen Defaultwert.



ReplaySubject — Puffert die letzten *n* Events, hat aber keinen Defaultwert



ARS

Zusammenfassung

Um was ging es in diesem Modul?

- Push/Pull Programmierung
- Rx Paradigmen kennenlernen
- Bearbeitung von Datenströmen
- Fehlerbehandlung im funktional reaktiven Programmieren
- Eigenschaften von ausgewählten Subjects

Wozu brauche ich das? Was werde ich damit machen?

- Code schreiben der stark auf Komposition anstatt auf Vererbung beruht
- Angular Events besser verstehen und anwenden

ARS

Kontrollfragen

- Was ist der grundlegende Unterschied zwischen push und pull Programmierung?
- Was ist eine Observable?
- Was machen Operatoren?
- Nennen Sie je ein Beispiel für eine Hot und Cold Observable. Was sind deren Unterschiede?
- Wie unterscheiden sich Observables von Subjects?