RxJs 2/4
ReactiveX for Javascript

1 OBSERVABLE ERZEUGEN 2
1.1 Visualizer 2
1.2 Array – from 2
1.2.1 Mit Visualizer 2
1.3 tap 3
1.3.1 Mit Visualizer 3
1.4 Werte 3
1.4.1 of 3
1.4.2 range 4
1.5 Timer 4
1.5.1 interval 4
1.5.2 timer 4
1.6 Events – fromEvent 5
1.7 http – from 5
1.8 create 6
2 OPERATOREN 7
2.1 filter 7
2.1.1 tap 7
2.2 map 8
2.3 skip/take 9
2.4 first/last/elementAt 9
2.5 distinct/distinctUntilChanged 9
2.6 max/min/count 10
2.7 Asynchron 10
2.8 Kombination 10
2.9 Eigener Operator 11
2.9.1 Parameter 12

1 Observable erzeugen

Da RxJs auf dem Umgang mit Observables basiert, muss man diese erzeugen können. Dazu gibt es unterschiedlichste Möglichkeiten.
In den folgenden Beispielen wird beim Observable immer mit dem erwähnten Observer mittels **subscribe**() registriert:

```
const observer = {
  next: value => console.log(`next: ${value}`),
  error: error => console.error(error),
  complete: () => console.log('Completed')
};
```

1.1 Visualizer

Bei der Verwendung des Visualizers werden (falls nicht explizit anders angegeben) zu Beginn immer folgende zwei Zeilen ausgeführt:

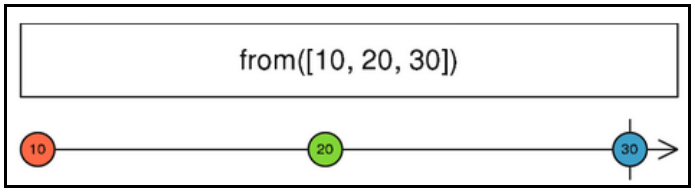
```
RxJsVisualizer.prepareCanvas(['A', 'B']);
RxJsVisualizer.startVisualize();
```

Folgende Einstellungen werden im Tutorial verwendet:

```
RxJsVisualizer.init({
  canvasId: 'canvas',
  logDivId: 'logs',
  blockHeight: 50,
  shapeSize: 20,
  maxPeriod: 10000,
  tickPeriod: 1000,
  centerShapes: false,
  symbolMap: symbols,
  addNavigationButtons: true,
  DEBUG: false
});
RxJsVisualizer.useRandomSymbolsForNumbers(100);
```

Die letzte Zeile ist dazu da, damit anstelle der Zahlen Marble-Symbole gezeichnet werden.

1.2 Array – from



Mit **Rx.from()** erzeugt man aus einem Javascript-Array ein Observable.

```
const array = [11, 22, 33];
const observable = Rx.from(array);
observable.subscribe(observer);
```

```
next: 11
next: 22
next: 33
Completed
```

Wie man sieht, werden die Werte sofort und ohne Verzögerung emittiert.

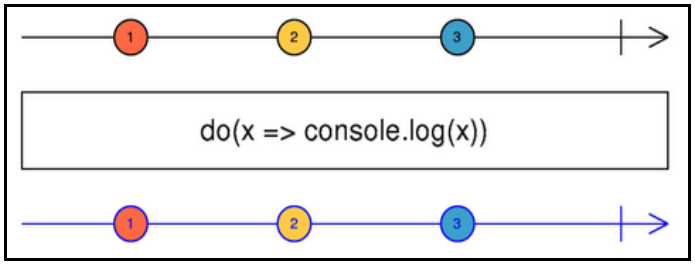
1.2.1 Mit Visualizer

Gibt man als Observer RxJsVisualizer.observerForLine(1) an, erfolgt die Ausgabe farbig direkt im Browser:

```
const array = [11, 22, 33];
const observable = Rx.from(array);
observable.subscribe(RxJsVisualizer.observerForLine(1));
```

```
17:54:30.494   11
17:54:30.494   22
17:54:30.496   33
17:54:30.497  Completed
```

1.3 tap



Mit `tap()` kann man Zwischenschritte im Ablauf kontrollieren. Damit kann man auch noch einmal veranschaulichen, dass

- ein Observable nichts macht, wenn sich kein Observer mit subscribe() registriert
- für jeden Observer ein „neues Laufbandsystem“ gebaut wird

```
tap 11
next: 11
tap 22
next: 22
tap 33
next: 33
Completed
tap 11
next Observer2: 11
tap 22
next Observer2: 22
tap 33
next Observer2: 33
Completed Observer2
```

```
const array = [11, 22, 33];
const observable = Rx.from(array)
  .pipe(tap(x => console.log(`tap ${x}`)));
observable.subscribe(observer);
observable.subscribe(observer2);
```

1.3.1 Mit Visualizer

Grafisch würde es so aussehen:

```
18:00:13.078  tap 11
18:00:13.079   11
18:00:13.080  tap 22
18:00:13.080   22
18:00:13.080  tap 33
18:00:13.081   33
18:00:13.081  Completed
18:00:13.082  tap 11
18:00:13.082   11
18:00:13.082  tap 22
18:00:13.084   22
18:00:13.084  tap 33
18:00:13.085   33
18:00:13.085  Completed
```

```
const array = [11, 22, 33];
const observable = Rx.from(array)
  .pipe(tap(x => RxJsVisualizer.writeToLine(0, `tap ${x}`)));
observable.subscribe(RxJsVisualizer.observerForLine(0));
observable.subscribe(RxJsVisualizer.observerForLine(1));
```

1.4 Werte

Werte können auch direkt angegeben werden.

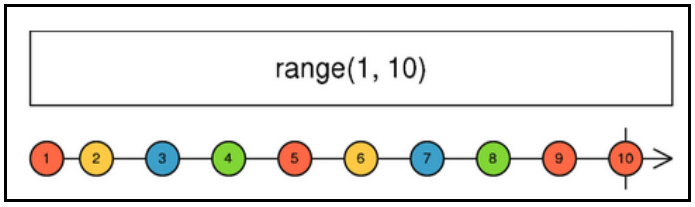
1.4.1 of

Sehr ähnlich zu `from()` funktioniert `of()`, bei dem man einfach die einzelnen Werte als Liste angibt.

```
18:01:53.344   11
18:01:53.346   22
18:01:53.347   33
18:01:53.348  Completed
```

```
RxJsVisualizer.prepareCanvas(['Val']);
Rx.of(11, 22, 33)
  .subscribe(RxJsVisualizer.observerForLine(0));
```

1.4.2 range



Die Funktion **`range()`** funktioniert sehr ähnlich, indem es beginnend bei einem Startwert eine Anzahl von Werten erzeugt:

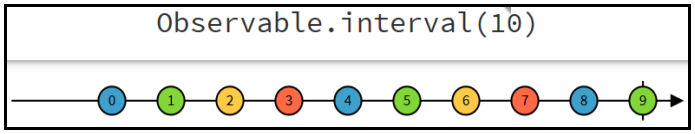
```
RxJsVisualizer.prepareCanvas(['Val']);
Rx.range(5, 3)
  .subscribe(RxJsVisualizer.observerForLine(0));
```

```
18:02:27.528   5
18:02:27.529   6
18:02:27.530   7
18:02:27.530  Completed
```

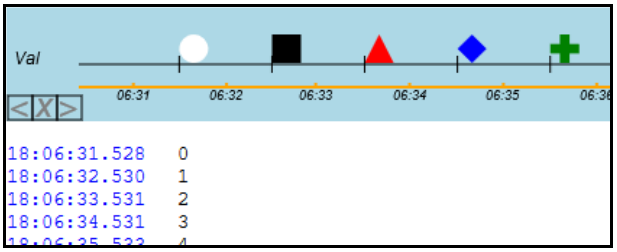
1.5 Timer

Man kann sich auch zyklisch Werte erzeugen lassen. Dafür gibt es zwei Funktionen.

1.5.1 interval



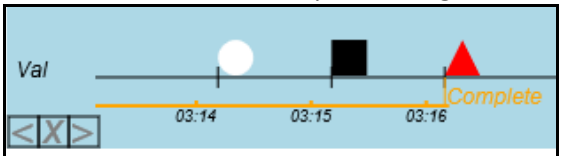
Die Funktion heißt **`interval`**() und erwartet als Parameter das Intervall in Millisekunden. Die Werte beginnen mit 0 und erhöhen sich immer um 1:




```
RxJsVisualizer.prepareCanvas(['Val']);
Rx.interval(1000)
  .subscribe(RxJsVisualizer.observerForLine(0));
```

```
18:06:31.528   0
18:06:32.530   1
18:06:33.531   2
18:06:34.531   3
```

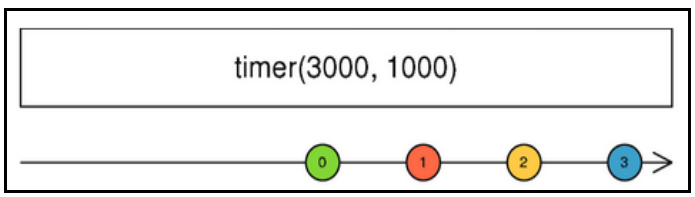
Man beachte, dass es sich dabei um ein Observable handelt, das nie endet.
Gibt man mit **`take`**() aber die Anzahl der Werte an, die man behandeln will, wird ein Completed ausgelöst:




```
RxJsVisualizer.prepareCanvas(['Val']);
Rx.interval(1000)
  .pipe(take(3))
  .subscribe(RxJsVisualizer.observerForLine(0));
```

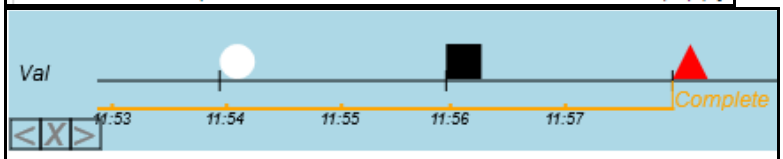
```
18:03:14.193   0
18:03:15.195   1
18:03:16.193   2
18:03:16.194  Completed
```

1.5.2 timer



`timer`() funktioniert ähnlich: der erste Parameter sind Millisekunden bis zum ersten Wert, der zweite ist dann die Verzögerung für jeden weiteren Wert:

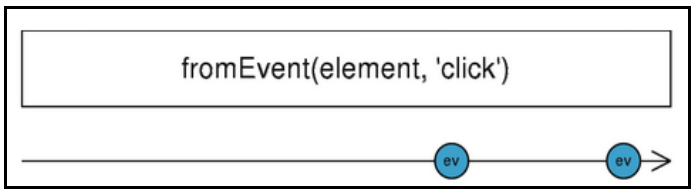
```
RxJsVisualizer.prepareCanvas(['Val']);
Rx.timer(1000, 2000)
  .pipe(take(3))
  .subscribe(RxJsVisualizer.observerForLine(0));
```




```
18:11:53.967   0
18:11:55.969   1
18:11:57.970   2
18:11:57.970  Completed
```

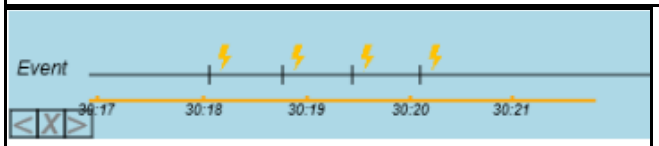
Fehlt der zweite Parameter, wird nach der Verzögerung ein Wert erzeugt und mit einem Completed abgeschlossen.

1.6 Events – fromEvent



Mit **fromEvent**() kann man aus einem beliebigen Event eines beliebigen HTML-Controls Observable erzeugen. Auch diese Observables sind offensichtlich endlos.

```
RxJsVisualizer.prepareCanvas(['Event']);
const btnProduce = document.querySelector('#btnClick');
Rx.fromEvent(btnProduce, 'click')
  .subscribe(RxJsVisualizer.observerForLine(0));
```

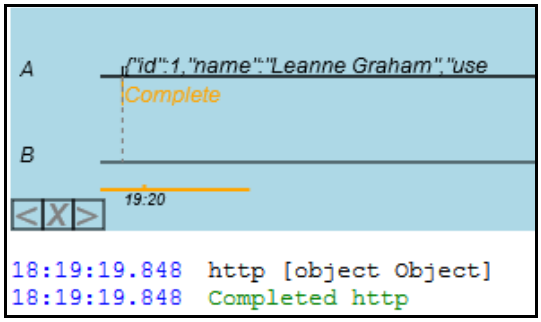



```
12:30:18.090   {"isTrusted":true}[MouseEvent]
12:30:18.807   {"isTrusted":true}[MouseEvent]
12:30:19.484   {"isTrusted":true}[MouseEvent]
12:30:20.148   {"isTrusted":true}[MouseEvent]
```

1.7 http – from

Daten aus einem WebService fallen ja asynchron an. D.h., die Daten stehen nicht gleich als Rückgabewert zur Verfügung, sondern erst in der Zukunft. Derartige asynchrone Szenarien kann man über Callbacks lösen, was aber zu unübersichtlichem Code führt („Callback hell“). Eine Möglichkeit ist die Lösung über Promises. Promises selbst werden hier nicht betrachtet, sondern nur die Möglichkeit, daraus ein Observable zu erzeugen.

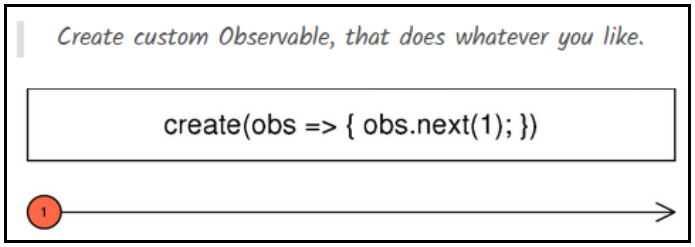
```
const getUsers = fetch('https://jsonplaceholder.typicode.com/users/1')
  .then(x => x.json());
Rx.from(getUsers)
  .pipe(draw(0, 'http', true, x => JSON.stringify(x).substr(0, 35)))
  .subscribe(x => console.log(JSON.stringify(x)));
```



{"id":1,"name":"Leanne Graham","username":"Bret","email":"Sincere@apr
Light","suite":"Apt. 556","city":"Gwenborough","zipcode":"92998-3874"
37.3159","lng":"81.1496"}},"phone":"1-770-736-8031 x56442","website":
{"name":"Romaguera-Crona","catchPhrase":"Multi-layered client-server
e-markets"}}

Wie man das z.B. mit einem Klick-Event verbindet siehe weiter unten.

1.8 create



Über die Methode **`create()`** kann man eine Funktion registrieren, die bei Subscription einen Observer als Parameter erhält, dem man dann mit **`next`**(), **`error`**() bzw. **`complete`**() Werte/Fehler emittieren lassen kann. Damit ist man also völlig flexibel.

Signatur:

```
public static create(onSubscription: function(observer: Observer): TeardownLogic): Observable
```

Als Beispiel soll auf Buttonklicks reagiert werden und eine der erwähnten Funktionen aufgerufen werden:

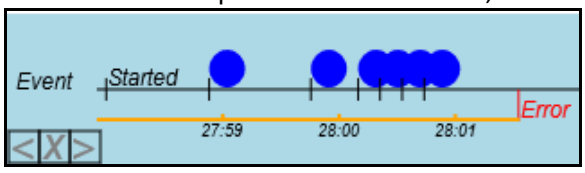
```
const btnNext = document.querySelector('#btnNext');
const btnError = document.querySelector('#btnError');
const btnComplete = document.querySelector('#btnComplete');
```

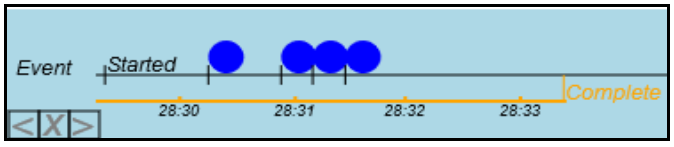
```
const subscription = Observable.create(obs => {
  obs.next('Started');
  btnNext.onclick = ev => obs.next(ev);
  btnError.onclick = _ => obs.error('Error!');
  btnComplete.onclick = _ => obs.complete();
}).subscribe(RxJsVisualizer.observerForLine(0));
```

Je nachdem, welche Buttons geklickt werden, sieht die Ausgabe aus.

Zur Wiederholung:

- nach Error ist das Observable beendet. Es wird kein Complete ausgelöst
- nach Complete kann weder next, noch error ausgelöst werden.

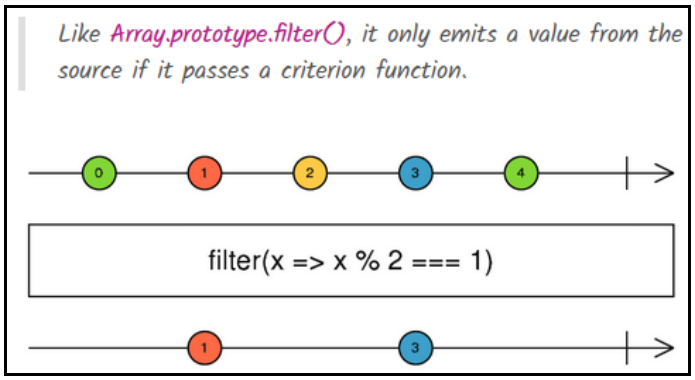




2 Operatoren

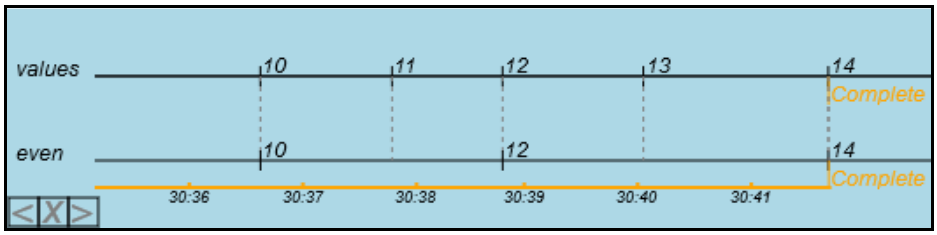
Bevor man sich auf ein Observable mit subscribe() registriert, kann man die Objekte auf der Timeline noch durch viele Operatoren verändern. Die Anwendung und auch die Namen sind sehr ähnlich zu LINQ. Sie werden mit **`pipe()`** der Stream-Pipeline hinzugefügt.

2.1 filter



Entspricht dem Where von LINQ.

```
RxJsVisualizer.prepareCanvas(['values', 'even']);
const obs = RxJsVisualizer.createStreamFromArraySequence([10, 11, 12, 13, 14]);
obs.subscribe(RxJsVisualizer.observerForLine(0, '', true));
obs
  .pipe(
    filter(x => x % 2 === 0),
  )
  .subscribe(RxJsVisualizer.observerForLine(1, 'even', true));
```

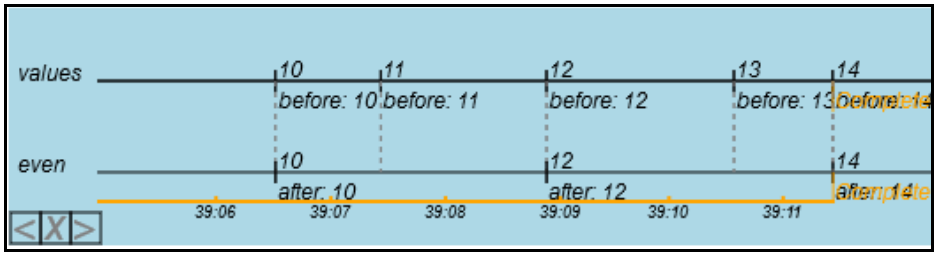



```
18:30:36.651   10
18:30:36.653  even 10
18:30:37.824   11
18:30:38.807   12
18:30:38.809  even 12
18:30:40.060   13
18:30:41.708   14
18:30:41.709  even 14
18:30:41.711  Completed
18:30:41.712  Completed even
```

2.1.1 tap

In diesem Zusammenhang sei noch einmal auf die Funktion **`tap`**() hingewiesen, die es erleichtert, etwaige Fehler in den verschiedenen Operator-Sequenzen zu finden. Das ist oft die einzige Möglichkeit, RxJs-Konstrukte zu debuggen.

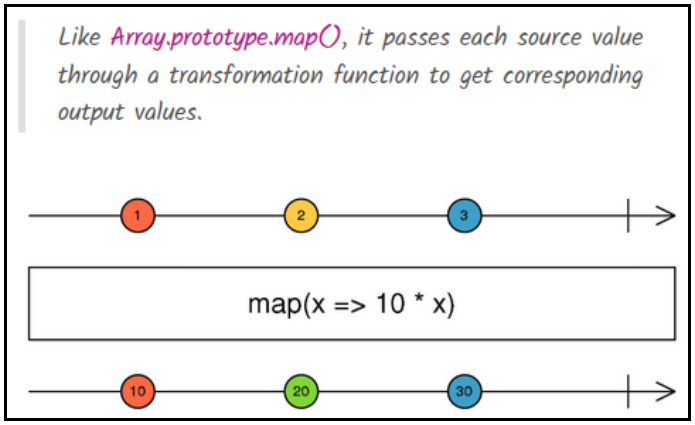
```
obs
  .pipe(
    tap(x => RxJsVisualizer.writeToLine(0, `before: ${x}`)),
    filter(x => x % 2 === 0),
    tap(x => RxJsVisualizer.writeToLine(1, `after: ${x}`))
  )
  .subscribe(RxJsVisualizer.observerForLine(1, 'even', true));
```




```
18:39:06.531   10
18:39:06.531  before: 10
18:39:06.532  after: 10
18:39:06.533  even 10
18:39:07.461   11
18:39:07.463  before: 11
18:39:08.926   12
18:39:08.927  before: 12
18:39:08.927  after: 12
18:39:08.928  even 12
18:39:10.581   13
18:39:10.582  before: 13
18:39:11.453   14
18:39:11.455  before: 14
18:39:11.456  after: 14
18:39:11.457  even 14
18:39:11.457  Completed
18:39:11.459  Completed even
```

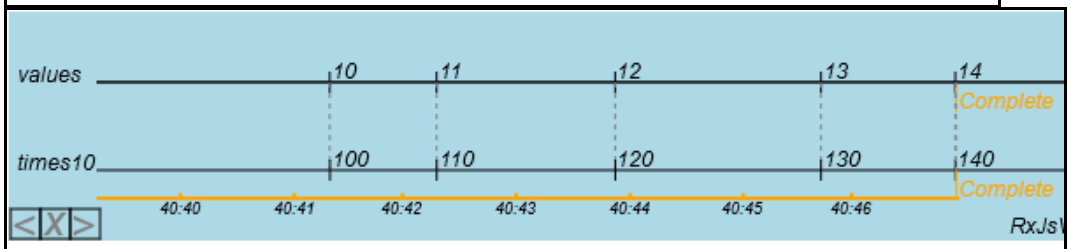
2.2 map

> *Like Array.prototype.map(), it passes each source value through a transformation function to get corresponding output values.*



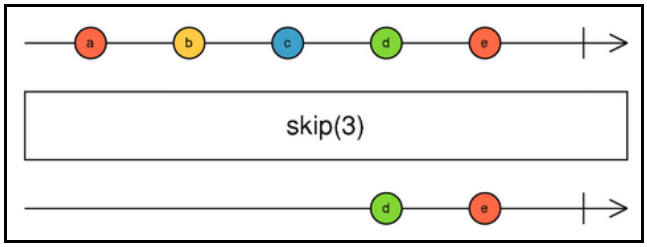
Keine Überraschungen - entspricht dem Select von LINQ.

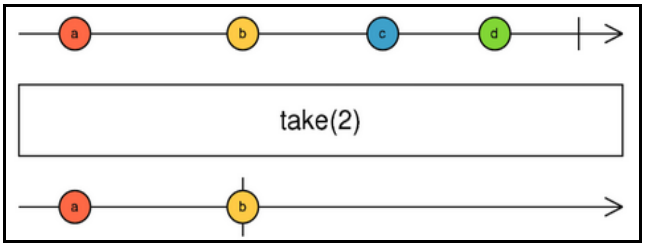
```
RxJsVisualizer.prepareCanvas(['values', 'times10']);
RxJsVisualizer.createStreamFromArraySequence([10, 11, 12, 13, 14])
  .pipe(
    draw(0, '', true),
    map(x => x * 10)
  )
  .subscribe(RxJsVisualizer.observerForLine(1, '*10:', true));
```




```
18:40:41.361   10
18:40:41.363  *10: 100
18:40:42.313   11
18:40:42.314  *10: 110
18:40:43.902   12
18:40:43.903  *10: 120
18:40:45.733   13
18:40:45.734  *10: 130
18:40:46.928   14
18:40:46.931  *10: 140
18:40:46.936  Completed
18:40:46.938  Completed *10:
```

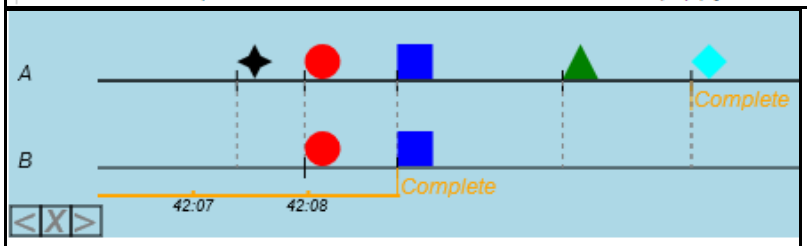
2.3 skip/take





Verhalten sich ebenfalls wie erwartet.

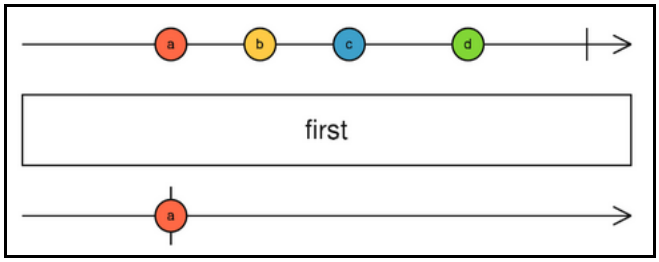
```
const obs = RxJsVisualizer.createStreamFromArraySequence([10, 11, 12, 13, 14]);
obs.subscribe(RxJsVisualizer.observerForLine(0));
obs
  .pipe(
    skip(1),
    take(2)
  )
  .subscribe(RxJsVisualizer.observerForLine(1));
```

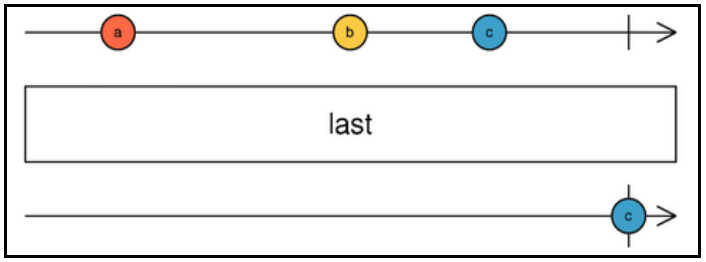



```
18:42:07.416   10
18:42:08.013   11
18:42:08.014   11
18:42:08.825   12
18:42:08.827   12
18:42:08.828  Completed
18:42:10.280   13
18:42:11.411   14
18:42:11.411  Completed
```

Daneben gibt es mit skipLast und takeLast auch Varianten, die sich auf das Ende der Elemente beziehen bzw. die von LINQ gekannten skipUntil, skipWhile, takeUntil, takeWhile.

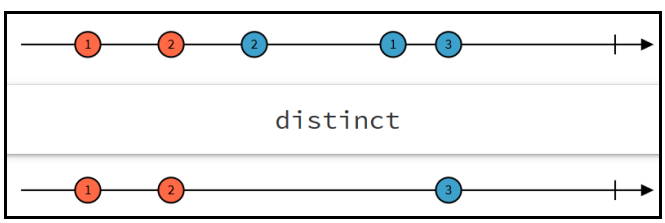
2.4 first/last/elementAt

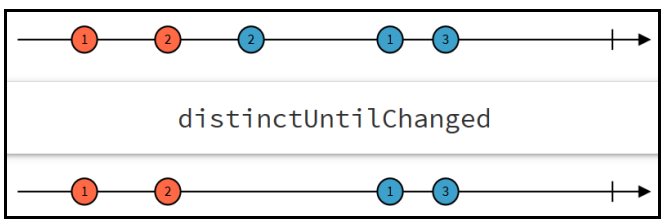




Der Name ist Programm…

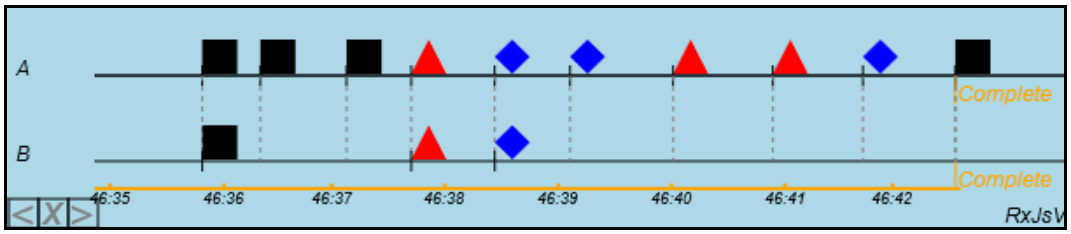
2.5 distinct/distinctUntilChanged





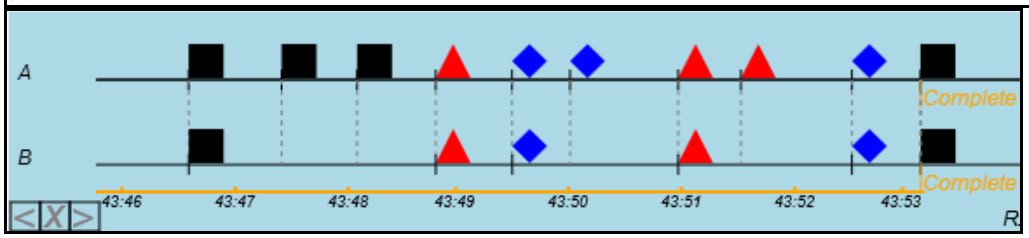
```
RxJsVisualizer.createStreamFromArraySequence([1, 1, 1, 2, 3, 3, 2, 2, 3, 1], 500, 1000)
  .pipe(
    draw(0),
    distinct()
  )
  .subscribe(RxJsVisualizer.observerForLine(1));
```

Funktioniert wieder wie bei LINQ:



Zusätzlich gibt es noch die Funktion distinctUntilChanged, die anhand des Marble-Diagramms bzw. des Beispiels verständlich sein sollte.

```
RxJsVisualizer.createStreamFromArraySequence([1, 1, 1, 2, 3, 3, 2, 2, 3, 1], 500, 1000)
  .pipe(
    draw(0),
    distinctUntilChanged()
  )
  .subscribe(RxJsVisualizer.observerForLine(1));
```



2.6 max/min/count

Die Funktionen sum und average gibt es nicht, die muss man z.B. über reduce lösen.

```
const observable = Rx.of(5, 12, 4);
observable.pipe(max()).subscribe(x => console.log(`max=${x}`));
observable.pipe(min()).subscribe(x => console.log(`min=${x}`));
observable.pipe(count()).subscribe(x => console.log(`count=${x}`));
```

```
18:27:07.530: max=12
18:27:07.534: min=4
18:27:07.536: count=3
```

Hinweis: diese Funktionen sind erst verfügbar, wenn das Objservable completed ist.

2.7 Asynchron

Man muss aber bei allen Operatoren immer bedenken, dass der Ablauf asynchron ist.
D.h. folgende Annahme ist falsch:

```
let maxNr = observable.max();
console.log(`maxNr=${maxNr}`);
```

Sondern max() liefert **wiederum ein Observable**, auf das man subscriben muss, um die Erzeugung des Wertes auszulösen!

```
observable.max().subscribe(maxNr => console.log(`maxNr=${maxNr}`));
```

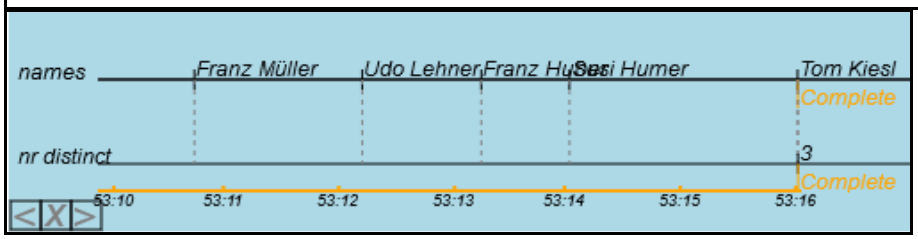
Das ist auch ein wesentlicher Unterschied zu LINQ, wo das Ergebnis synchron berechnet wird und direkt den Wert liefert:

```
var maxNr = numbers.Max(); //ok in LINQ
```

2.8 Kombination

Operatoren kann man wie erwartet in der Pipe kombinieren, man muss nur wie gesagt daran denken, dass das Ergebnis erst bei subscribe zur Verfügung steht.

```
RxJsVisualizer.prepareCanvas(['names', 'nr distinct']);
RxJsVisualizer.createStreamFromArraySequence([
    'Franz Müller',
    'Udo Lehner',
    'Franz Huber',
    'Susi Humer',
    'Tom Kiesl'
  ])
  .pipe(
    draw(0),
    filter(x => x.endsWith('er')),
    map(x => x.split(' ')),
    map(x => x[0]),
    distinct(),
    count()
  )
  .subscribe(RxJsVisualizer.observerForLine(1, 'distinct', true));
```



2.9 Eigener Operator

Kommt man mit den vorhandenen Operatoren nicht aus, kann man relativ einfach eigene Funktionen in die Pipe einhängen.

```
Rx.of('Franz Müller', 'Udo Lehner', 'Franz Huber', 'Susi Humer', 'Tom Kiesl')
  .pipe(
    myOperator()
  )
  .subscribe(observer);
```

Ein Operator ist eine Funktion, die wiederum eine Funktion zurückgibt. Als Parameter erhält die Funktion ein Observable, der Rückgabewert muss ebenfalls ein Observable sein.
Der Operator **`myOperator`** könnte dann so aussehen:

```
function myOperator() {
  return srcObservable => {
    return new Observable(subscriber => {
      srcObservable.subscribe(
        data => {
          const items = data.split(' ');
          const first = items[0];
          const last = items[1].toUpperCase();
          if (first.length > 3) subscriber.next(`${last} ${first.substr(0,3)}.`);
        },
        err => subscriber.error(err),
        () => subscriber.complete()
      );
    });
  };
}
```

Man hängt sich also als Listener auf das Eingangs-Observable (**`srcObservable`**) und schreibt seine Ergebnisse auf ein neues Observable. Dieses wird auch zurückgegeben.

```
10:43:09.473  next: MÜLLER Fra.
10:43:09.475  next: HUBER Fra.
10:43:09.476  next: HUMER Sus.
10:43:09.477  Completed
```

Man nimmt also alle anfallenden Werte der Pipe und legt diese in anderer Form (oder auch gar nicht) wieder auf die Pipe.

2.9.1 Parameter

Mit einem Parameter funktioniert es praktisch genauso.

```
function myOperator(nrChars) {
  return srcObservable => {
    return new Observable(subscriber => {
      srcObservable.subscribe(
        data => {
          const items = data.split(' ');
          const first = items[0];
          const last = items[1].toUpperCase();
          if (first.length > nrChars) subscriber.next(`${last} ${first.substr(0,nrChars)}.`);
        },
        err => subscriber.error(err),
        () => subscriber.complete()
      );
    });
  };
}
```

```
10:47:15.385  next: MÜLLER Fr.
10:47:15.387  next: LEHNER Ud.
10:47:15.388  next: HUBER Fr.
10:47:15.388  next: HUMER Su.
10:47:15.389  next: KIESL To.
10:47:15.389  Completed
```

```
Rx.of('Franz Müller', 'Udo Lehner', 'Franz H
  .pipe(
    myOperator(2)
  )
  .subscribe(observer);
```