RxJs 4/4
ReactiveX for Javascript

1 COLD/HOT/WARM/CONNECTABLE 2
1.1 cold 2
1.2 hot: publish + connect 3
1.3 warm: publish+refCount bzw. share 5
1.3.1 share 6
1.4 Angular HttpClient 6
2 SUBJECT VS. OBSERVABLE 8
2.1 BehaviorSubject 8
2.2 ReplaySubject 9
2.2.1 Replay 10

1 cold/hot/warm/connectable

Aufpassen muss man immer, ob ein Observable hot oder cold ist. Ein RxJs-Observable verhält sich meist nicht so, wie man das von einem Observer-Pattern gewohnt ist.
Die Definition auf der Homepage lautet:
*„**Cold observables start running upon subscription, i.e., the observable sequence only starts pushing values to the observers when Subscribe is called. (…) This is different from hot observables such as mouse move events or stock tickers which are already producing values even before a subscription is active.**"*
Das Problem ist, dass das meist nicht direkt erkennbar ist.

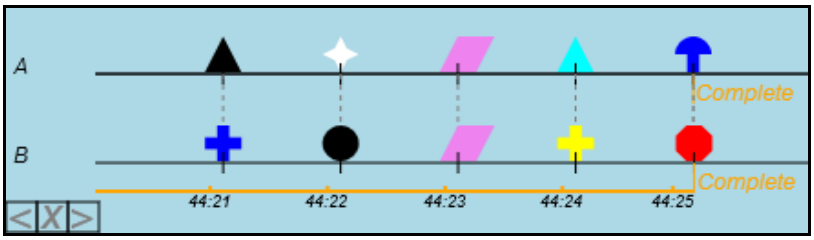
1.1 cold

Man würde erwarten, dass bei mehreren Observern sich diese einfach den gemeinsamen Wert teilen:

```
const observable = Rx.interval(1000)
  .pipe(take(5), map(_ => new Date().getTime() % 1000));
observable.subscribe(RxJsVisualizer.observerForLine(0, 'subscriber A'));
observable.subscribe(RxJsVisualizer.observerForLine(1, 'subscriber B'));
```

Das ist aber nicht so, wie man an den Werten des Timestamps (sie werden auf Millisekunden gemappt) erkennt:

```
09:44:21.147  subscriber A 146
09:44:21.148  subscriber B 148
09:44:22.155  subscriber A 154
09:44:22.155  subscriber B 155
09:44:23.160  subscriber A 160
09:44:23.162  subscriber B 160
09:44:24.169  subscriber A 168
09:44:24.170  subscriber B 170
09:44:25.176  subscriber A 175
09:44:25.179  Completed subscriber A
09:44:25.183  subscriber B 183
09:44:25.184  Completed subscriber B
```

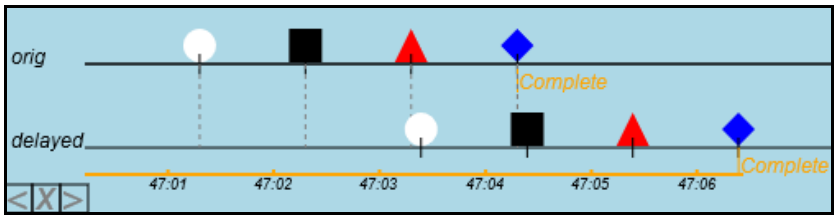


Jede Subscription erzeugt also eine komplett neue, unabhängige Förderbandstruktur.
Noch besser sieht man das, wenn man den zweiten Observer verzögert startet:

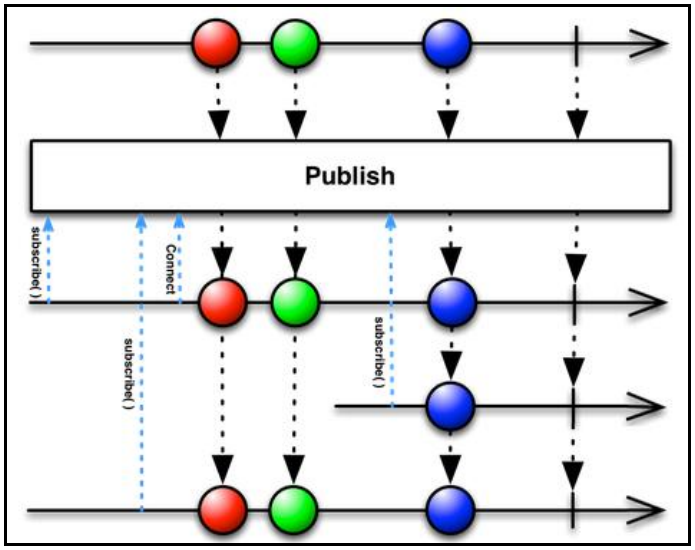
```
const observable = Rx.interval(1000)
  .pipe(take(4));
observable.subscribe(RxJsVisualizer.observerForLine(0, 'subscriber A'));
setTimeout(() => observable.subscribe(RxJsVisualizer.observerForLine(1, 'subscriber B')), 2100);
```

Dann bekommt der zweite Observer zwar verzögert, aber trotzdem alle Werte vollständig:

```
09:47:01.314  subscriber A 0
09:47:02.321  subscriber A 1
09:47:03.328  subscriber A 2
09:47:03.422  subscriber B 0
09:47:04.338  subscriber A 3
09:47:04.340  Completed subscriber A
09:47:04.432  subscriber B 1
09:47:05.437  subscriber B 2
09:47:06.442  subscriber B 3
09:47:06.444  Completed subscriber B
```



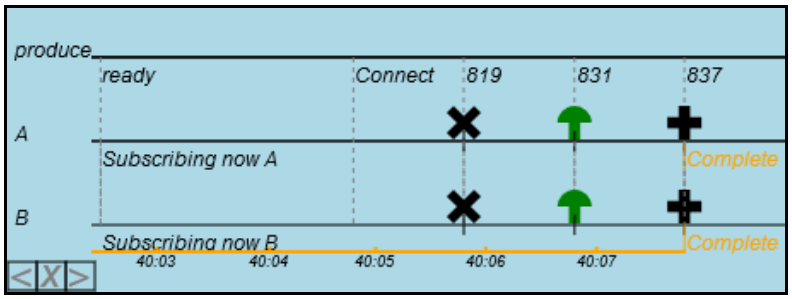
1.2 hot: publish + connect



Ein **`ConnectableObservable`** wird durch **`publish`**() erzeugt und verhält sich praktisch wie ein gewöhnliches Observable, jedoch beginnt die Produktion erst **dann (und nur dann)**, wenn **`connect`**() aufgerufen wurde.

```
RxJsVisualizer.prepareCanvas(['produce', 'A', 'B']);
const observable = Rx.interval(1000).pipe(
  take(3),
  map(_ => new Date().getTime() % 1000),
  tap(x => RxJsVisualizer.writeToLine(0, `${x}`))
);
RxJsVisualizer.writeToLine(0, 'ready');
const connectable = observable.pipe(publish());
RxJsVisualizer.writeToLine(1, 'Subscribing now A');
connectable.subscribe(RxJsVisualizer.observerForLine(1, 'subscriber A'));
RxJsVisualizer.writeToLine(2, 'Subscribing now B');
connectable.subscribe(RxJsVisualizer.observerForLine(2, 'subscriber B'));
setTimeout(_ => {
  RxJsVisualizer.writeToLine(0, 'Connect');
  connectable.connect();
}, 2300);
```

```
10:40:02.498  ready
10:40:02.499  Subscribing now A
10:40:02.500  Subscribing now B
10:40:04.811  Connect
10:40:05.819  819
10:40:05.820  subscriber A 819
10:40:05.821  subscriber B 819
10:40:06.832  831
10:40:06.833  subscriber A 831
10:40:06.834  subscriber B 831
10:40:07.837  837
10:40:07.838  subscriber A 837
10:40:07.838  subscriber B 837
10:40:07.839  Completed subscriber A
10:40:07.839  Completed subscriber B
```



Wie man sieht, hat publish() zusätzlich noch die Funktion, dass es mehrere Subscriptions aufnimmt und sich selbst als einzige Subscription beim Observable anmeldet. Das Connectable Observable fungiert somit als **Proxy**, das vom darunterliegenden Observable Werte erhält und diese an alle Subscriber verteilt.

Wichtig: würde **`connect`**() fehlen, würden keine Werte produziert!

Das heißt aber auch, dass die Produktion beginnt, auch wenn sich beim Connectable Observable niemand anmeldet:

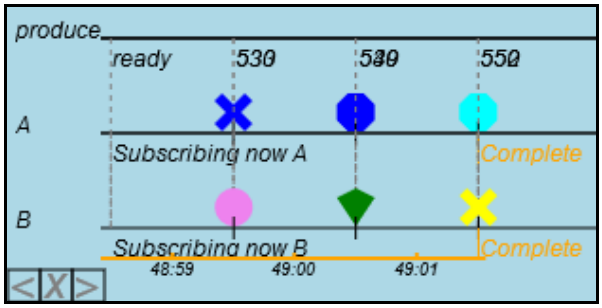
```
const observable = Rx.interval(1000).pipe(
  take(3),
  map(_ => new Date().getTime() % 1000),
  tap(x => RxJsVisualizer.writeToLine(0, `${x}`))
);
RxJsVisualizer.writeToLine(0, 'ready');
const connectable = observable.pipe(publish());
setTimeout(_ => {
  RxJsVisualizer.writeToLine(0, 'Connect');
  connectable.connect();
}, 2300);
```

```
10:46:21.586  ready
10:46:23.898  Connect
10:46:24.905  903
10:46:25.917  916
10:46:26.920  919
```

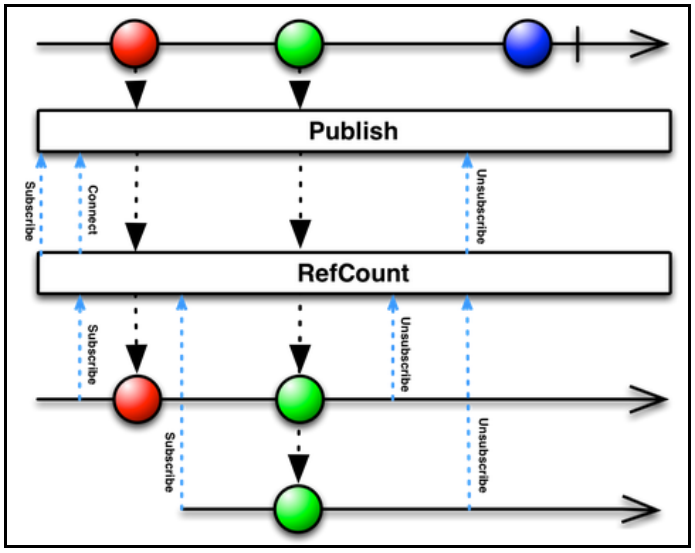
Im Vergleich noch einmal die Version ohne publish()/connect():

```
RxJsVisualizer.prepareCanvas(['produce', 'A', 'B']);
const observable = Rx.interval(1000).pipe(
  take(3),
  map(_ => new Date().getTime() % 1000),
  tap(x => RxJsVisualizer.writeToLine(0, `${x}`))
);
RxJsVisualizer.writeToLine(0, 'ready');
RxJsVisualizer.writeToLine(1, 'Subscribing now A');
observable.subscribe(RxJsVisualizer.observerForLine(1, 'subscriber A'));
RxJsVisualizer.writeToLine(2, 'Subscribing now B');
observable.subscribe(RxJsVisualizer.observerForLine(2, 'subscriber B'));
```

```
10:48:58.519  ready
10:48:58.519  Subscribing now A
10:48:58.520  Subscribing now B
10:48:59.530  530
10:48:59.532  subscriber A 530
10:48:59.533  533
10:48:59.533  subscriber B 533
10:49:00.540  539
10:49:00.540  subscriber A 539
10:49:00.540  540
10:49:00.541  subscriber B 540
10:49:01.550  550
10:49:01.551  subscriber A 550
10:49:01.552  Completed subscriber A
10:49:01.552  552
10:49:01.553  subscriber B 552
10:49:01.553  Completed subscriber B
```



1.3 warm: publish+refCount bzw. share

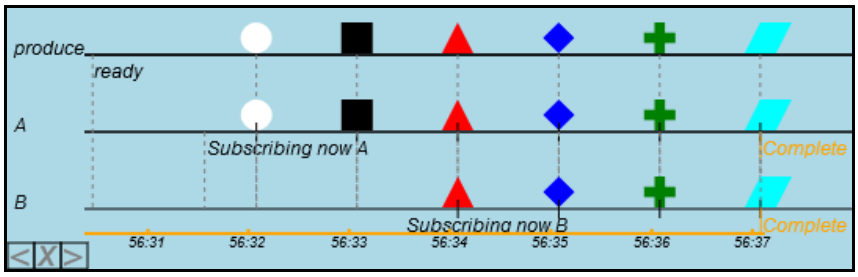


Mit **refCount**() kann man das connect und unsubscribe noch weiter automatisieren: der erste Subscriber bewirkt automatisch ein **connect**(), ist aber kein Subscriber mehr registriert, wird automatisch ein **unsubscribe**() durchgeführt.
Daher kommt auch der Name: es wird ein **reference count** mitgeführt, es ist also immer die Anzahl der Listener bekannt. Aus Sicht des zugrundeliegenden Observables gibt es aber nur einen (das ist aber aufgrund von publish() so).
Damit verhält sich das Observable am ehesten so, wie man sich das bei einem Observer-Pattern vermutlich erwarten würde.
Man kann es sich auch so vorstellen: Bei einem Live-Konzert beginnt die Band zu spielen, wenn der erste Besucher kommt und hört auf, wenn der letzte Besucher geht.

```
RxJsVisualizer.prepareCanvas(['produce', 'A', 'B']);
const observable = Rx.timer(500, 1000)
  .pipe(
    take(6),
    tap(x => RxJsVisualizer.writeToLine(0, `${x}`)),
    publish(),
    refCount(),
  );
RxJsVisualizer.writeToLine(0, 'ready');
let subA, subB;
setTimeout(_ => {
  RxJsVisualizer.writeToLine(1, 'Subscribing now A');
  subA = observable.subscribe(RxJsVisualizer.observerForLine(1, 'subscriber A'));
}, 1100);
setTimeout(_ => {
  RxJsVisualizer.writeToLine(2, 'Subscribing now B');
  observable.subscribe(RxJsVisualizer.observerForLine(2, 'subscriber B'));
}, 3100);
```

Jetzt bekommt der später startende SubscriberB auch die aktuellen Werte (und beginnt nicht mit einer „eigenen Produktion“):

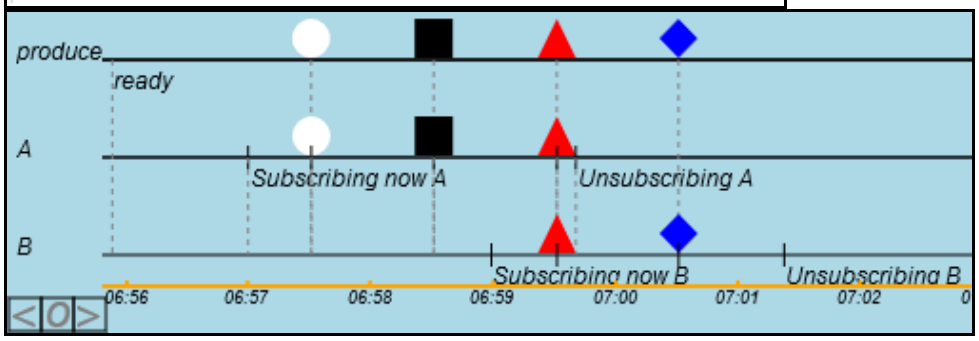
```
10:56:30.463  ready
10:56:31.579  Subscribing now A
10:56:32.093  0
10:56:32.094  subscriber A 0
10:56:33.096  1
10:56:33.097  subscriber A 1
10:56:33.569  Subscribing now B
10:56:34.100  2
10:56:34.100  subscriber A 2
10:56:34.102  subscriber B 2
10:56:35.106  3
10:56:35.108  subscriber A 3
10:56:35.108  subscriber B 3
10:56:36.112  4
10:56:36.113  subscriber A 4
10:56:36.113  subscriber B 4
10:56:37.114  5
10:56:37.115  subscriber A 5
10:56:37.116  subscriber B 5
10:56:37.118  Completed subscriber A
10:56:37.119  Completed subscriber B
```



Es ist aber nach wie vor so, dass der erste Subscriber das Observable startet. Registriert sich SubscriberA erst nach etwas mehr als einer Sekunde, bekommt er trotzdem den Wert 0, weil ja er erst das Observable startet:

Und so sieht es aus, wenn man sich mit unsubscribe() wieder abmeldet:

```
setTimeout(_ => {
  RxJsVisualizer.writeToLine(1, 'Unsubscribing A');
  subA.unsubscribe();
 }, 3800);
 setTimeout(_ => {
  RxJsVisualizer.writeToLine(2, 'Unsubscribing B');
  subB.unsubscribe();
 }, 5500);
```



1.3.1 share

Der Aufruf von **share**() bewirkt genau dasselbe:

```
let observable = Rx.Observable.interval(1000)
    .take(5)
    // .publish()
    // .refCount();
    .share();
```

1.4 Angular HttpClient

Der HttpClient von Angular ist ein cold Observable!!!!
Daher bewirken mehrfache subscribe() ein mehrfaches Absenden eines Requests. Bei einem GET-Request ist das in Bezug auf Performance schlecht, aber nicht wirklich problematisch. Bei einem POST ist das aber fatal, weil man dann Daten mehrfach in die Datenbank einträgt.
Also immer Vorsicht, wenn man versucht ist, derartigen Code zu schreiben:

```
getProducts(): Observable<Product[]> {
  const url = `${this.baseUrl}${this.urlProducts}`;
  console.log(`Services.getProducts - url=${url}`);
  const observable = this.http.get<Product[]>(url);
  observable.subscribe(x => console.log(`${x.length} products received`));
  return observable;
}
```

In diesem Fall immer explizit ein warm observable erzeugen:

```
getProducts(): Observable<Product[]> {
  const url = `${this.baseUrl}${this.urlProducts}`;
  console.log(`Services.getProducts - url=${url}`);
  const observable = this.http.get<Product[]>(url)
    .pipe(share());
  observable.subscribe(x => console.log(`${x.length} products received`));
  return observable;
}
```

2 Subject vs. Observable

Wie bereits erwähnt ist ein Subject sowohl ein **Observable** als auch ein **Observer**. Dadurch kann ein Subject Werte emittieren und andere Observer benachrichtigen. Sehr oft möchte man aber verhindern, dass auch an anderen Stellen Werte emittiert werden können. Das erreicht man, indem aus dem Subject mit der Methode **asObservable**() ein Observable zurückgibt:

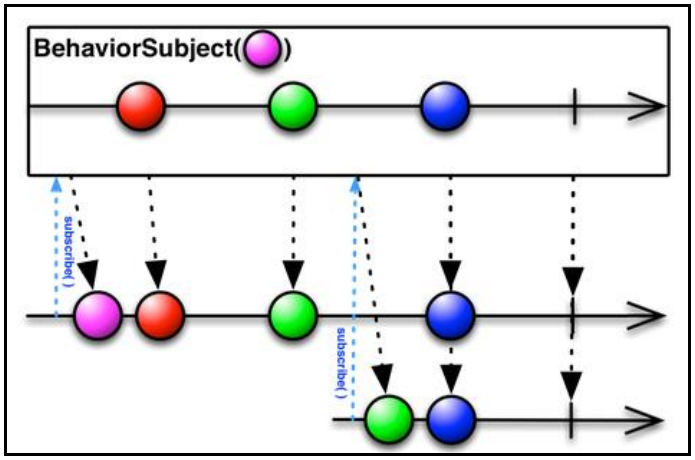
```
function getObservable() {
  const btnNext = document.querySelector('#btnNext');
  const btnError = document.querySelector('#btnError');
  const btnComplete = document.querySelector('#btnComplete');
  const subject = new Subject();
  subject.next('Started');
  setInterval(() => subject.next(new Date().getSeconds()), 1000);
  btnNext.onclick = ev => subject.next(ev);
  btnError.onclick = _ => subject.error('Error!');
  btnComplete.onclick = _ => subject.complete();
  return subject.asObservable();
}
```

Das hat zur Folge, dass „von außen“ keine Werte mehr emittiert werden können. Das sollte man so einsetzen, wie man in C# bei einer Klasse auch die Variablen **private** deklariert und somit schützt und nur über **public** Setter Zugriff darauf gewährt:

```
const btnUnsubscribe = document.querySelector('#btnUnsubscribe');
const observable = getObservable();
observable.next('xxx');
const subscription = observable.subscribe(RxJsVisualizer.observerForLine(0));
btnUnsubscribe.onclick = _ => subscription.unsubscribe();
```

```
TypeError: observable.next is not a function
```

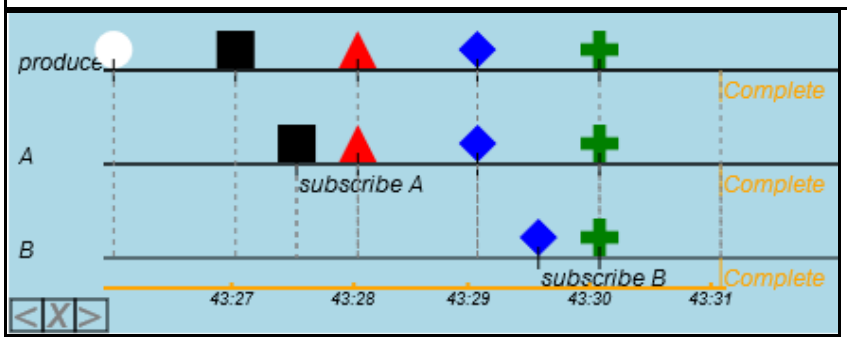
2.1 BehaviorSubject



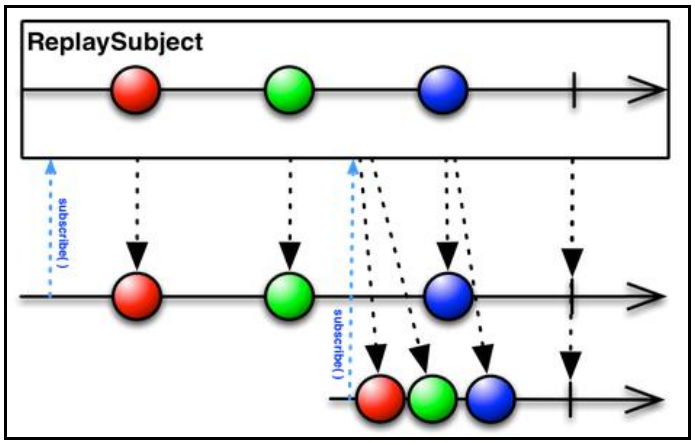
http://reactivex.io/rxjs/manual/overview.html#behaviorsubject: „BehaviorSubjects are useful for representing "values over time". For instance, an event stream of birthdays is a Subject, but the stream of a person's age would be a BehaviorSubject.”

Ein BehaviorSubject merkt sich den letzten Wert. Damit bekommt ein neuer Subscriber sofort einen Wert und nicht erst, wenn ein neuer emittiert wird. Daher muss im Konstruktor auch schon ein Wert mitgegeben werden.

```
RxJsVisualizer.prepareCanvas(['produce', 'A', 'B']);
const subject = new BehaviorSubject(0);
subject.subscribe(RxJsVisualizer.observerForLine(0, 'produce'))
Rx.timer(1000).subscribe(_ => subject.next(1));
Rx.timer(2000).subscribe(_ => subject.next(2));
Rx.timer(3000).subscribe(_ => subject.next(3));
Rx.timer(4000).subscribe(_ => subject.next(4));
Rx.timer(5000).subscribe(_ => subject.complete());
Rx.timer(1500).subscribe(_ => {
  RxJsVisualizer.writeToLine(1, 'subscribe A');
  subject.subscribe(RxJsVisualizer.observerForLine(1, 'ObserverA'));
});
Rx.timer(3500).subscribe(_ => {
  RxJsVisualizer.writeToLine(1, 'subscribe B');
  subject.subscribe(RxJsVisualizer.observerForLine(2, 'ObserverB'));
});
```

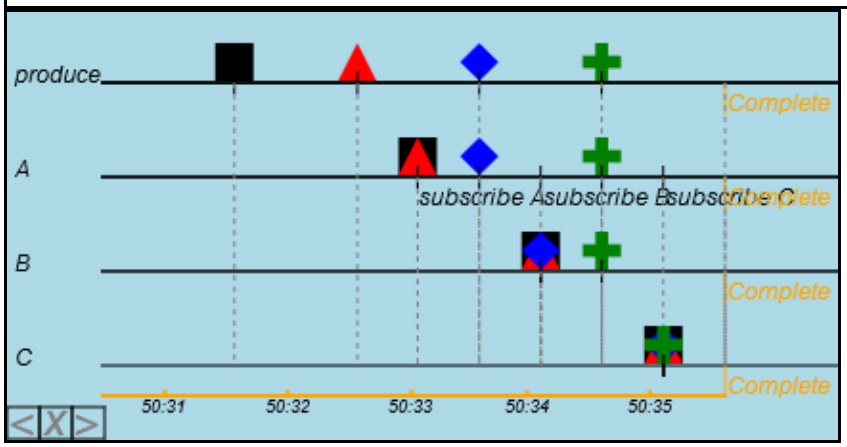


2.2 ReplaySubject



http://reactivex.io/rxjs/manual/overview.html#replaysubject: "A **`ReplaySubject`** records multiple values from the Observable execution and replays them to new subscribers."
Auch hier bekommt ein späterer Subscriber sofort einen Wert, aber nicht nur den letzten, sondern alle bisher angefallenen.

```
RxJsVisualizer.prepareCanvas(['produce','A', 'B', 'C']);
const subject = new ReplaySubject();
subject.subscribe(RxJsVisualizer.observerForLine(0, 'produce'))
Rx.timer(1000).subscribe(_ => subject.next(1));
Rx.timer(2000).subscribe(_ => subject.next(2));
Rx.timer(3000).subscribe(_ => subject.next(3));
Rx.timer(4000).subscribe(_ => subject.next(4));
Rx.timer(5000).subscribe(_ => subject.complete());
Rx.timer(2500).subscribe(_ => {
  RxJsVisualizer.writeToLine(1, 'subscribe A');
  subject.subscribe(RxJsVisualizer.observerForLine(1, 'ObserverA'));
});
Rx.timer(3500).subscribe(_ => {
  RxJsVisualizer.writeToLine(1, 'subscribe B');
  subject.subscribe(RxJsVisualizer.observerForLine(2, 'ObserverB'));
});
Rx.timer(4500).subscribe(_ => {
  RxJsVisualizer.writeToLine(1, 'subscribe C');
  subject.subscribe(RxJsVisualizer.observerForLine(3, 'ObserverC'));
});
```

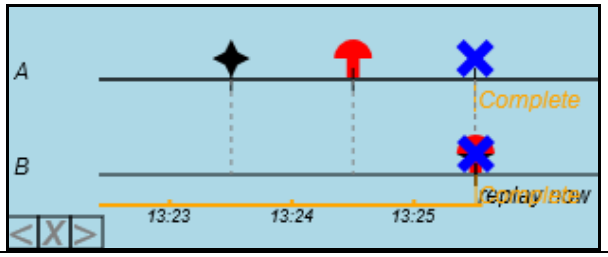


2.2.1 Replay

Da ein Subject sowohl Observer als auch Observable ist, kann man ein ReplaySubject benutzen, um Daten zu speichern und wiederzuverwenden.

```
const replaySubject = new ReplaySubject();
const observable = Rx.interval(1000).pipe(map(x => (x + 1) * 10), take(3));
observable.subscribe(RxJsVisualizer.observerForLine(0, 'produce'));
observable.subscribe(null, null,
  _ => {
    RxJsVisualizer.writeToLine(1, 'replay now');
    replaySubject.subscribe(RxJsVisualizer.observerForLine(1, 'replay'));
  });
observable.subscribe(replaySubject);
```

```
12:13:23.540  produce 10
12:13:24.546  produce 20
12:13:25.552  produce 30
12:13:25.552  Completed produce
12:13:25.554  replay now
12:13:25.556  replay 10
12:13:25.557  replay 20
12:13:25.559  replay 30
12:13:25.560  Completed replay
```



Die Werte werden also reproduziert, in der richtigen Reihenfolge zwar, aber unverzögert.
Wie könnte man eine Verzögerung einbauen, damit es so aussieht?