WEB 3, Session 10

Reactive Programming

Reactive Programming

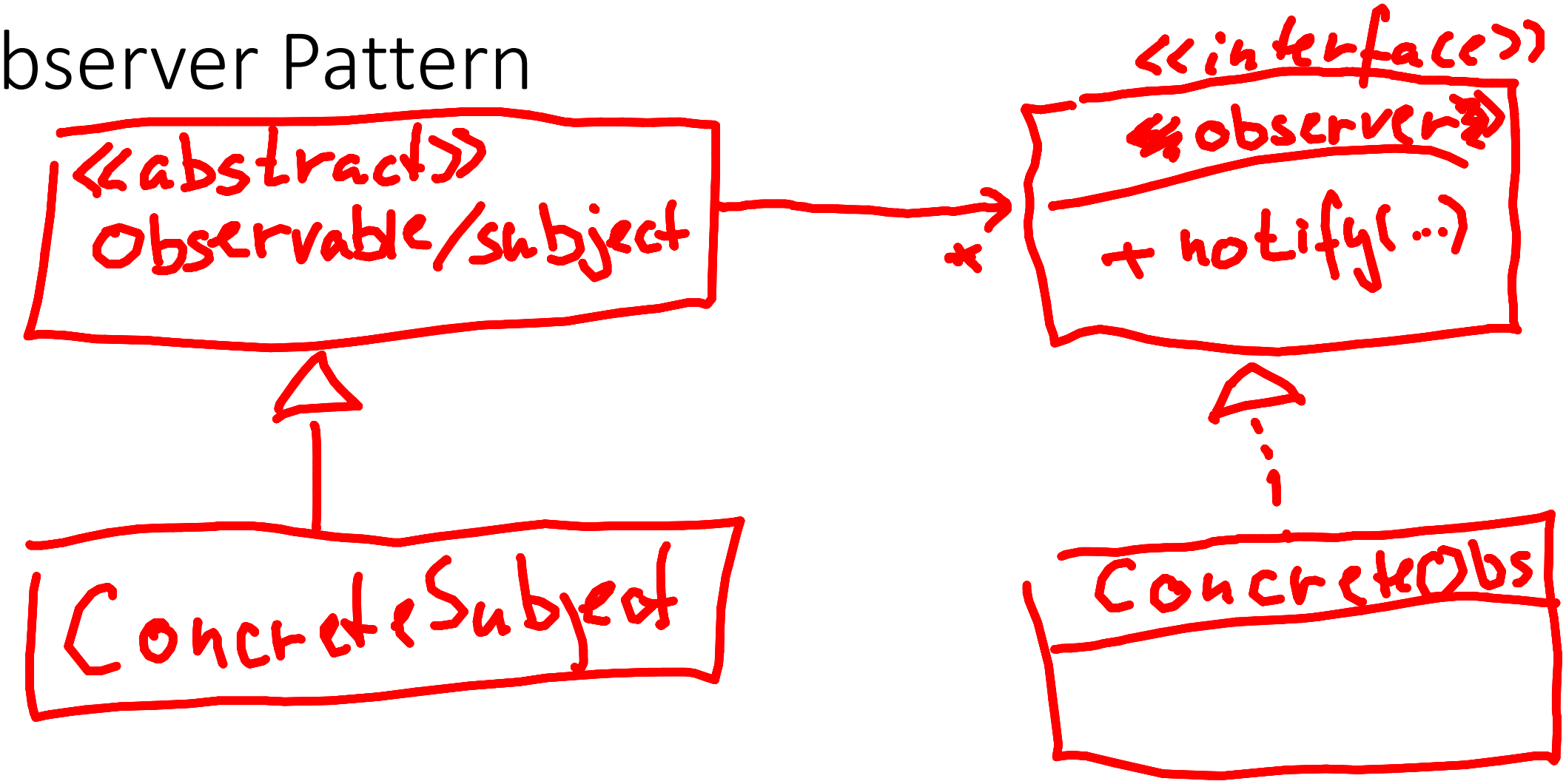
- (Not React)
- Wikipedia:
  "In computing, *reactive programming* is a declarative programming paradigm concerned with data streams and the propagation of change."
- ReactiveX:
  "The Observer pattern done right"
- "ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming"

RxJS

- Reactive programming for JavaScript
  - Part of the ReactiveX family
- Designed to consume
  - A Kafka Stream
  - A RabbitMQ queue
  - Web sockets - including GraphQL subscription
  - Server-send events
- Far from universally used
  - Really good when you need it
- Particularly good for event-driven architectures

Ole Hougaard, olehougaard@gmail.com

Observer Pattern



Observer Pattern enhancements: Events

- Event
  - The usual observer pattern events
- Error
  - Because not everything succeeds
  - Marks the end of the event stream
- Complete
  - An event to mark the successful end of the stream

Subscribing to events

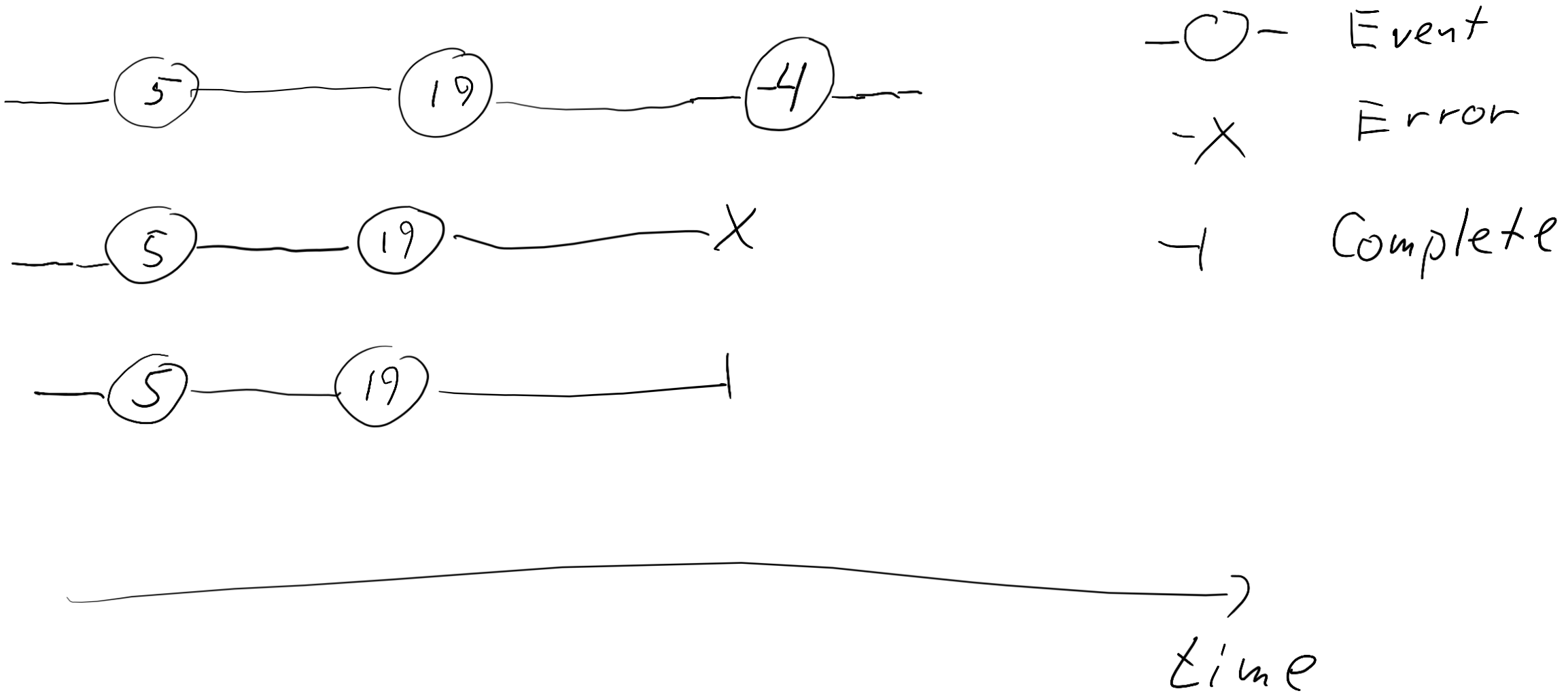
```
const observable = ...

observable.subscribe(evt => {...})


observable.subscribe({
  next(evt) {...},
  error(err) {...},
  complete() {}
})
```

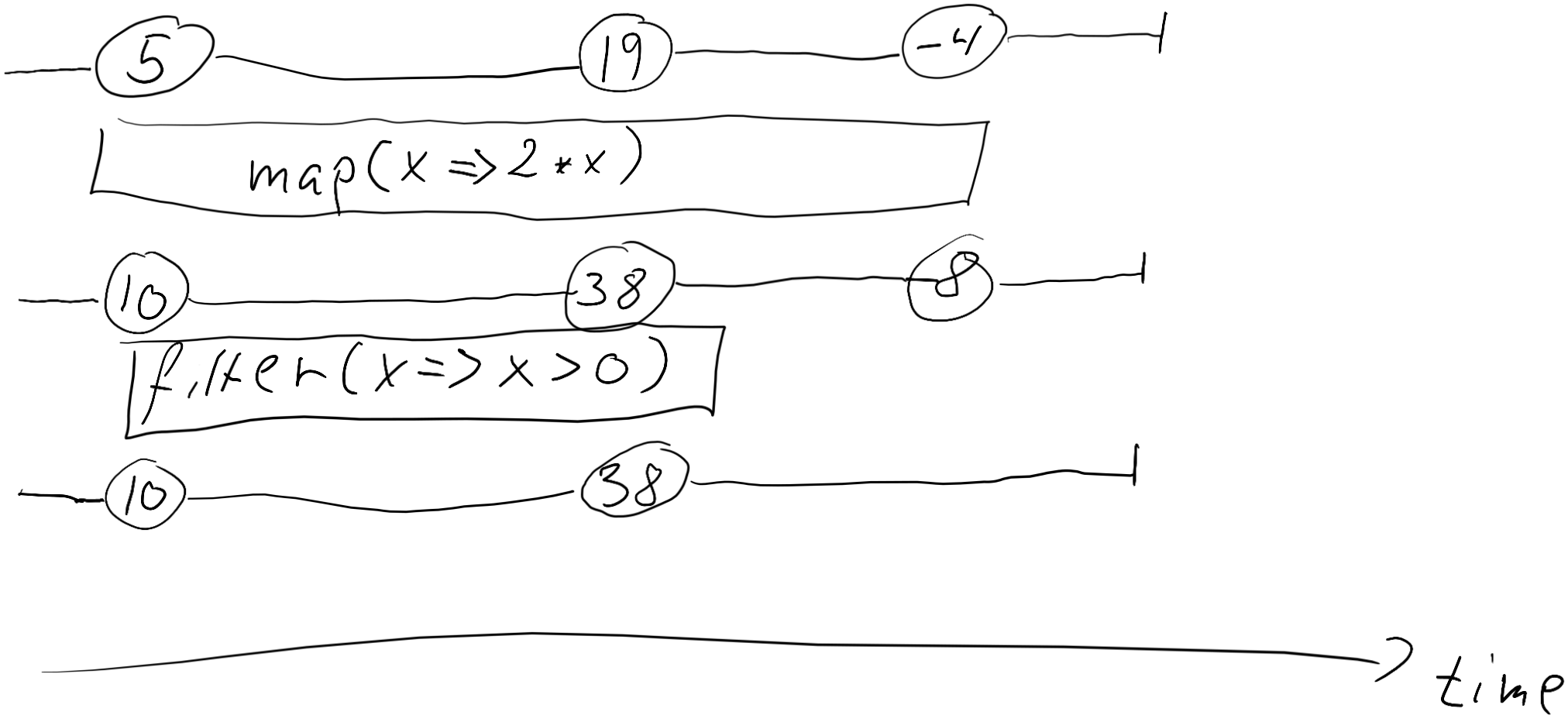
Event Sources ("producers")

- fromEvent
  - From DOM events
  - `fromEvent(button, "click")`
- ajax
  - Server responses
  - 1 event, then complete
- interval
  - A timer that keeps sending events 0, 1, 2, 3, 4, 5, …
- WebSocket
  - Streams web socket messages


5
19
4
5
19
X
5
19
Event
Error
Complete
Time

Operator functions

- Functions that take observables and return observables
- An operator can
  - Transform the events
  - Block the events
  - Emit new events
  - Complete the stream
  - Emit errors
  - Handle errors
- And combine several of the above


5
19
-4
map(x => 2 * x)
10
38
8
filter(x => x > 0)
10
38
time

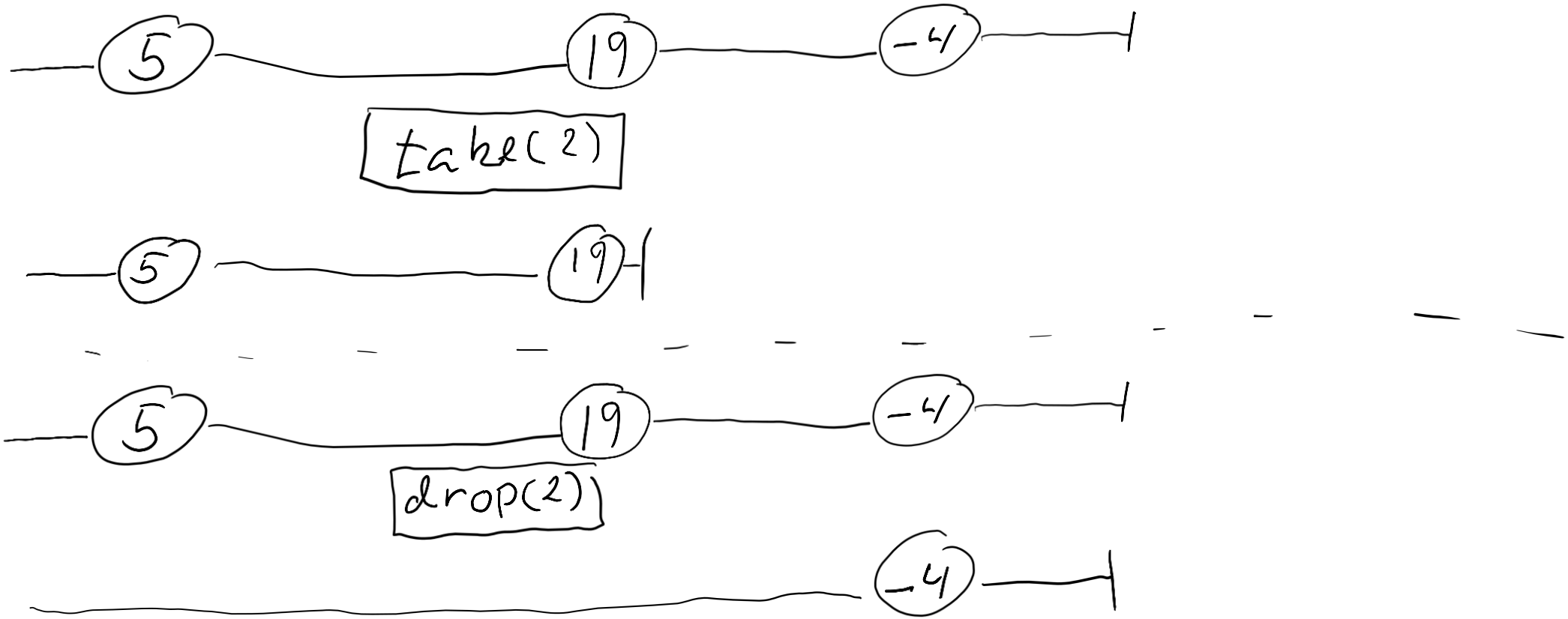
Pipelines

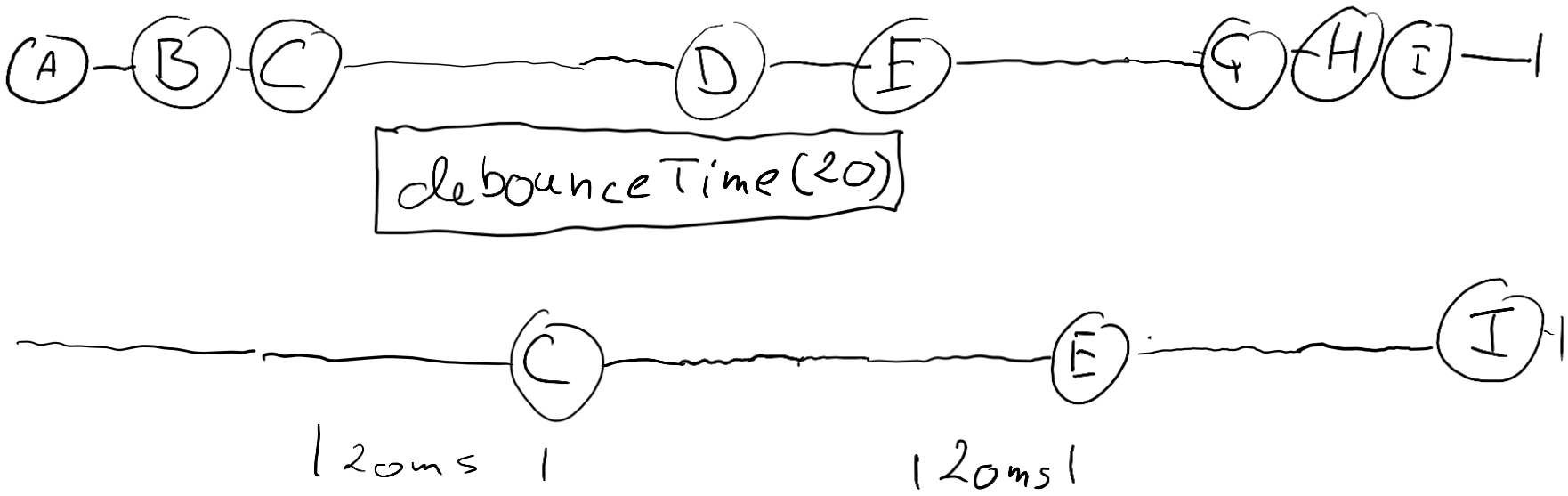
```
observable.pipe(
  map(x => 2 * x),
  filter(x => x > 0)
)
```

Ole Hougaard, olehougaard@gmail.com

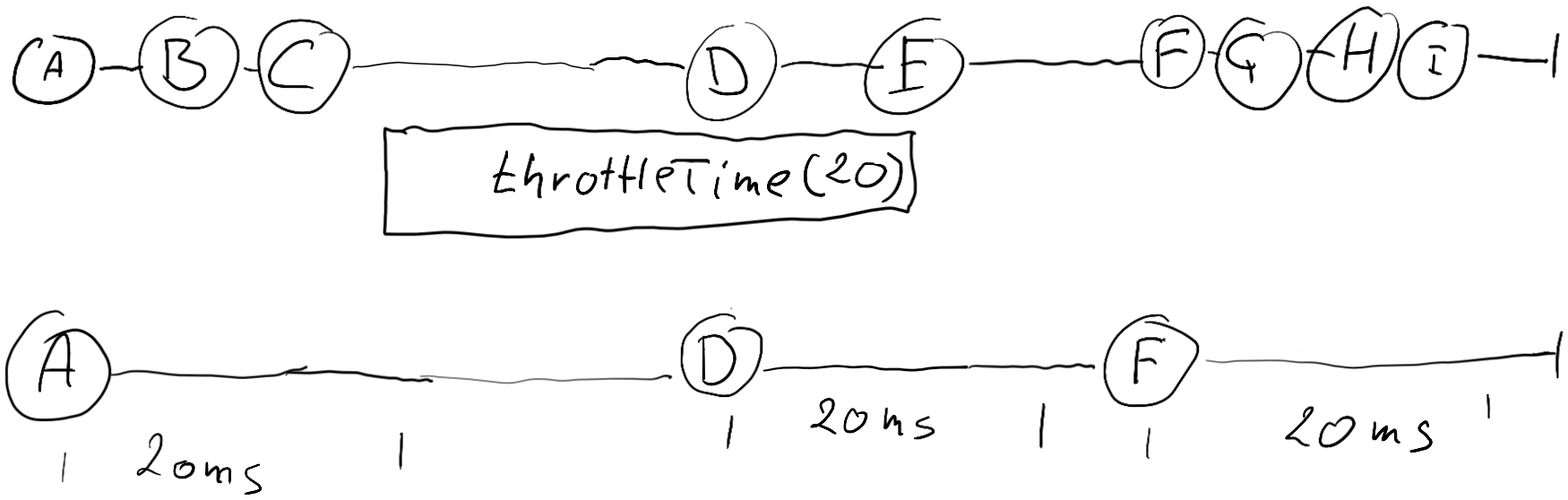
fromEvent example

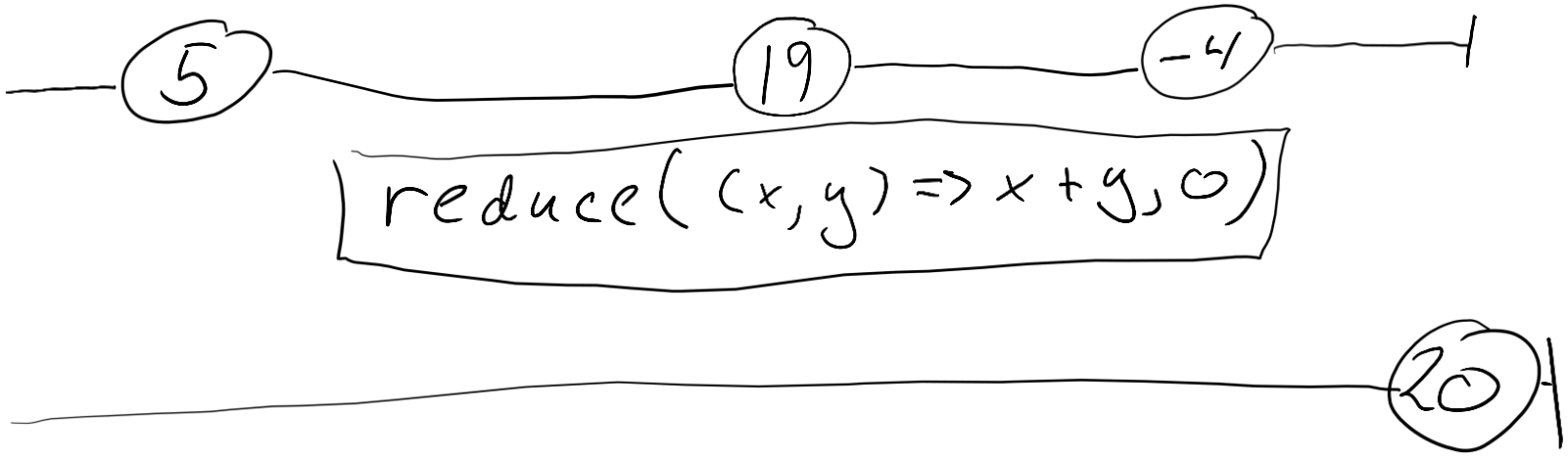
```
fromEvent(button, "click").pipe(
  map(getText),
  tap(text => console.log(text)), // For debugging.
  filter(text => text !== '')
)
.subscribe(addBullet)
```


5
19
-4
take(2)
5
19
5
19
-4
drop(2)
-4

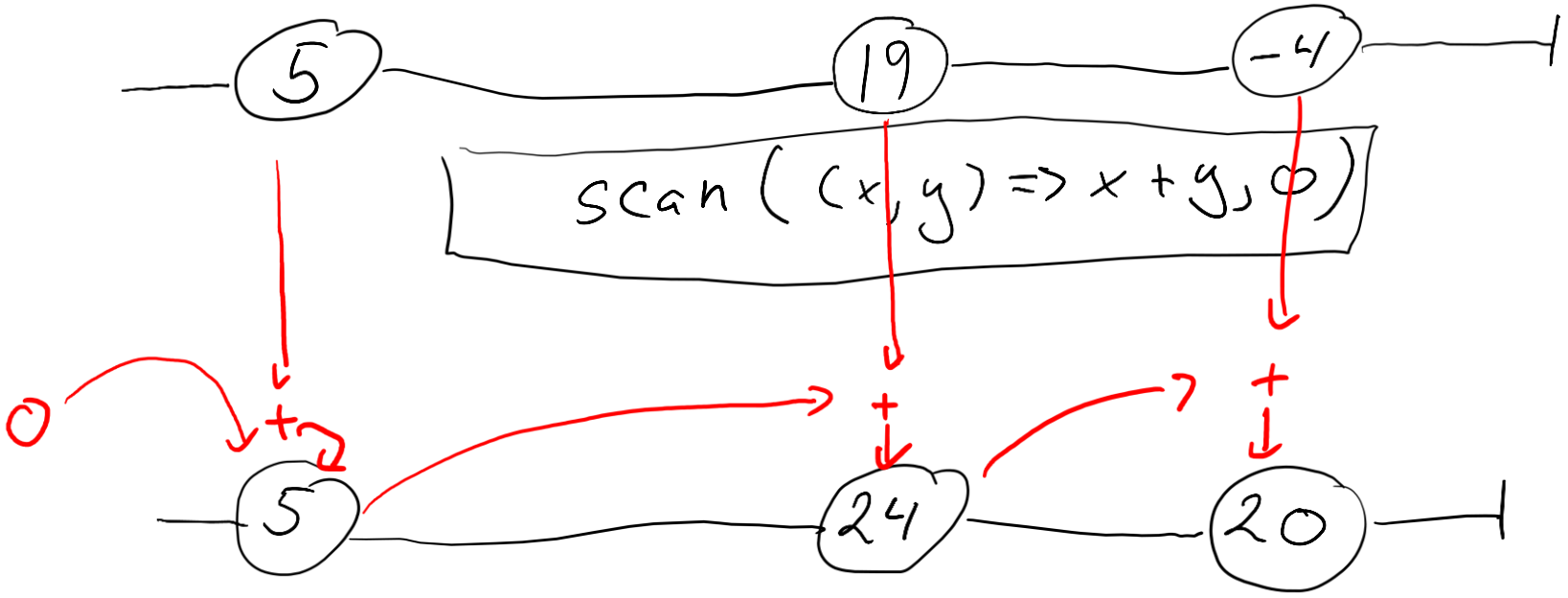

A
B
C
D
E
G
H
I
debounceTime(20)
C
E
I
120ms
120ms

Ole Hougaard, olehougaard@gmail.com




5
19
-4
reduce((x, y) => x + y, 0)
20

Ole Hougaard, olehougaard@gmail.com



Observable vs Subject

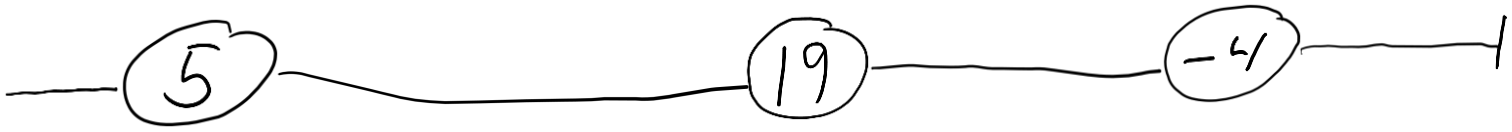
- Observable is *single-cast*
  - It can only have one subscriber
- Subject is *multi-cast*
  - It can have as many subscribers as needed
- Making an observable multi-cast
  - `obs.pipe(share())`

Sharing an observable

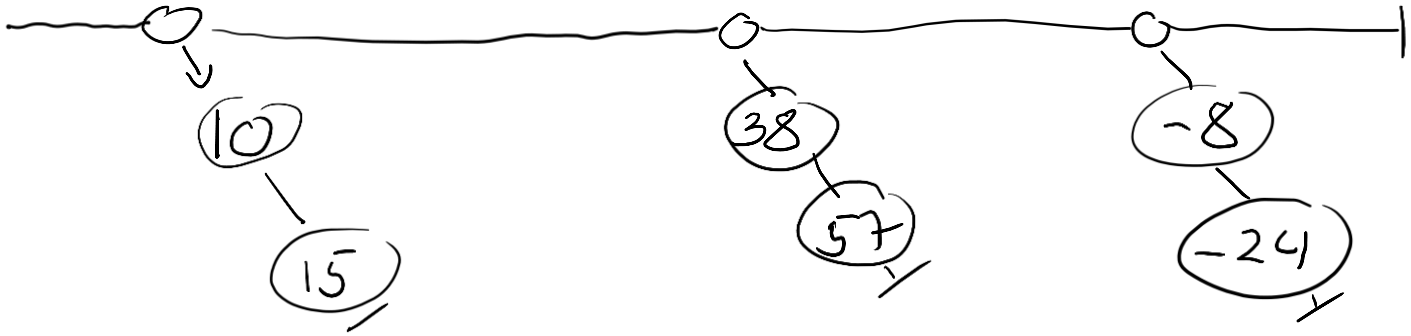
```
function subscribeToPlayerJoining(game: Game): Observable<Game> {
  return gameSubscription('game_' + game.gameNumber)
    .pipe(
      filter(game => game.ongoing),
      take(1),
      share()
    )
}
```

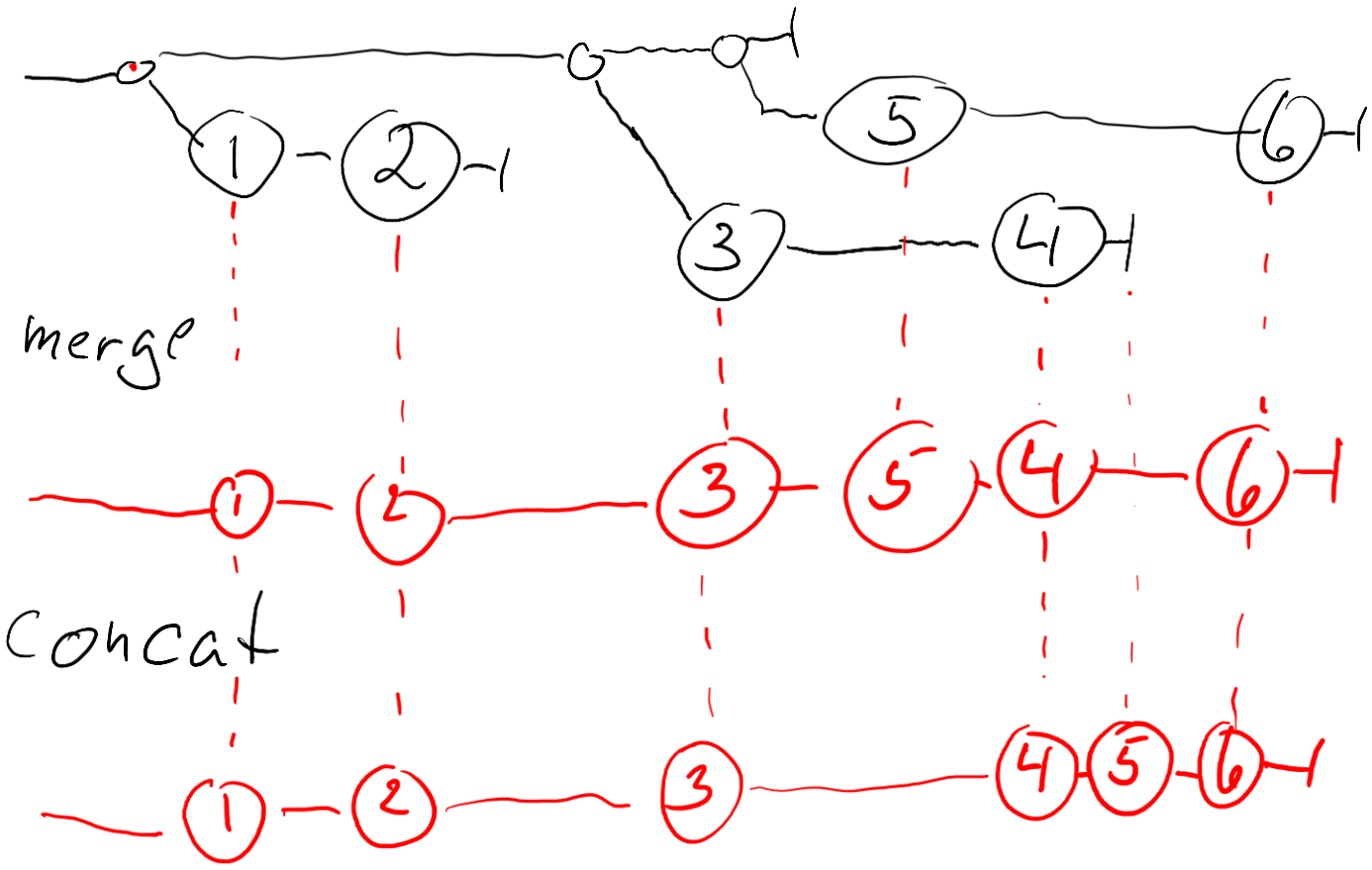
Observable of observable

- When does it happen?
  - You consume an event stream/web socket/…
  - For every event you start listening to a new stream/socket/…
  - Typically: map() the events into streams
- How to handle it? Merge or concat.
- Merge
  - Chronological
  - When you want to react to the events as they happen
- Concat
  - Ordered
  - When you want one stream to finish before you react to the next one




```
map(x => of(x*2, x*3))
```

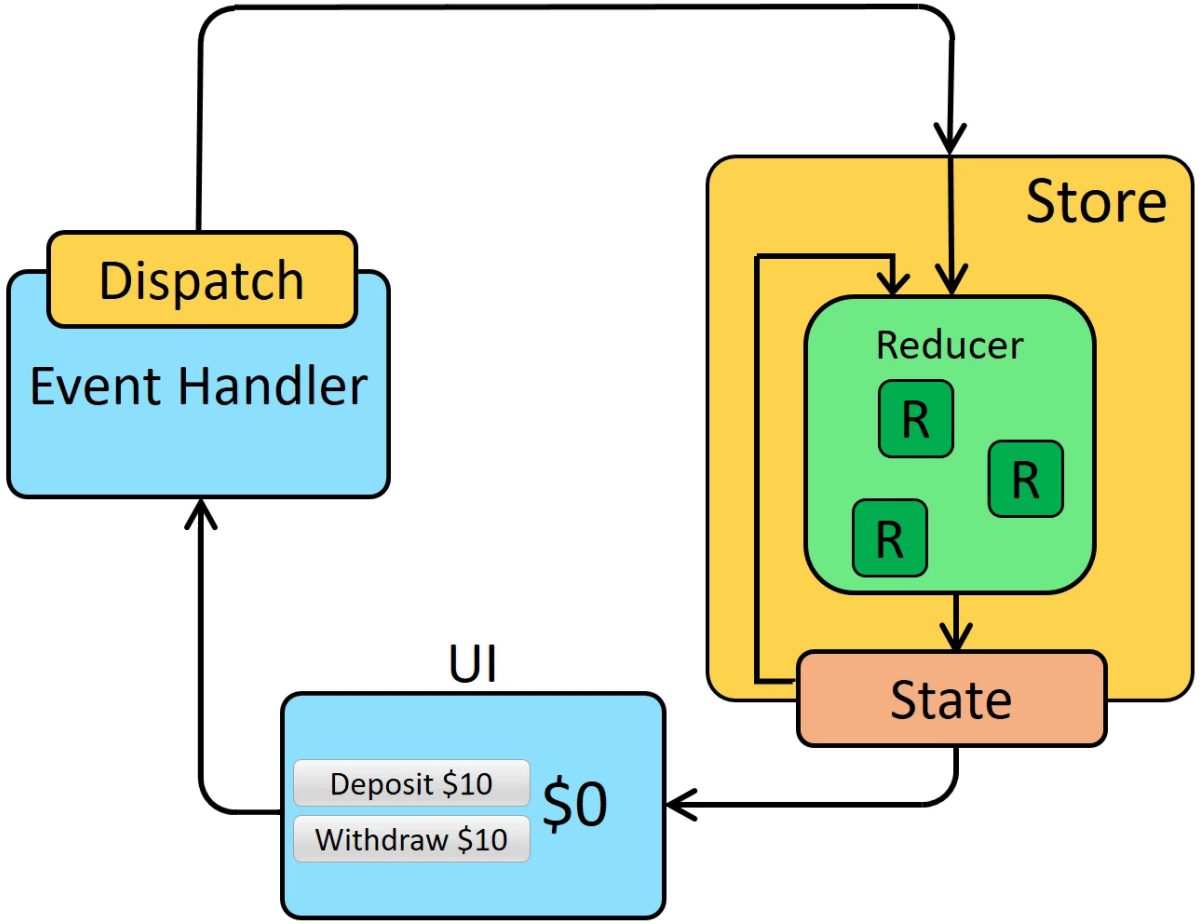


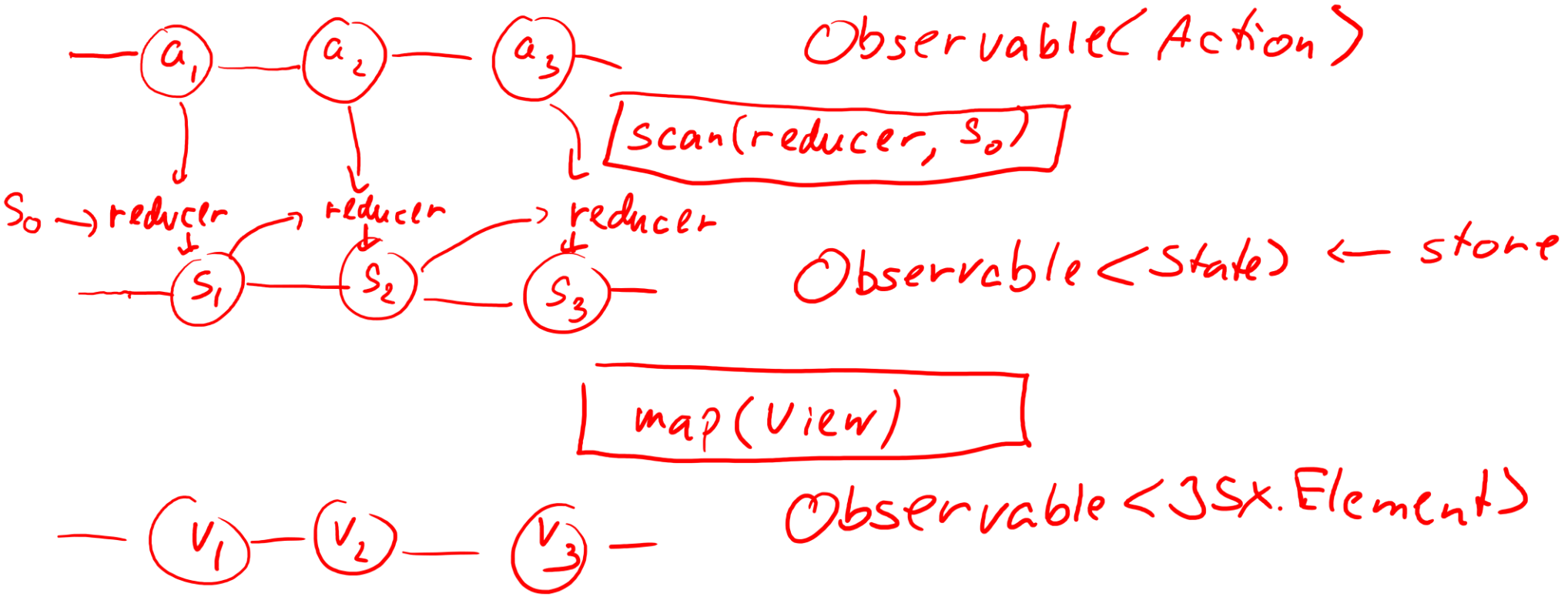

1
2
3
4
5
6
merge
1
2
3
5
4
6
Concat
1
2
3
4
5
6

Towards event driven architecture

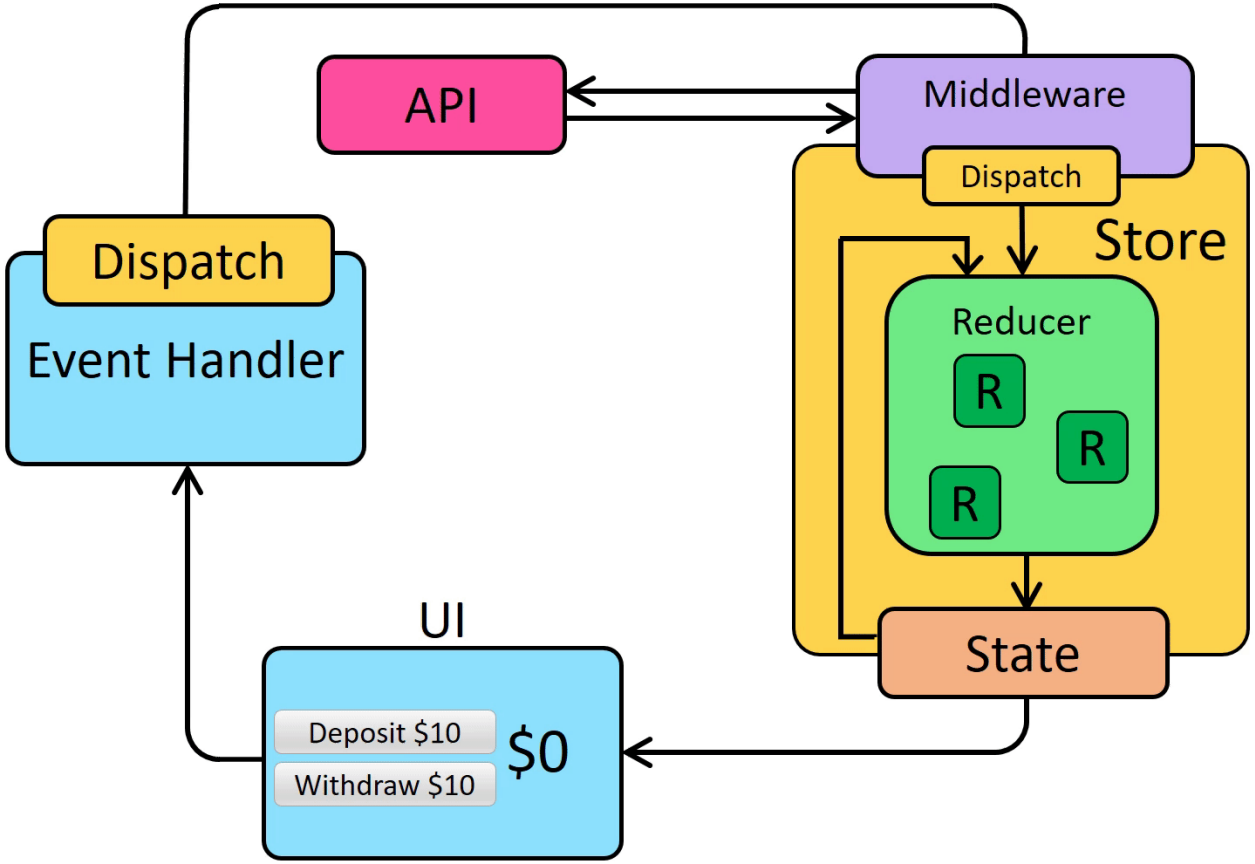
- Why event driven architecture?
  - More flexible
  - Lower coupling
  - More uniform if you have events anyway
- Why not event driven architecture?
  - Relentlessly asynchronous
  - Difficult to keep things in order
  - Difficult to follow the flow
- Let's look at 1-way data flow as an example
  - I do not expect you to do it or hear about it in the exam

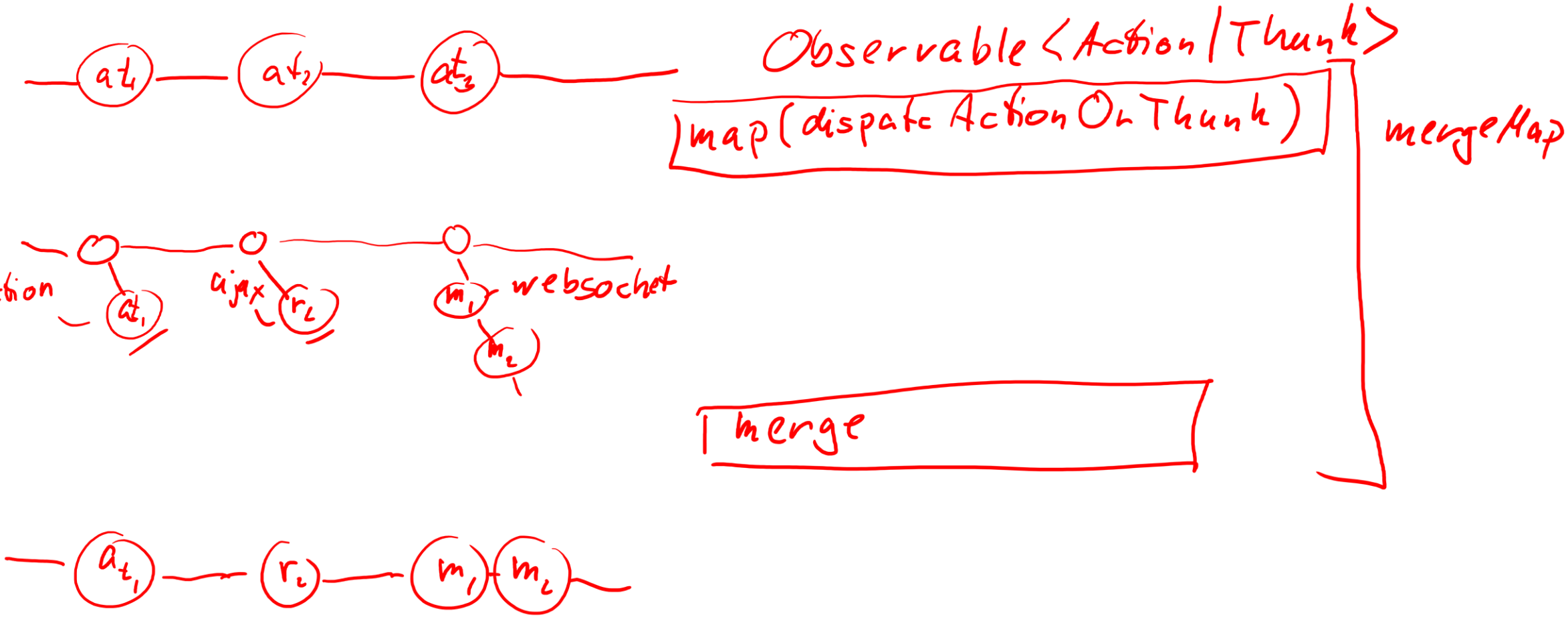
The Redux flow




a1
a2
a3
Observable<Action>
scan(reducer, s0)
s0
reducer
reducer
reducer
s1
s2
s3
Observable<State> ← store
map(view)
v1
v2
v3
Observable<JSX.Element>

The Redux flow with middleware




Observable<Action|Thunk>
map(dispatchActionOrThunk)
mergeMap
merge
action
ajax
websocket

Ole Hougaard, olehougaard@gmail.com

Code for dispatch

```
export type Thunk = () => Observable<Action>

export const actionSource = new Subject<Thunk | Action>()

export const dispatch = (actionThunk: Thunk | Action) =>
  actionSource.next(actionThunk)
```

Ole Hougaard, olehougaard@gmail.com

Code for dispatchThunk

```
export const dispatchThunk = (actionThunk: Action | Thunk): Observable<Action> =>
{
  if (typeof actionThunk === 'function')
    return actionThunk()
  else
    return of(actionThunk)
}
```

Ole Hougaard, olehougaard@gmail.com

Code for the Redux flow

```
actionSource.pipe(
  mergeMap(dispatchThunk),
  scan(reducer, init_state),
  map(view),
).subscribe(render)
```

Conclusion

- RxJS makes event handling concise
- It has high level of abstraction
- Use marble diagrams + the operator decision tree to design the pipeline
- RxJS can theoretically replace the Redux enhanced store
  - I don't know if you want to