



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE

# Reactive Extensions for JS Basics

Alex Thalhammer

# Outline

- Motivation
  - History of design pattern
  - Pull vs Push & Concurrency
  - Why reactive programming?
- Observable
- Observer
- Subscription
- Factories
- Subjects
- Managing Subscriptions
- Hot vs. Cold Observables
- Observables vs. Promises



# Motivation



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

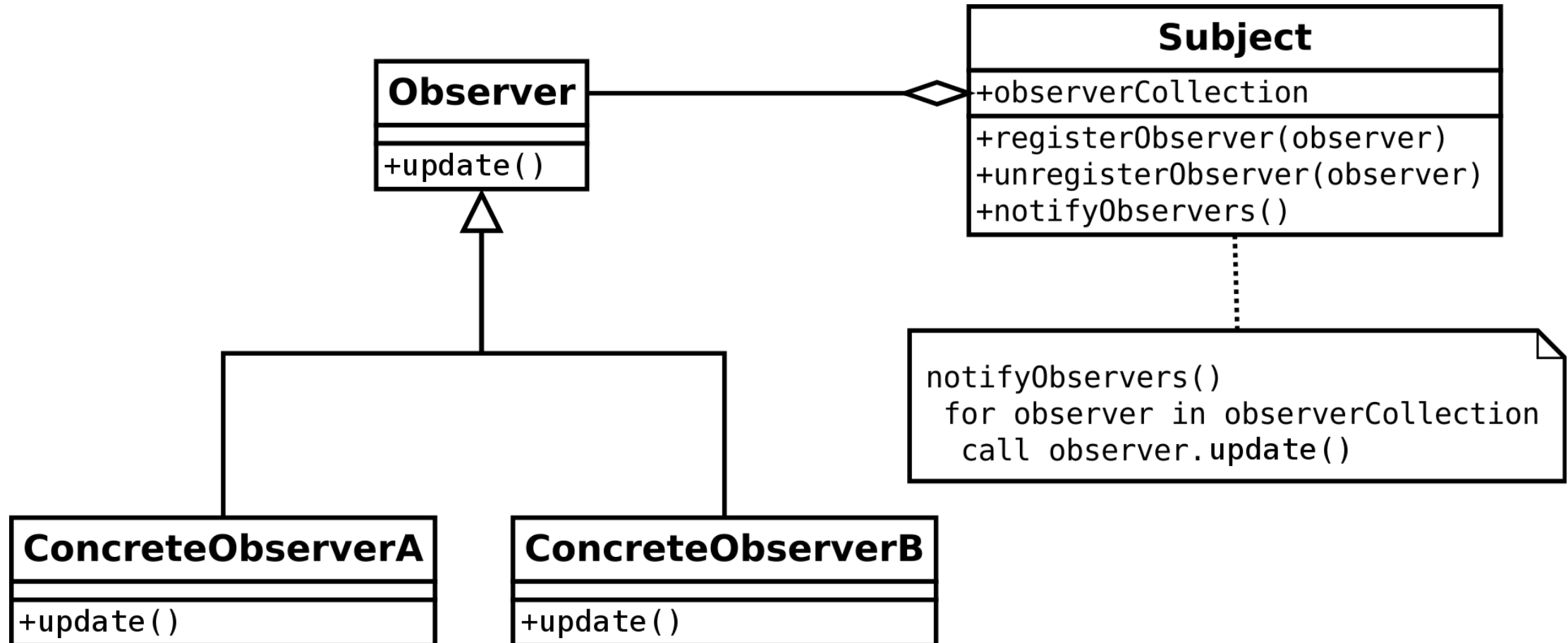
# Once upon a time

- Design Patterns (1994 - Gang Of Four)
  - Iterator Pattern (Behavioral Design Pattern)
    - Decouple data from algorithms

```
class Iterable {  
  [Symbol.iterator]() {  
    ...  
  }  
}  
  
const iterable = new Iterable();  
for (const item of iterable) {  
  ...  
}
```

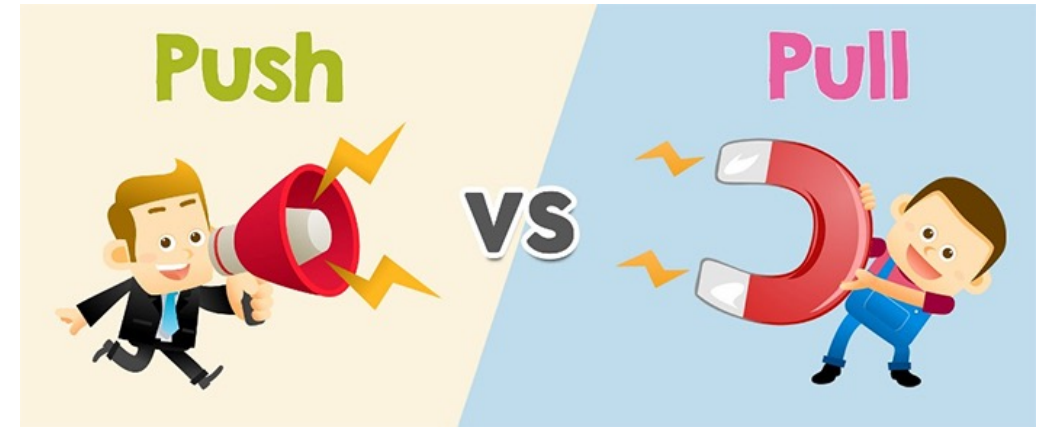


# Observer pattern (Behavioral DP)



# Pull vs Push Architecture (I)

- Pull-based
  - Consumer decide when data is pulled
  - Producer unaware when
  - Every function is a producer
- Push-based
  - Get notified when changes happen
  - E.g. Mobile App Push Notifications



# Pull vs Push Architecture (II)

	Producer	Consumer
Pull	<b>Passive:</b> produces data when requested.	<b>Active:</b> decides when data is requested.
Push	<b>Active:</b> produces data at its own pace.	<b>Passive:</b> reacts to received data.

# Concurrency (I)

- Synchronous vs. asynchronous computing
  - Latency → wait time
- Non-blocking code with callbacks
  - Often used in JavaScript





# Concurrency (II)

	Single items	Multiple items
synchronous / Pull	Function	Iterable (Array)
asynchronous / Push	Promise / async   await	?



# Concurrency (II)

	Single items	Multiple items
synchronous / Pull	Function	Iterable (Array)
asynchronous / Push	Promise / async   await	<b>Observable</b>



# Why asynchronicity?

Asynchronous  
operations  
(API requests)

Interactive  
behavior  
(user input)

Websockets

Server Send  
Events (Push)



# Why reactive programming?

- Enhances the user experience to be more fluid and responsive
- Simple to manage by developer
  - avoid "callback hell" → instead cleaner, readable code base
  - simple to compose / combine streams of data
  - simpler than traditional threading
- Powerful RxJS operators (best practices)
- But **difficult to learn** and can it cause **memory leaks**

# Observables & Observer



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# What are observables?

- Represents (asynchronous) data that is published over time
- A collection of values over any amount of time
  - 0..N values could be emitted
- Cancellable
- Lazy
- Operator support
  - Ton of functionality 😊



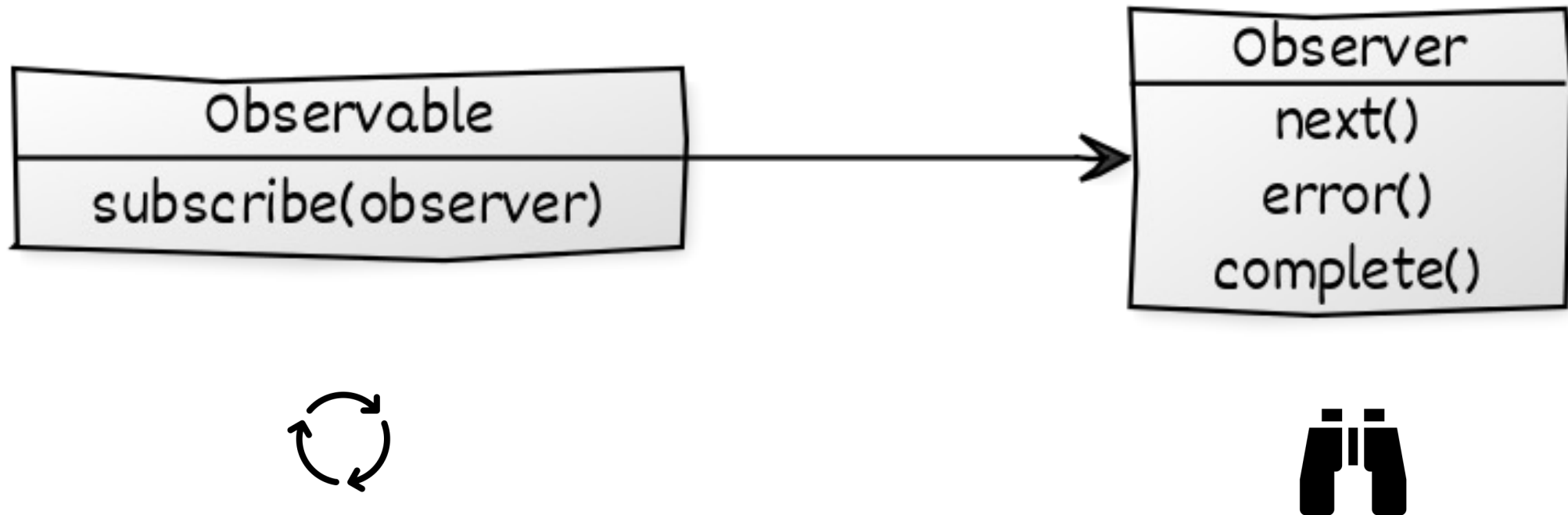
**Observable  
„Source“**

**Operator  
(z. B. map)**

**Observer  
„Destination“**



# Observable and Observer





# Subscribing an Observer



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Observer

```
myObservable.subscribe(  
  (value) => { ... }  
);
```

← **Observer**



# Observer

```
myObservable.subscribe(  
  (value) => { ... },  
  (err) => { ... },  
  () => { ... }  
);
```

← **Observer**

Option with multiple parameters  
was deprecated in V 6.4 and will  
not be supported in V 8 (soon!)



# Observer

```
myObservable.subscribe(  
  next: (value) => { ... },  
  error: (err) => { ... },  
  complete: () => { ... }  
));
```

← **Observer**



# Creating Observables



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Creating an Observable (rarely done this way)

```
let observable = new Observable((sender) => {  
    sender.next(4711);  
    sender.next(815);  
  
    // sender.error("err!");  
  
    sender.complete();  
});
```

} Sync/Async, Event-driven

```
let subscription = observable.subscribe(...);
```

```
subscription.unsubscribe();
```



# Creation Operators (Factories)

[<https://www.learnrxjs.io>]

fromEvent

of

throwError

interval

timer



# Subjects



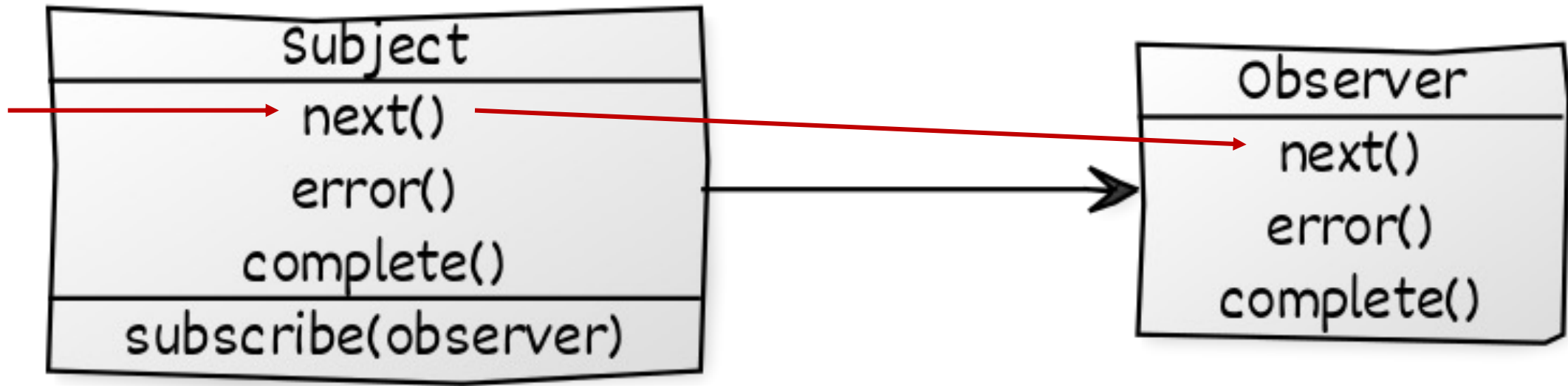
ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



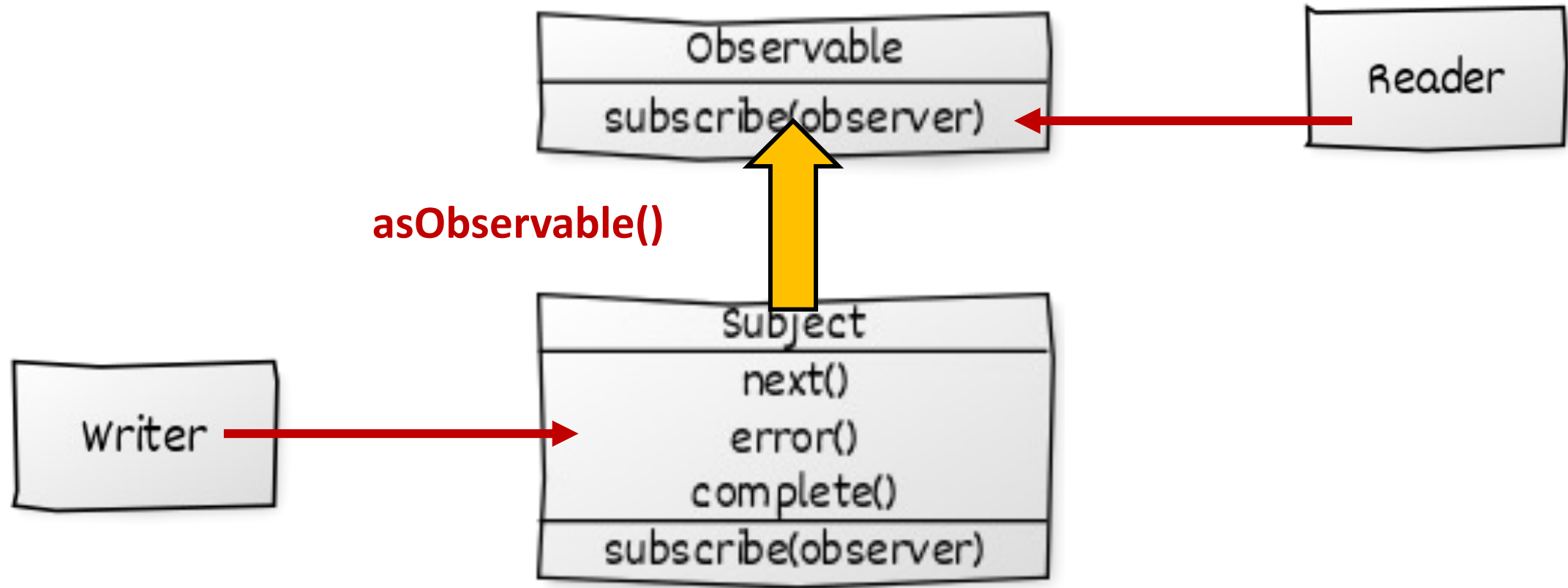
SOFTWARE  
**ARCHITECT**



# Subjects: Special Observables



# Convert Subject into Observable



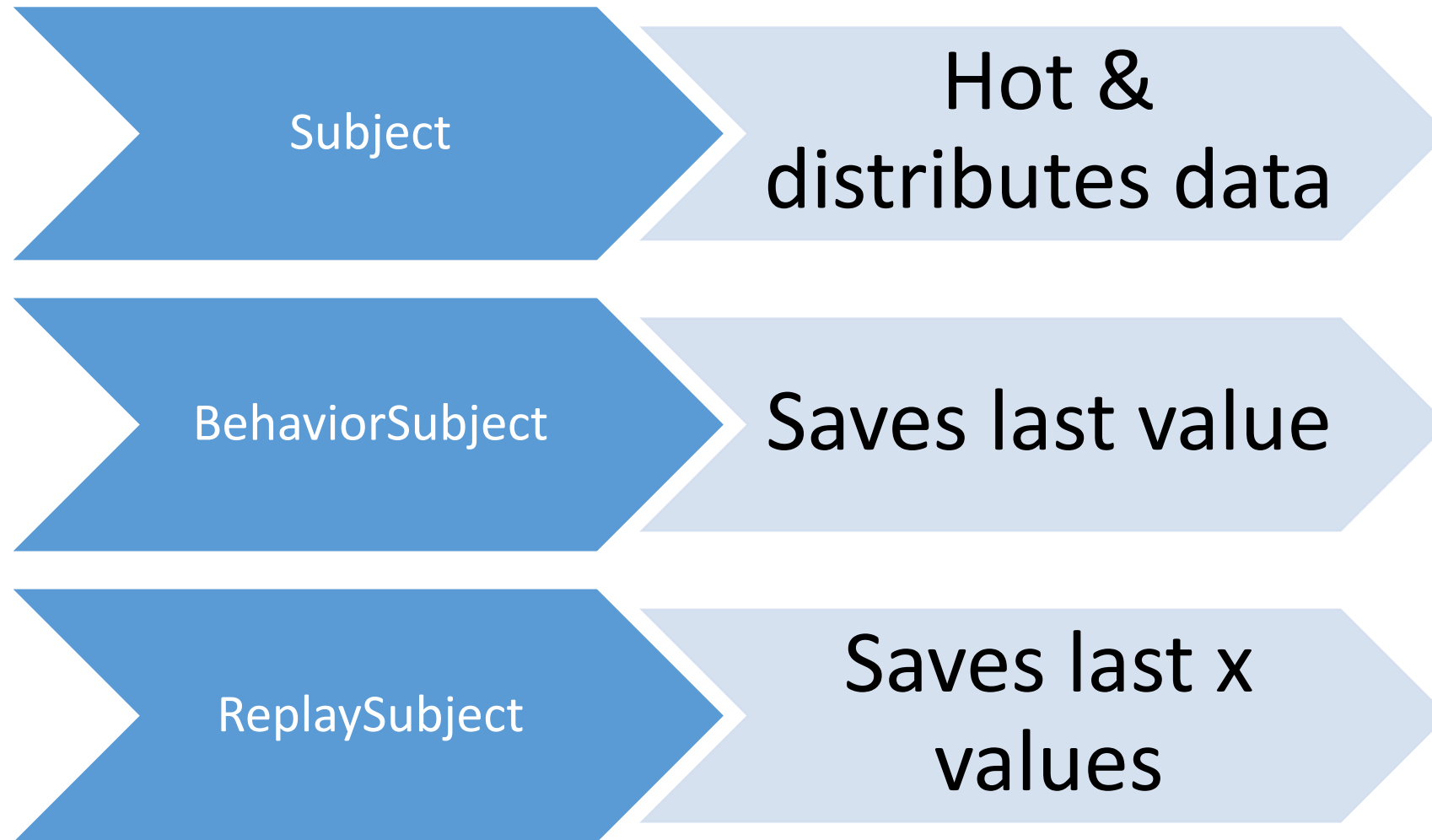
# asObservable

```
private subject = new Subject<Flight>();  
readonly observable = subject.asObservable();
```

```
[...]  
this.observable.subscribe(...)
```

```
[...]  
this.subject.next(...)
```

# Subjects



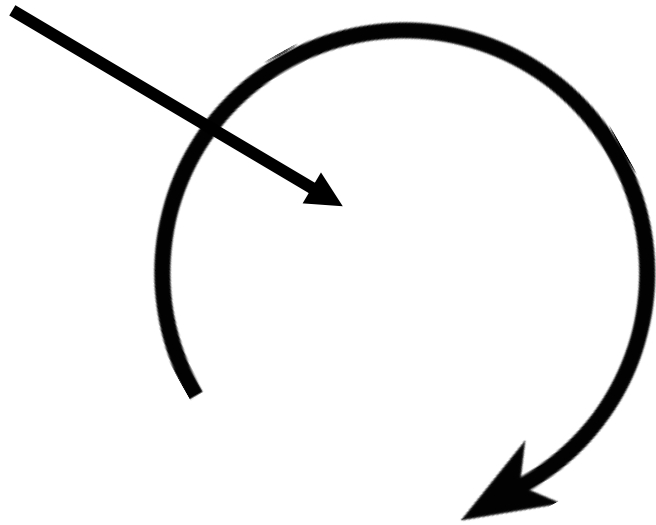
# Eventing with Subject

```
const sub = new Subject<Flight>();  
  
sub.subscribe((flight) => console.debug(flight));  
  
sub.next({ id: 1, ...})
```



# Subjects

Data/Notification



Subject

```
.subscribe(  
  (result) => { ... },  
  (error) => { ... },  
  () => { ... }  
));
```

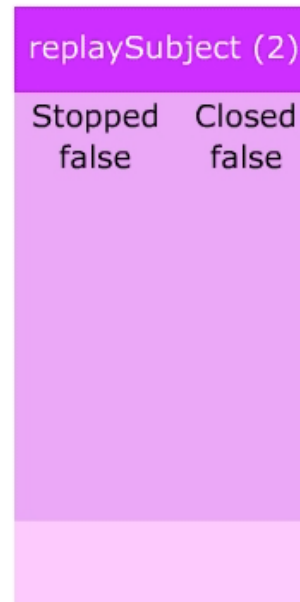
Observer

# State with BehaviorSubject

```
const temperature = new BehaviorSubject<number>(0);  
  
temperature.subscribe((temp) => console.debug(temp));  
  
temperature.next(-5);
```

# Diff with ReplaySubject

```
const diff = new ReplaySubject<number>(2);
```





# Managing Subscriptions

# Why do we (always!) need to unsubscribe?

Avoid side  
effects

Avoid  
memory leaks



Also for HttpClient's get / post ...

# Howto cancel subscriptions

- Explicitly

```
let subscription = observable$.subscribe(...);  
// subscription.add(otherObservable$.subscribe(...)); // also possible since V6  
subscription?.unsubscribe();
```

- Implicitly

- observable\$.pipe(**takeUntil(otherObservable)**).subscribe(...);
  - observable\$.pipe(**takeWhile(boolean)**).subscribe(...);
- } **last operator!**

- Implicitly with async-Pipe in Angular

{{ observable\$ | **async** }}  **also triggers a cdr.markForCheck for OnPush**

- Automatic by Angular

- Angular Router Params (the only 1 I know where unsubscribing is not needed)

# DEMO: Cancelling Subscriptions



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Cold vs. Hot Observables



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Cold vs. Hot Observables

## Cold

- Point to point
- Lazy: Only starts at subscription

## Hot

- Multicast
- Eager: Sender starts without subscriptions

Default



# Create Hot Observable

```
let o = this.find(from, to)
    .pipe(publish()) as ConnectableObservable<Flight[]>;

o.subscribe(...);

o.connect();

o.subscribe(...);
```



# Create Hot Observable

```
let o = this.find(from, to).pipe(pipe(share()));
```

```
o.subscribe(...);
```



```
o.subscribe(...);
```

**Sender starts with first subscription**

**Sender stops after all receiver have  
been unsubscribed**





# Create Hot Observable

```
let o = this.find(from, to)
    .pipe(shareReplay(1));

o.subscribe(...);

o.subscribe(...);
```



# DEMO: Hot Observable



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Lab

RxJS Basics



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Observables vs Promises

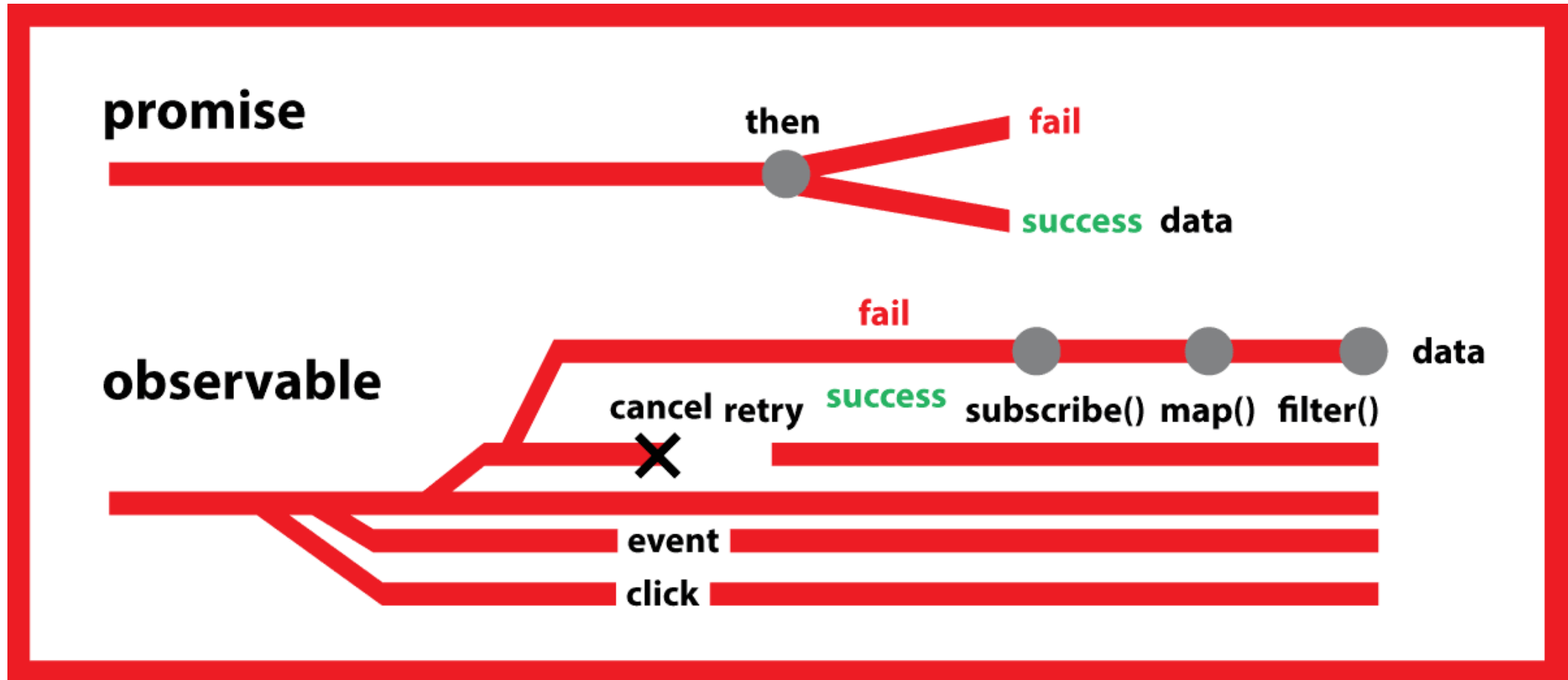


ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Observables vs Promises – Overview



# Observables vs Promises – Details

Observables (Streams)	Promises (Single Event)
More features	Less powerful
Can emit zero, <b>one or multiple</b> values over time.	Emit a <b>single</b> value at a time.
<b>Lazy</b> : they're not executed until we subscribe using the subscribe() method.	<b>Eager</b> : execute immediately after creation.
Subscriptions are <b>cancellable</b> using the unsubscribe() method, which stops the listener from receiving further values.	Are <b>not cancellable</b> .
<b>RxJS</b> provides a <b>ton of functionality</b> to operate on observables like the map, forEach, filter, reduce, retry, and retryWhen operators.	Don't provide any operations.
Deliver errors to the subscribers.	Push errors to the child promises.
Used by HTTP Client, Reactive Forms & Route Params	Used by Angular in Router.navigate



# Recap



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**