



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE

# Reactive Extensions for JS Basics

Alex Thalhammer

# Outline

- Motivation
- Observable
- Observer
- Subscription
- Factories
- Subjects
- Closing Subscription
- Hot vs. Cold Observables
- Observables vs. Promises



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# 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) {  
  ...  
}
```

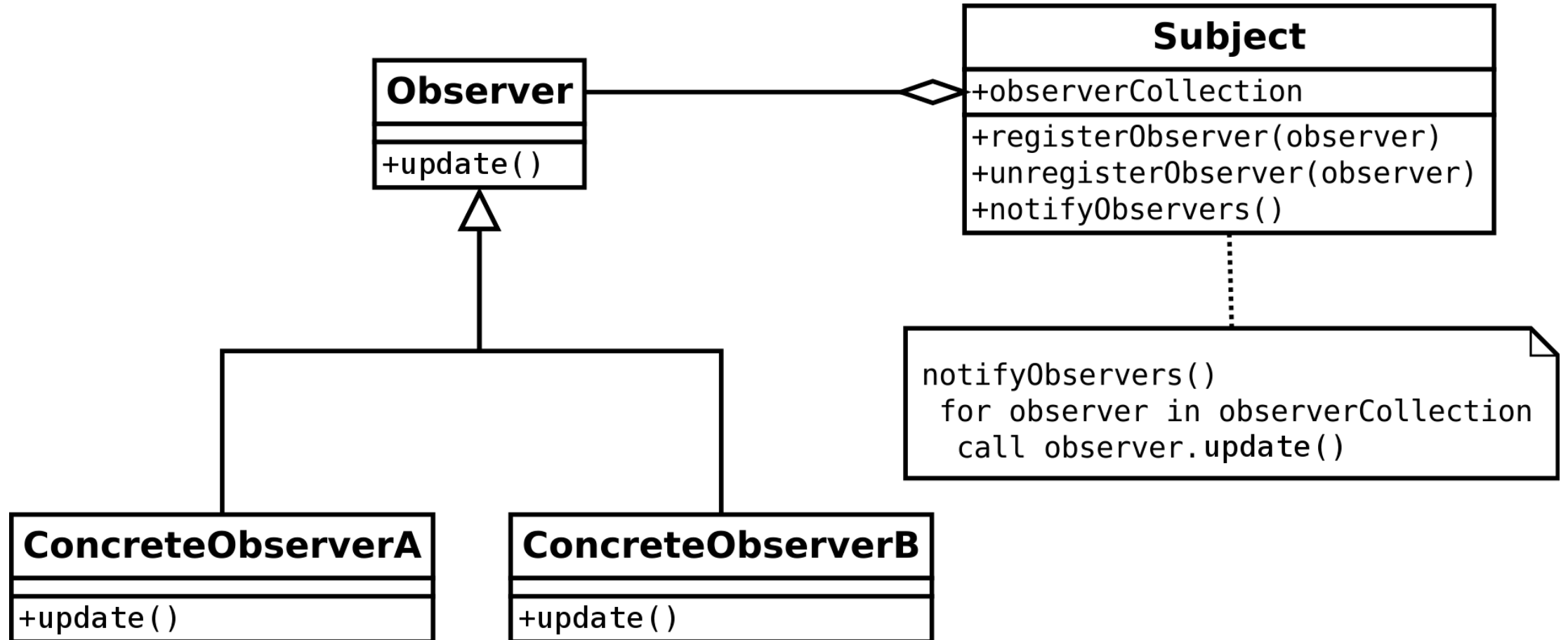


ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



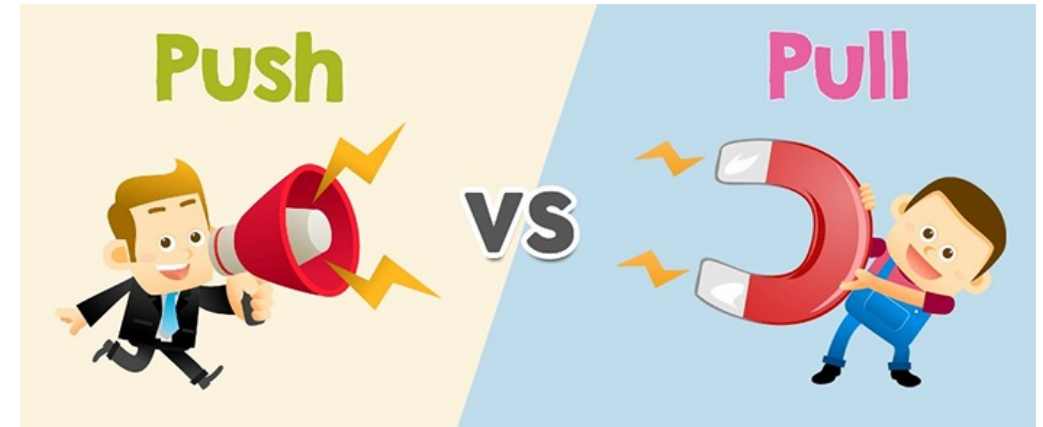
SOFTWARE  
**ARCHITECT**

# 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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Pull vs Push Architecture (II)

	Producer	Consumer
<b>Pull</b>	<b>Passive:</b> produces data when requested.	<b>Active:</b> decides when data is requested.
<b>Push</b>	<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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**



# Concurrency (II)

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



# Concurrency (II)

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

# Observables & Observer



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Why Observables?

Asynchronous  
operations

Interactive  
(reactive)  
behavior



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
- Cancelable
- Lazy
- Operator support



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Observable (asynchnornous) data streams?

- User Input
  - Mouse / Keyboard Interactions (e.g. mousemove, click, keydown)
- HTTP requests
- Websockets
- Server Send Events



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**



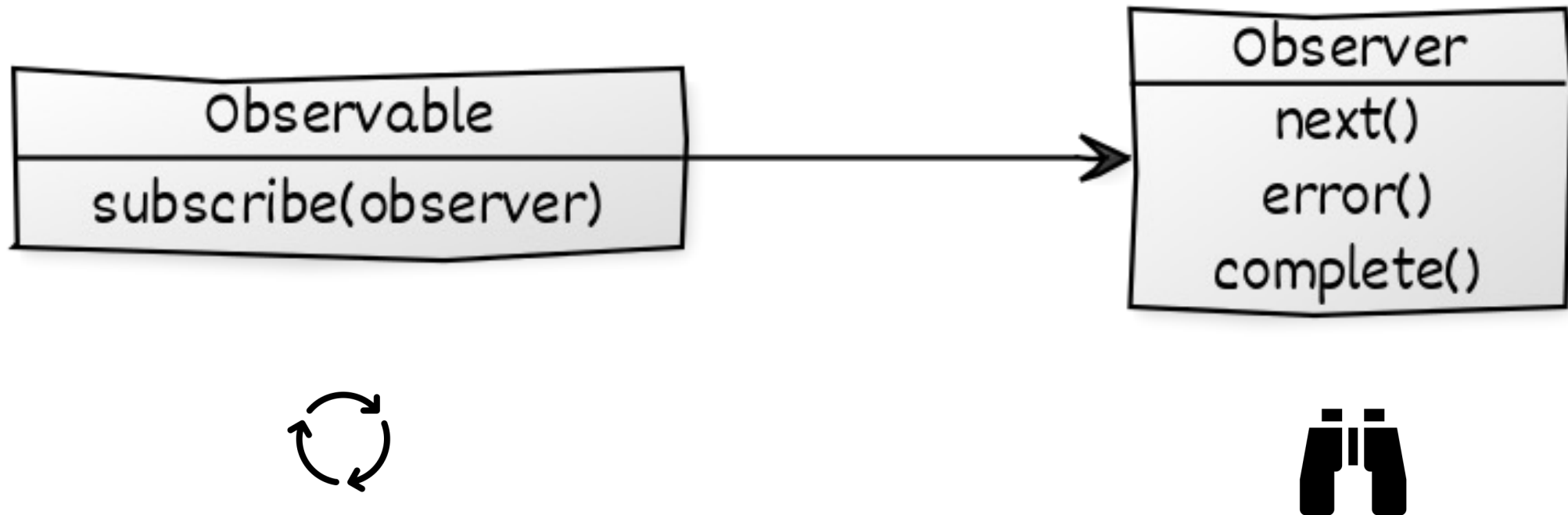
**Observable  
„Source“**

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

**Observer  
„Destination“**



# Observable and Observer





# Subscribing an Observer



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Observer

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

← **Observer**



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Observer

Option with more than one  
parameter is now deprecated!

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

← **Observer**



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# Observer

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



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT

# DEMO: Observable



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Creating Observables



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Creating an Observable

```
let observable = new Observable((sender) => {  
    sender.next(4711);  
    sender.next(815);  
    // sender.error("err!");  
    sender.complete();  
    return () => { console.debug('Bye bye'); };  
});
```

} **Sync/Async, Event-driven**

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

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



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Creation Operators (Factories)

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

fromEvent

of

throwError

interval

timer



ANGULAR  
ARCHITECTS  
INSIDE KNOWLEDGE



SOFTWARE  
ARCHITECT



# Subjects

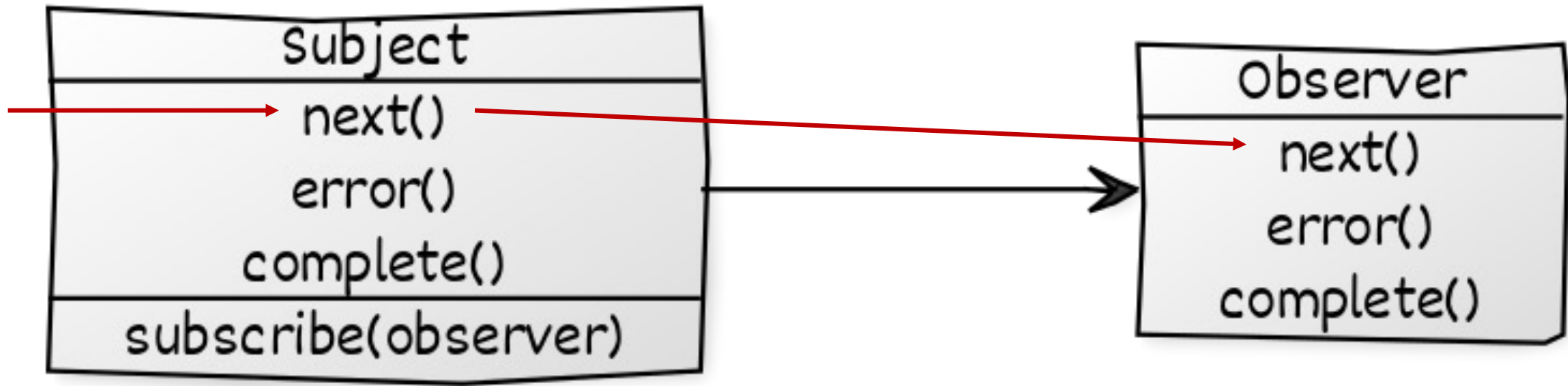


ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE

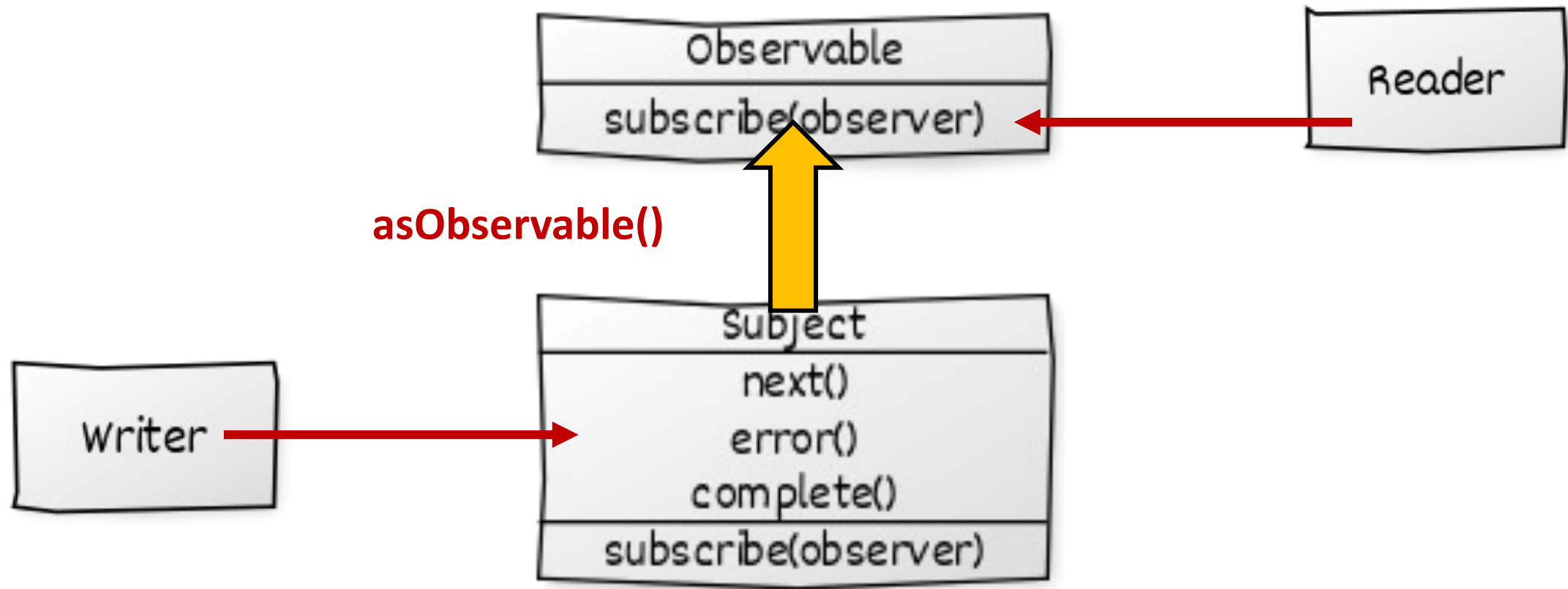


SOFTWARE  
**ARCHITECT**

# Subjects: Special Observables



# Convert Subject into Observable



# asObservable

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

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

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

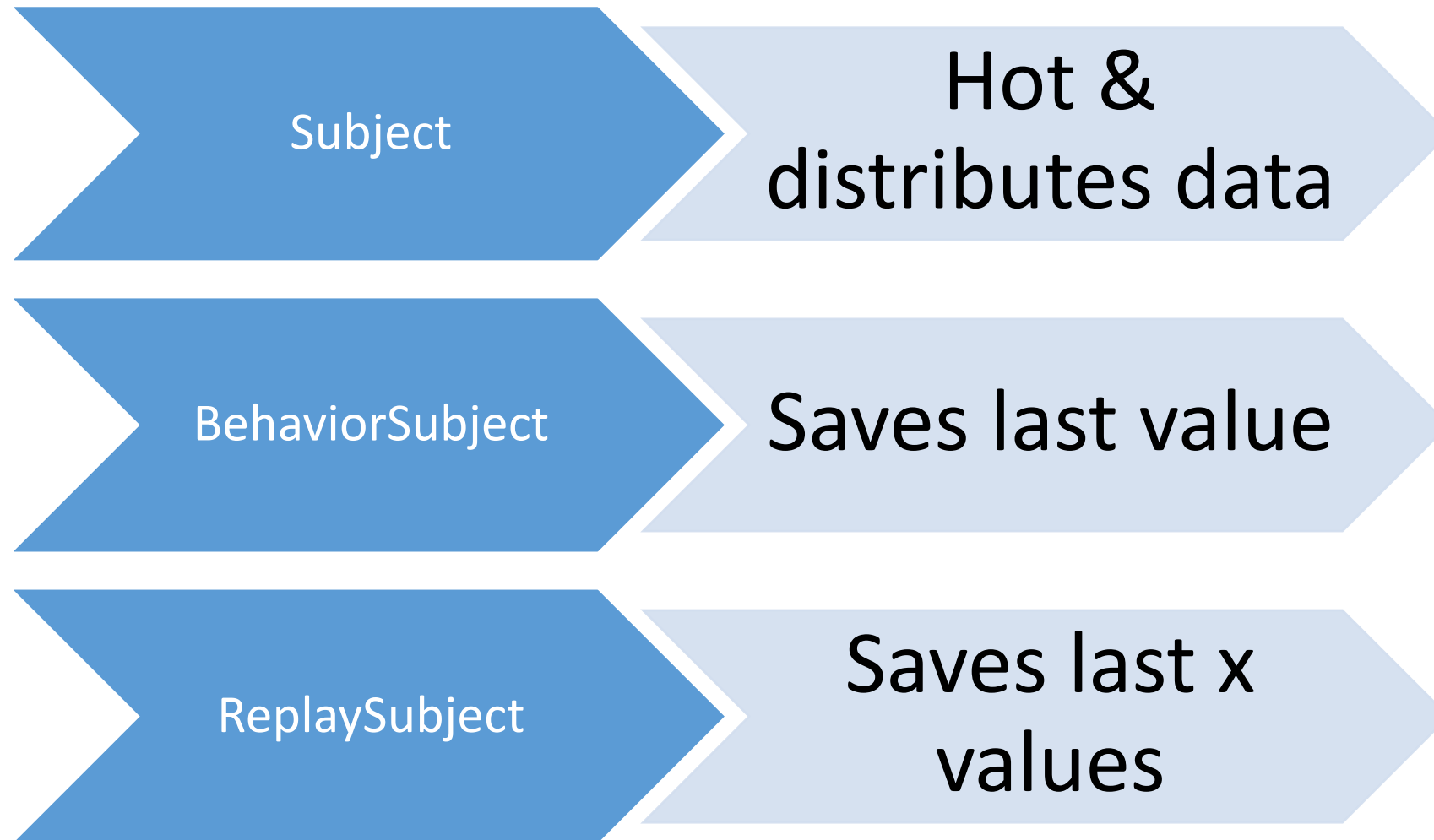


ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Subjects



# Eventing mit Subject

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



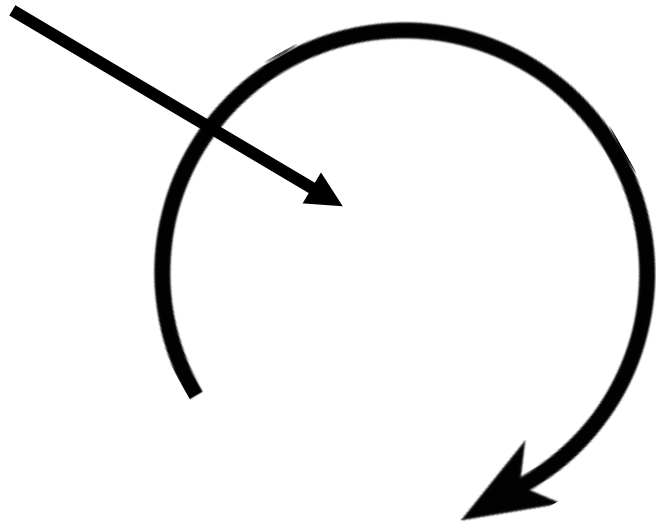
ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Subjects

Data/Notification



Subject

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

Observer



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Closing Subscription



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**



# Closing Subscription

- Explicitly

```
let subscription = observable$.subscribe(...);  
subscription?.unsubscribe();
```

- Implicitly

- observable\$.pipe(**takeUntil(otherSubject)**).subscribe(...);
- observable\$.pipe(**takeWhile(boolean)**).subscribe(...);

- Implicitly with async-Pipe in Angular

```
{{ observable$ | async }}
```

- Automatic by Angular

- Angular Router Params



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# DEMO: Closing Subscription



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**

# Cold vs. Hot Observables

# Cold vs. Hot Observables

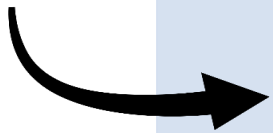
## Cold

- 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

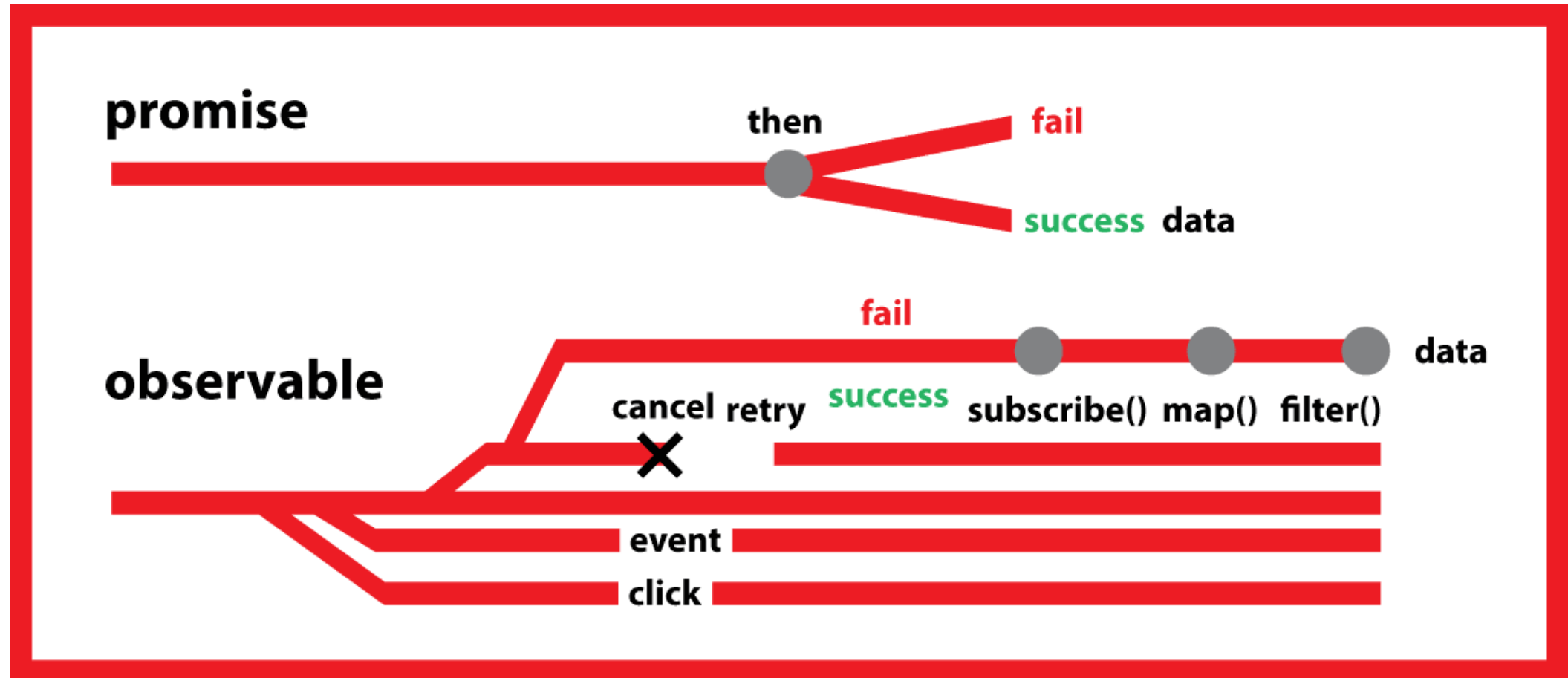


ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



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 Angular in HTTP Client & Route Params	Used by Angular in Router.navigate



# Recap



ANGULAR  
**ARCHITECTS**  
INSIDE KNOWLEDGE



SOFTWARE  
**ARCHITECT**