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
- Closing Subscription
- Hot vs. Cold Observables
- Observables vs. Promises

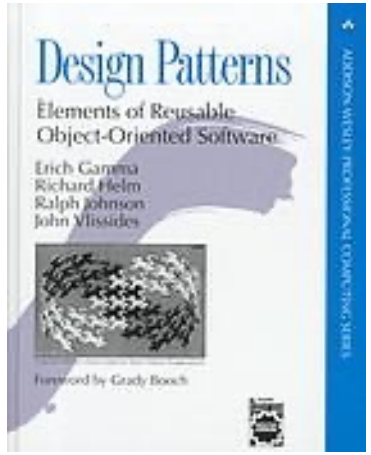
# Motivation

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

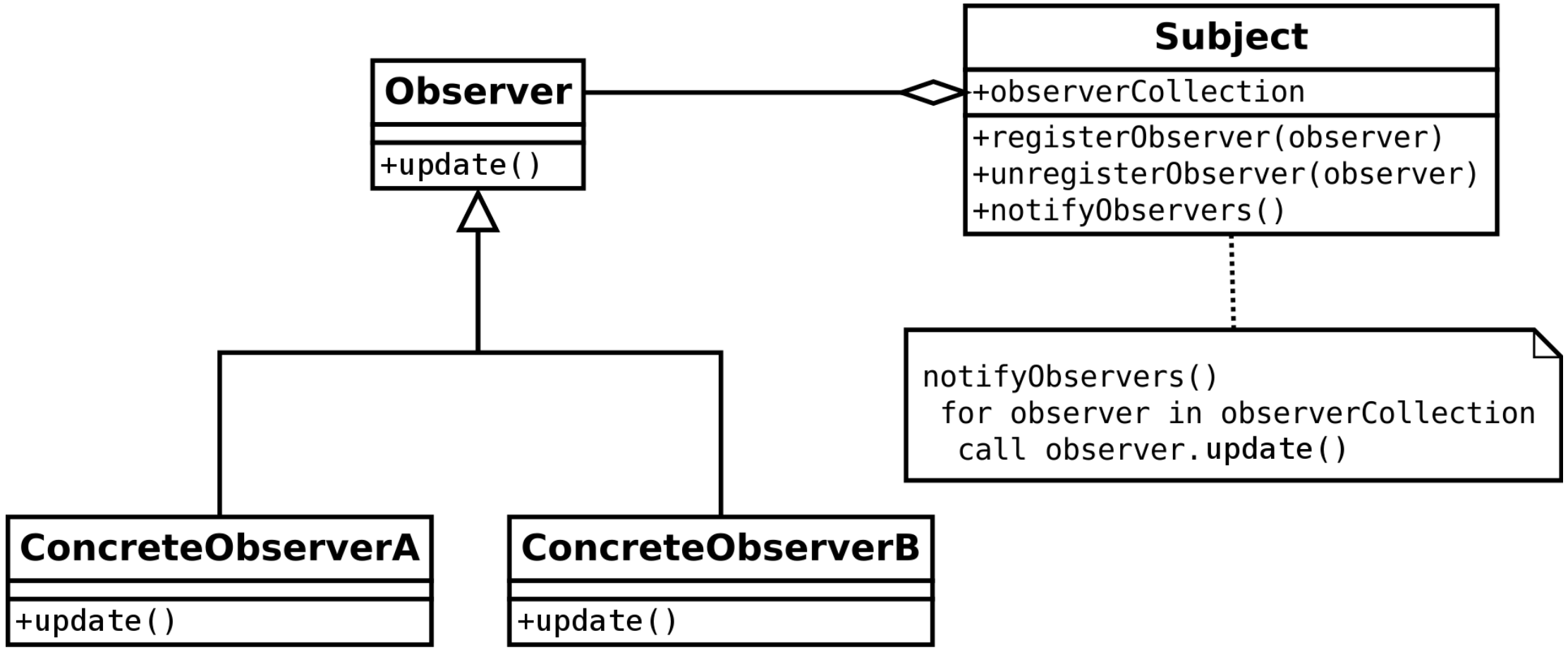
# Once upon a time

- Design Patterns (1994 - Gang Of Four)
  - Iterator Pattern (Behavioral Design Pattern)
    - Decouble data from alogrithms



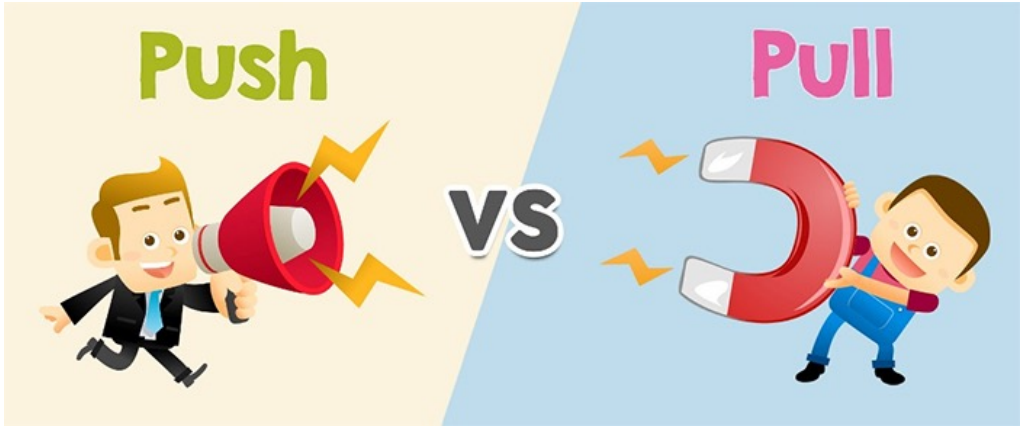

```
class Iterable {
 [Symbol.iterator]() {
   …
 }
}

const iterable = new Iterable();
for (const item of iterable) {
  …
}
```

# Observer pattern (Behavioral DP)



# Pull vs Push Architecure (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 Architecure (II)

| | Producer | Consumer |
|---|---|---|
| **Pull** | **Passive:** produces data when requested. | **Active:** decides when data is requested. |
| **Push** | **Active:** produces data at its own pace. | **Passive:** 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 | Mulitple items |
|---|---|---|
| synchronous / Pull | Function | Iterable |
| asynchronous / Push | Promise / async\|await | ? |

# Concurrency (II)

| | Single items | Mulitple items |
|---|---|---|
| synchronous / Pull | Function | Iterable (Array) |
| asynchronous / Push | Promise / async\|await | **Observable** |

# 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 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

# Why Observables?

Asynchronous operations

Interactive (reactive) behavior

# What are observables?

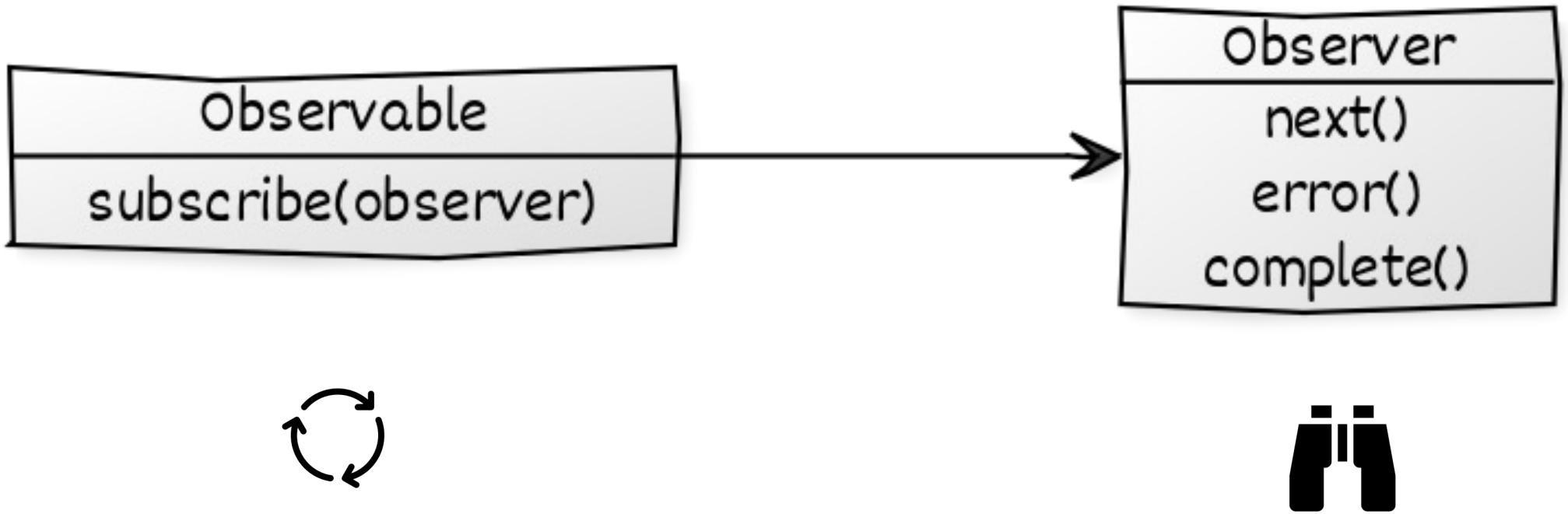
- Represents (asynchronous) data that is published over time
- A collection of values over any amount of time
  - 0..N values could be emitted
- Cancelable
- Lazy
- Operator support

# Observable (asynchornous) data streams?

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


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**

# Observer

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

**Observer**

Option with multiple parameters was deprecated in 6.4!

# Observer

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

# 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();

});
```

Sync/Async, Event-driven

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

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

# Creation Operators (Factories)

[https://www.learnrxjs.io]

fromEvent

of

throwError

interval

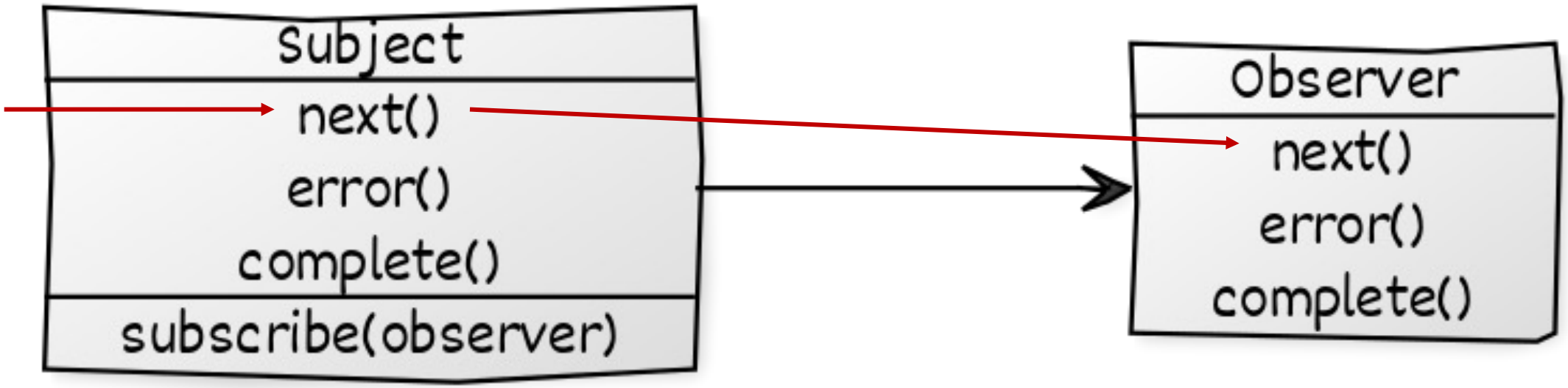
timer

# Subjects

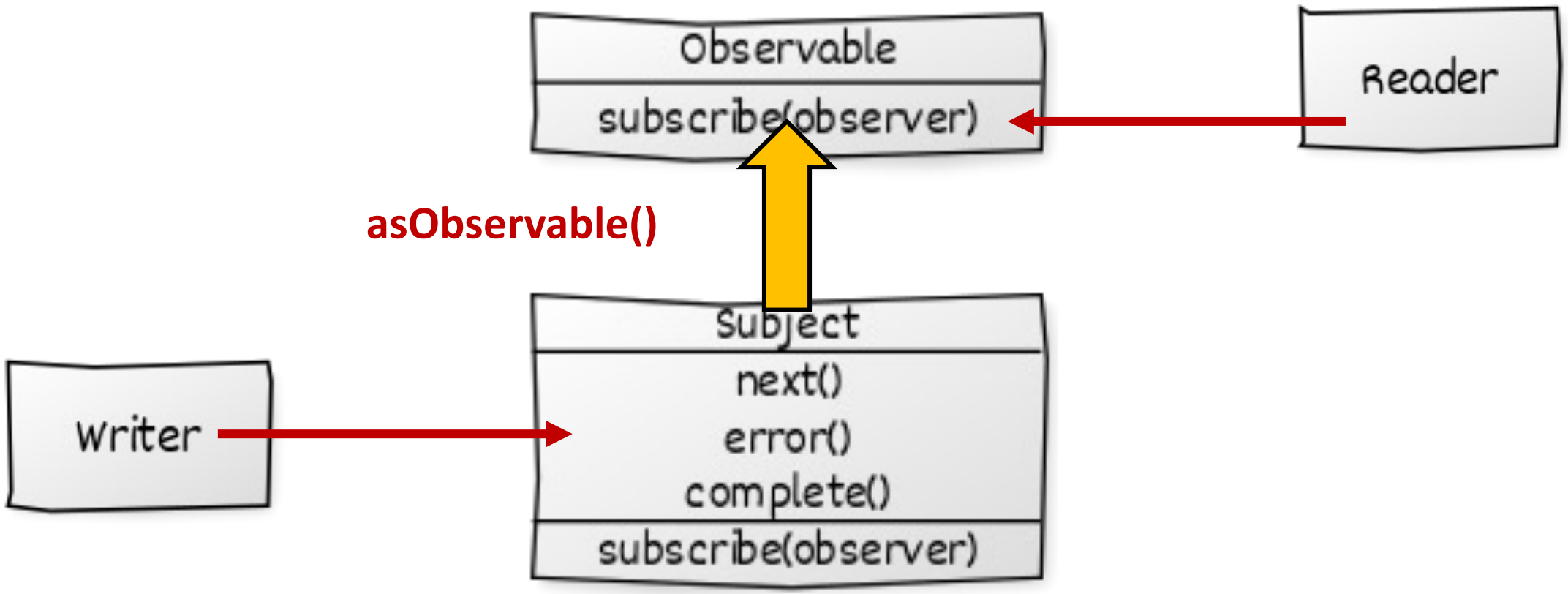
ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

SOFTWARE
ARCHITECT

# Subjects: Special Observables



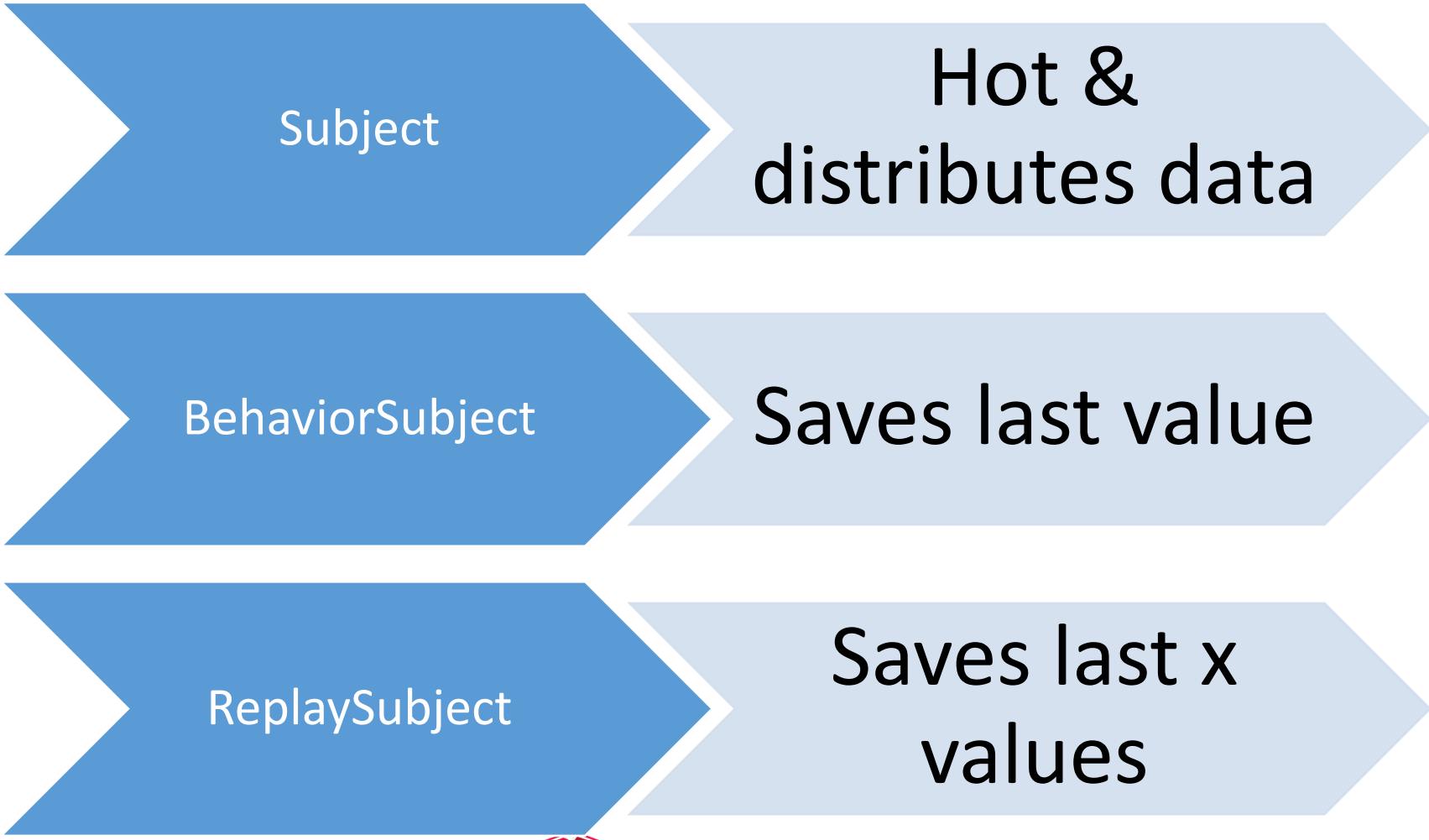
# Convert Subject into Observable



# asObservable

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

[…]
this.observable.subscribe(…)

[…]
this.subject.next(…)
```

# Subjects

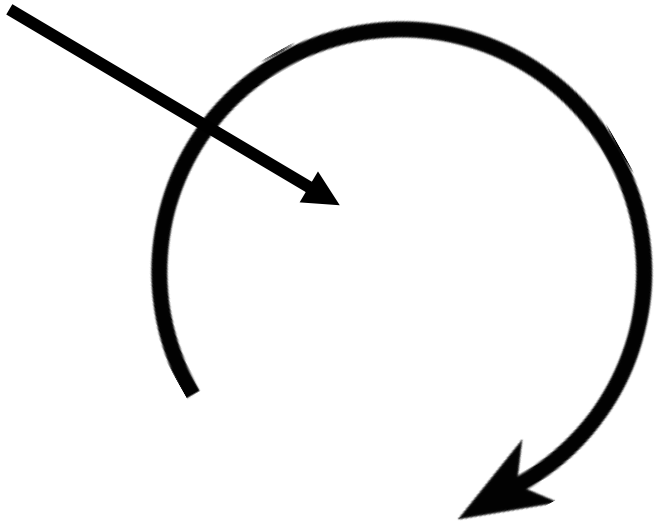


# Eventing with Subject

```
const sub = new Subject<Flight>();

sub.subscribe((flight) => console.debug(flight));

sub.next({ id: 1, ...})
```

# Subjects

Data/Notification



Subject

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

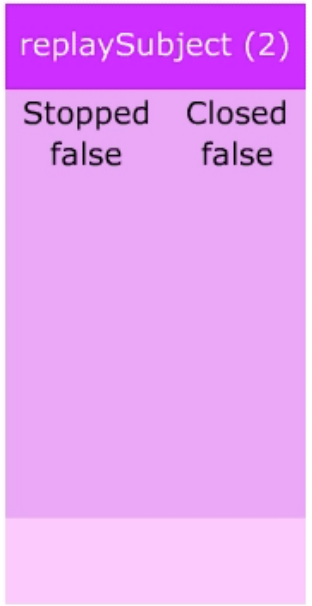
Observer

# 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);
```



# DEMO: Subjects

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(…);

**last operator!**

- Implicitly with async-Pipe in Angular

  {{ observable$ | **async** }}

- Automatic by Angular
  - Angular Router Params

# DEMO: Closing Subscription

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

Default

## Hot

- Multicast
- Eager: Sender starts without subscriptions

# 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(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

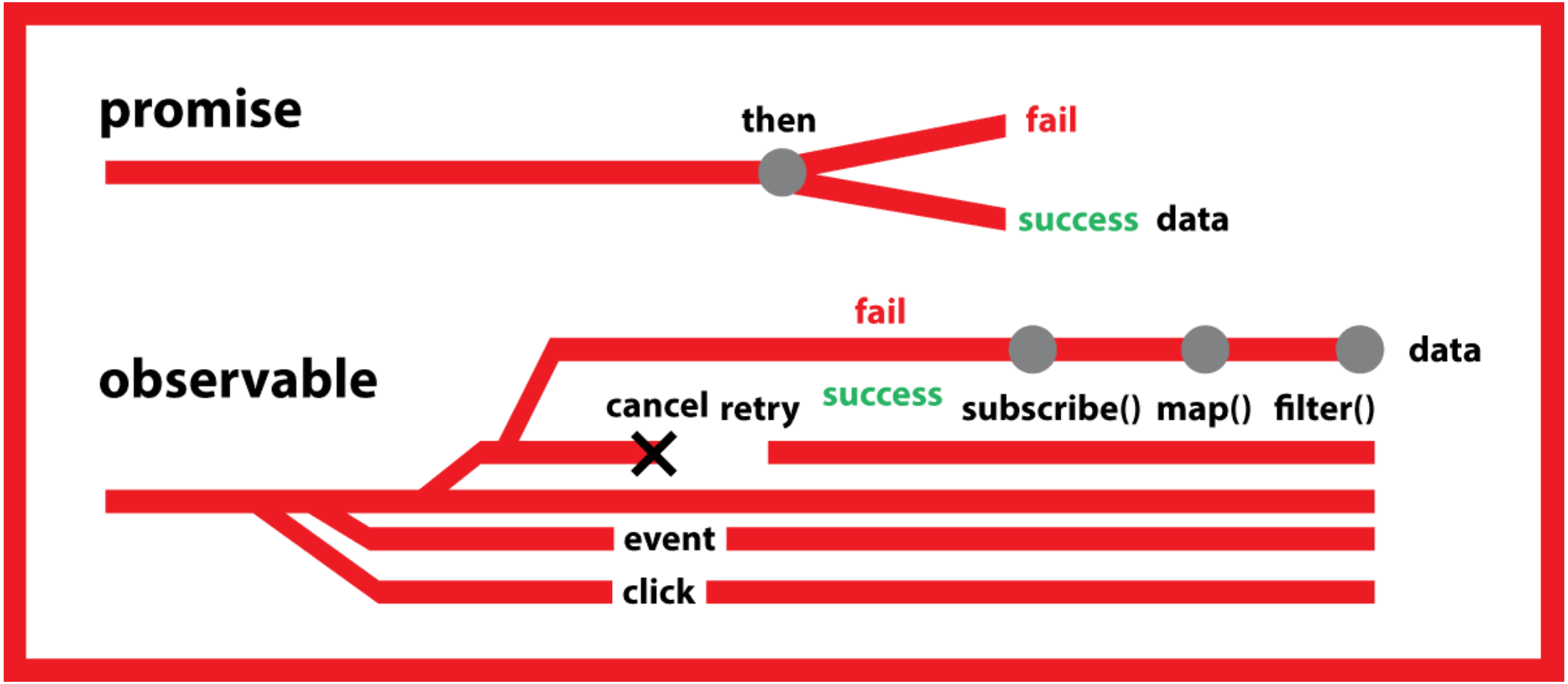
SOFTWARE
ARCHITECT

# 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, **one or multiple** values over time. | Emit a **single** value at a time. |
| **Lazy**: they're not executed until we subscribe using the subscribe() method. | **Eager**: execute immediately after creation. |
| Subscriptions are **cancellable** using the unsubscribe() method, which stops the listener from receiving further values. | Are **not cancellable**. |
| **RxJS** provides a **ton of functionality** 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