PRODYNA

# Reactive Programming With RxJS

TECH Workshop

Tobias Münch

2023

# Reactive Programming === just another buzzword?

# Reactive Programming !== just a another buzzword

# Agenda & Schedule

## Beginner 9h-12h

- Observable vs Promise
- The 3 kinds of Subjects
- Async Mappers
- Reactive combining
- Hands-on using `ReplaySubject`, `take`, `switchMap`, `forkJoin`, `combineLatest`

## Advanced 13h - 15h

- Theory: Paradigm & Classifications
- Side Effects, 3-Channel-Model, Error Handling
- Hands-on using `tap`, `catchError`, `finalize`

## Expert 15h-17h

- RxJS clean code
- Multicast vs Unicast
- Hot vs Cold Observables
- Writing custom pipe operators
- Hands-on using `share`, `shareReplay`

# Objectives

## Practitioner 9h-12h

- We write some medium-complex reactive code in RxJS
- On the way, we learn some fundamental things to think about when programming reactively

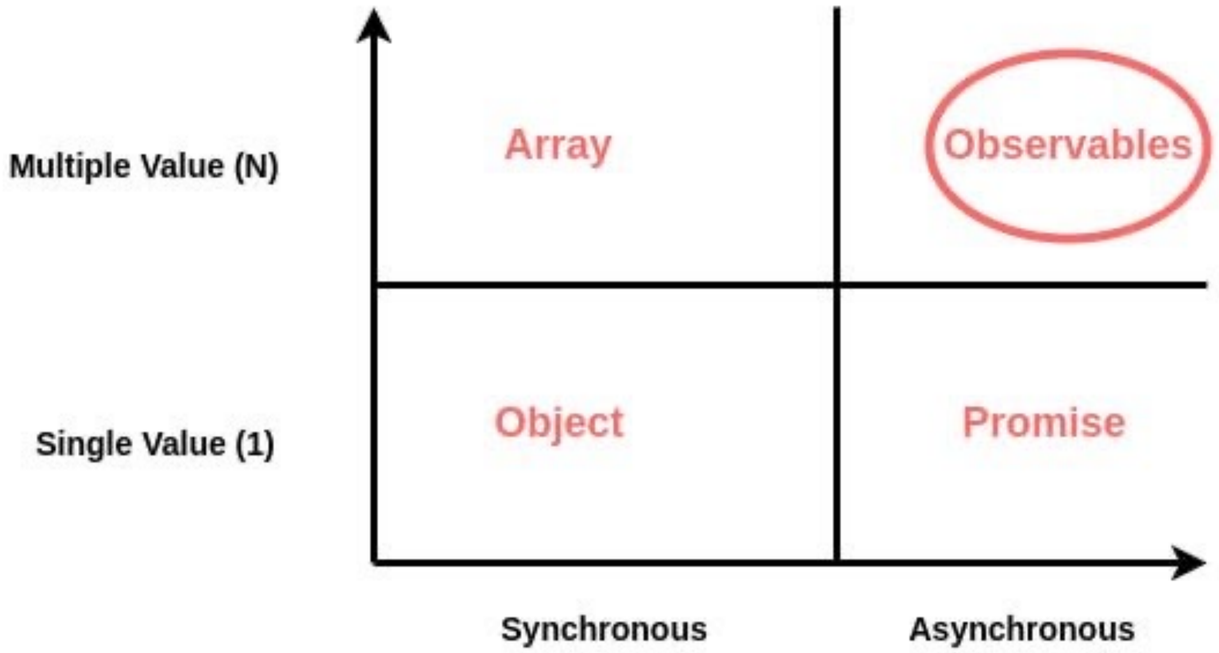
## Engineer 13h - 15h

- We understand some theoretical fundamentals of Reactive Programming
- We talk about the RxJS model
- We use the new knowledge to extend our code

## Lead Engineer 15h-17h

- We clarify many things we used implicitly all the time, but never really realized
- We learn the right words to crush everyone in PR reviews!
- We learn how to write clean reactive code

# Observable vs. Promise



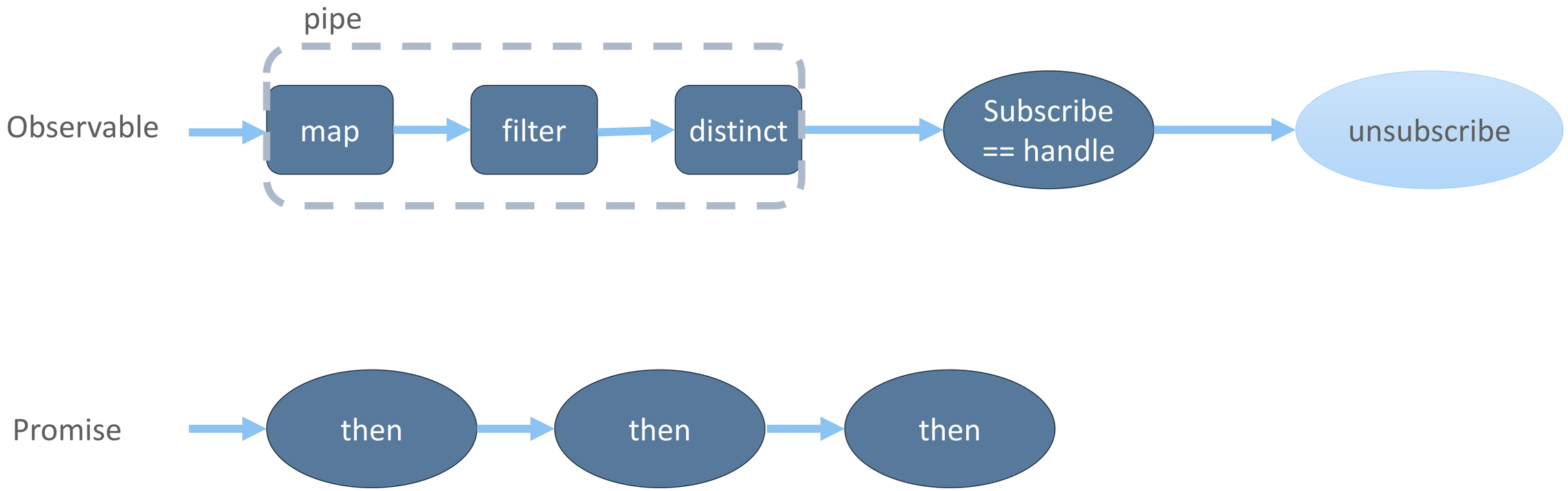
# Observable vs. Promise

- A way to asynchronously react on future event
- Can react on error

- Can react multiple times on success events (“emit”)
- Can be “unsubscribed”, e.g. cancelling a HTTP Request
- Can be “empty”
- Can be altered in many ways via a rich set of operators in a chained & functional way (“piped”)
- Represents a stream of future values

- Can react only on one success event (“resolve”)
- Can only be chained via a simple “then” method
- Compatible with the “async/await” syntax
- Represents one single future value

# Observable vs. Promise



# Observable vs. Promise

- From Promise to Observable? Easy:

```
const observable = from(promise);
```

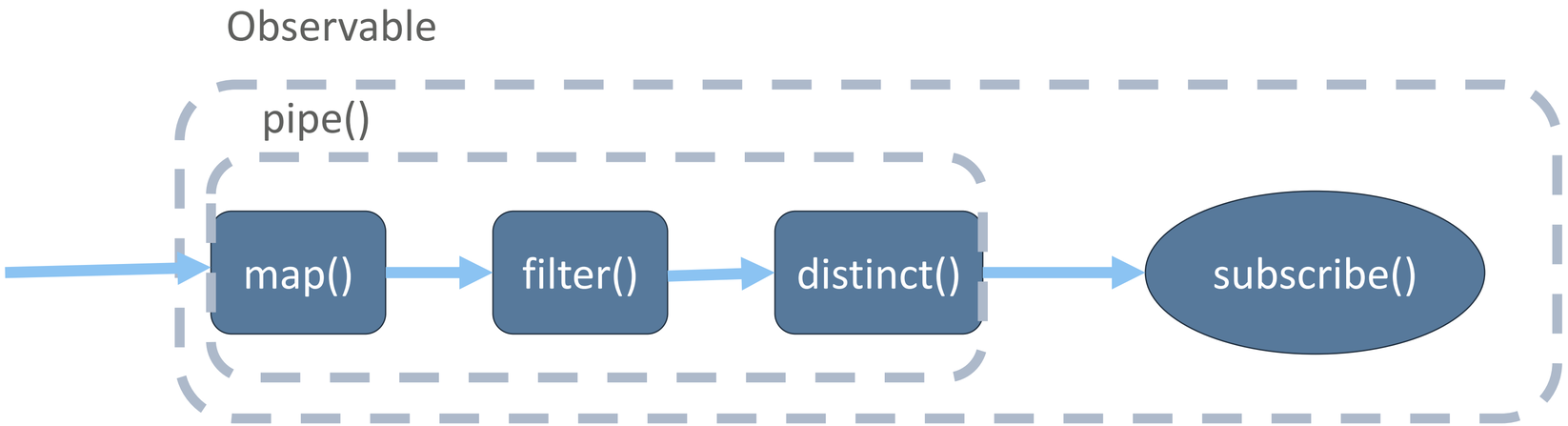
- From Observable to Promise? Depends...
  - take the first emission?
  - take the last emission?

```
const promiseFromFirstValue = firstValueFrom(observable);
const promiseFromLastValue = lastValueFrom(observable);
```

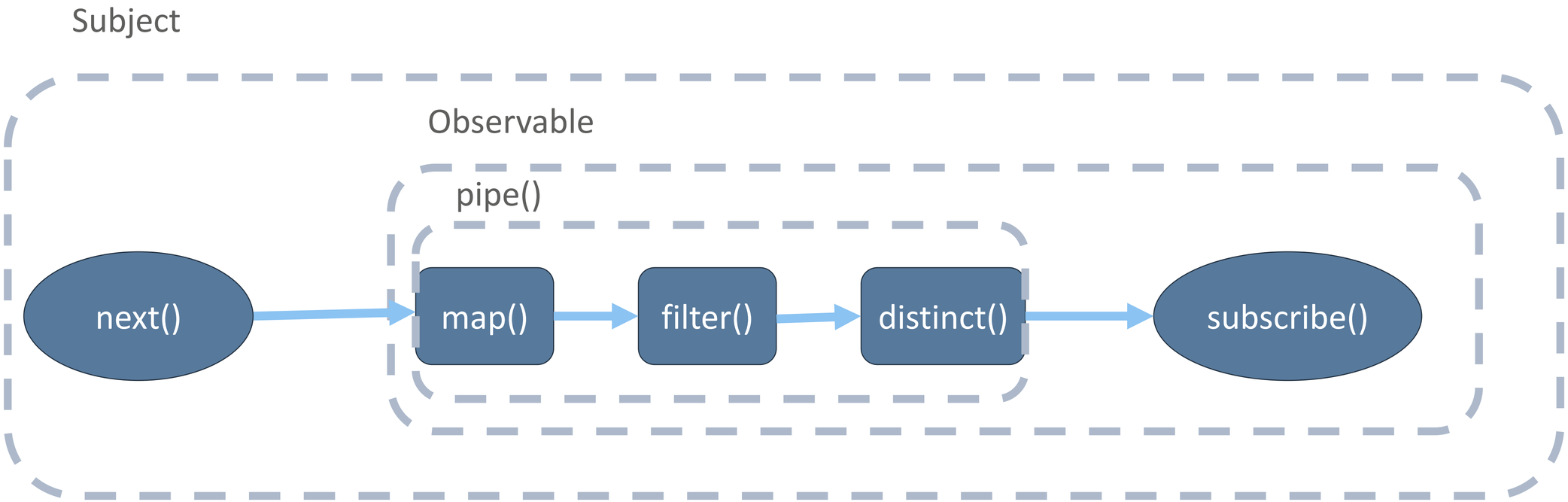
- Observable that emits just once like a promise?
  - RxJS's biggest fail... no type exists
  - limit the emissions in the pipe using take or first

```
const justEmitOnce = observable.pipe(take(1));
// OR (subtle difference not part of this lesson)
const justEmitOnce2 = observable.pipe(first());
```

# Subjects



# Subjects



# The 3 types of Subjects

- A **Subject** has a next() method to “insert” into the stream
  - it is also an Observable (i.e. has subscribe() and pipe() methods)

- A **ReplaySubject** has an internal buffer to keep n items (buffer size n defined by new ReplaySubject(n)
  - when new observers subscribe, the ReplaySubject immediately emits the n items (and as usual new future items)
  - when the buffer size is omitted, Infintiy is used, so never forget!

- A **BehaviorSubject** is a ReplaySubject with buffer size 1, and initial value: new BehaviorSubject(initialVal)
  - it also provides sync access  value / getValue()
  - therefore, it is like a fusion between normal variable and Observable

# Some notes on the Hands-ons

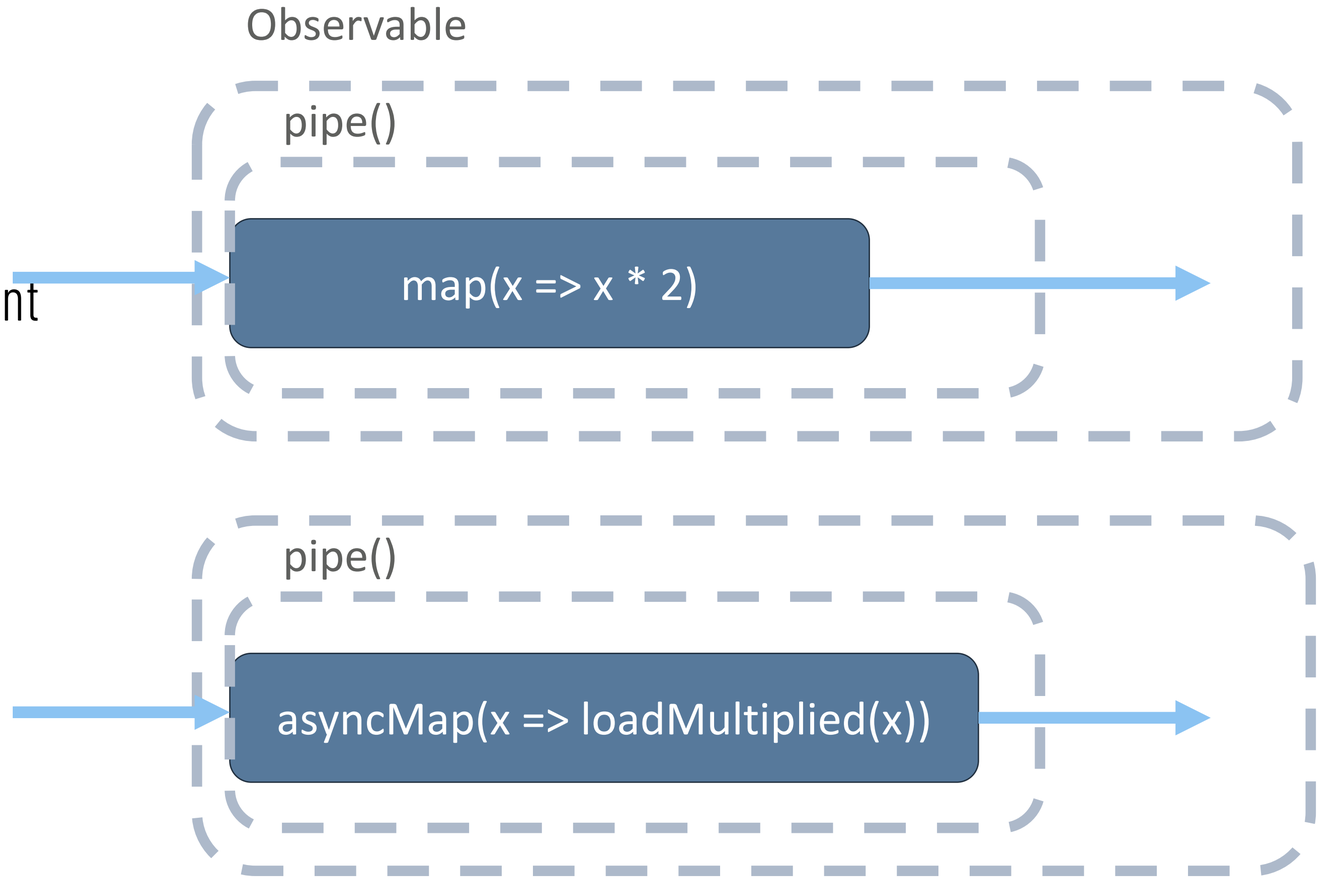
- there is always a clean branch for you
- you only ever edit file /src/app/components/buy-stock/buy-stock.component.ts
- you will use fake loading services that deliver values after some random delay
- at the beginning, there are many detailed TODO comments for you for orientation
- don't peek into the solution commit ;)
- use the quick-fix feature of your IDE to auto-import (VsCode: 'cmd + .', Intellij: 'cmd + enter')
- use the intelliSense, we are in Typescript!
  - however, in the observable pipe(), no autocompletion for the available operators :/ make sure to import from RxJS and not from lodash
- Saving the file (e.g. cmd+s) will run code formatting (prettier) and fix almost all linting (eslint) issues
  - rule of thumb: if your code gets confusing, just save and check if its better now..

# Hands-on 1: Subjects, Subscribing, take(1)

- Branch: git switch hands-on-1-subjects
- Choose an appropriate type of Subject
  - insert the current selection into the subject
  - subscribe and set the existing variable
  - use take(1), subscribe on button click and open google search
- Expected:
  - You see a text when options selected
  - You can click on the icon. A google search for the stock is opened (and you see a log in the console, and you can still further change e.g. the stock amount)
- Note: no worries, the resulting code might not “feel right”... Extra work:
  - Try to achieve the same result without any RxJS
  - Where is the “reactiveness” coming from?

# Async Mappers

- Mappers map one value to another via a "mapping function"
- In Reactive Programming, you might take the current emission and map it to an observable
  - hence, we have an 'async mapper'
  - we have an "outer observable and an "inner observable"
  - note that the inner observable could again emit zero to many values!



# Async Mappers

- Problem: What if the outer observable emits faster than the inner observable completes?
- 4 possible solutions
  - switchMap:
    - simply discard ongoing inner observable, switch to new one
  - exhaustMap:
    - simply stay on ongoing inner observable, discard all new values in the meantime
  - mergeMap:
    - handle up to n inner observables concurrently, merge the result in one stream
  - concatMap:
    - handle only one inner observable, in-order queue all new values and handle subsequently

safe, simple, mostly right choice

safe, but usually weird

- powerful, but you loose the order
- like with ReplaySubjects: if n is omitted, it defaults to Infinity!

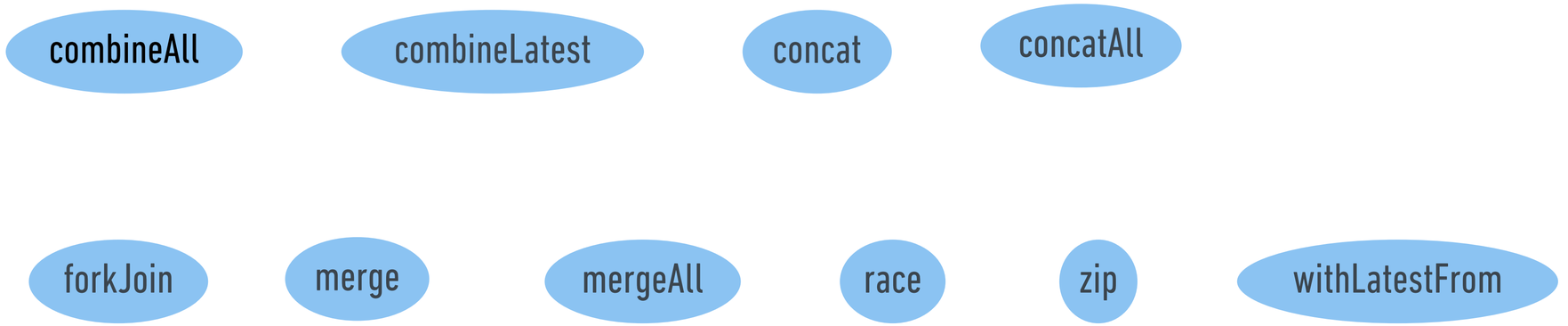
- keep the order
- prone to backpressure

# Hands-on 2: Async Mappers

- Branch: git switch hands-on-2-async-mappers
- Choose an appropriate type of async mapper
  - use the provided service stockPriceDataService to fake-load the price for the selection from SIX
    - Caution: the selection can be null, the service does not handle this for you!
    - Tip: Remember the of() function (maybe Copilot will do it for you..)
  - subscribe and set the existing variable stockPriceData
    - observe its type! its ok to set it to null...
- Expected:
  - You see the price for your selection after some short delay
- Extra work:
  - play around with other async mappers
  - think about clean-code rules and make the async mapper function is simple as possible

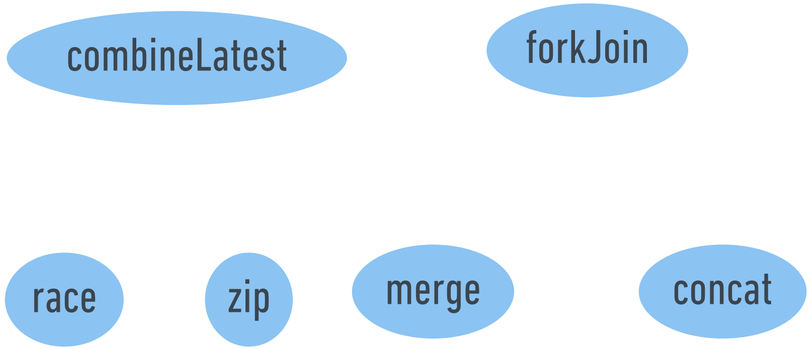
# Reactive combining

- There is a zoo of ways to combine multiple oberservables

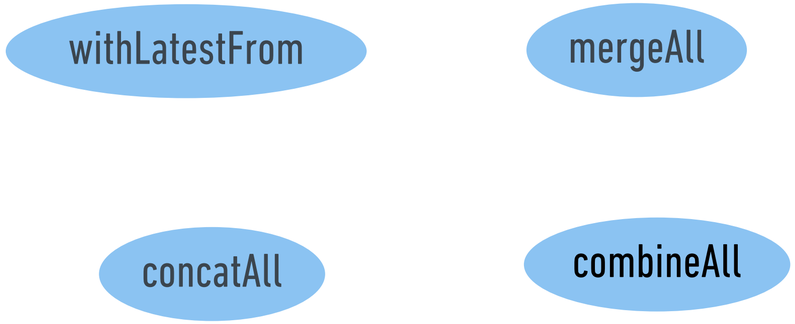


# Reactive combining

- generator (stand alone function)



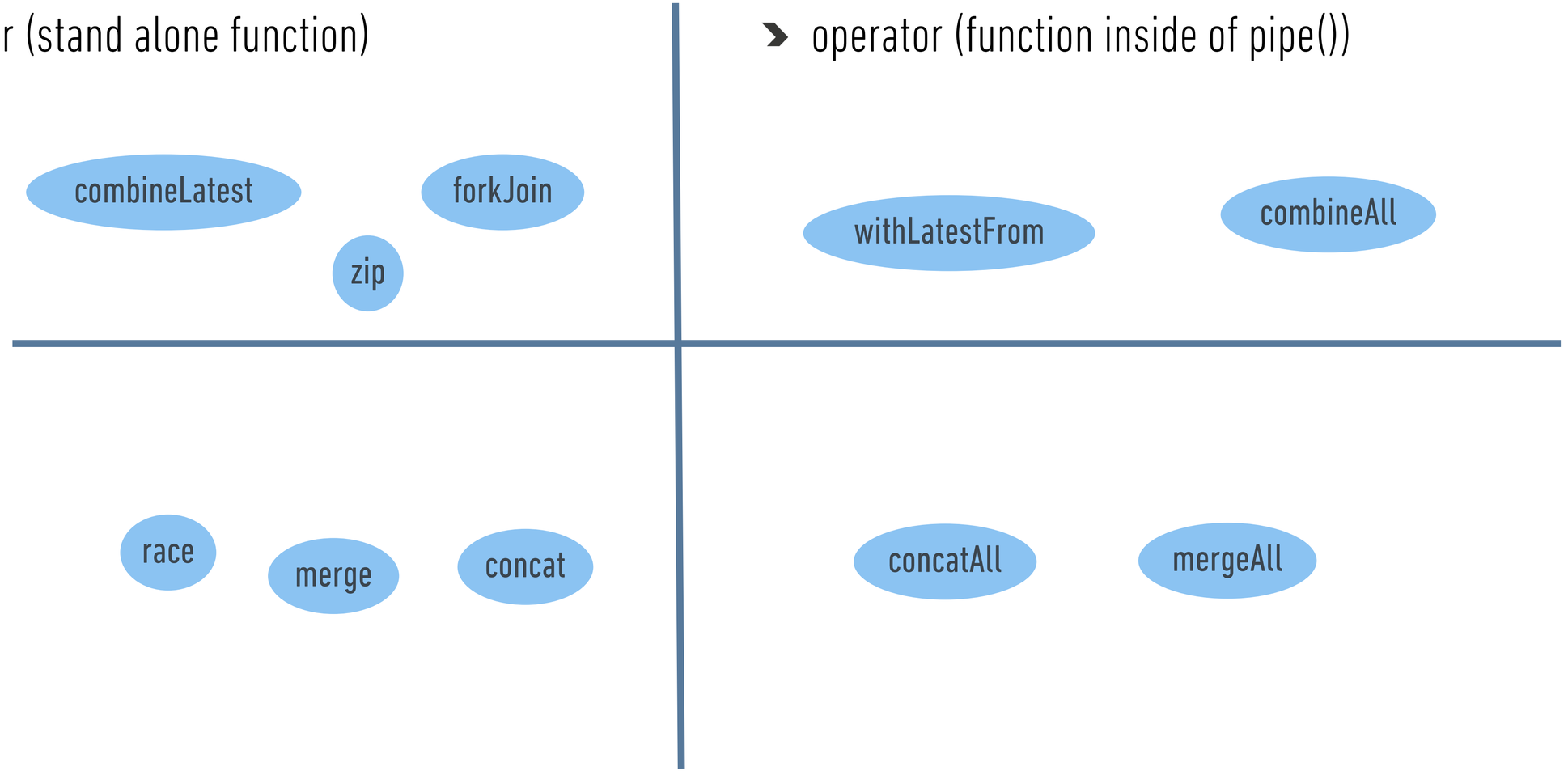

- operator (function inside of pipe())



# Reactive combining

- generator (stand alone function)
- operator (function inside of pipe())

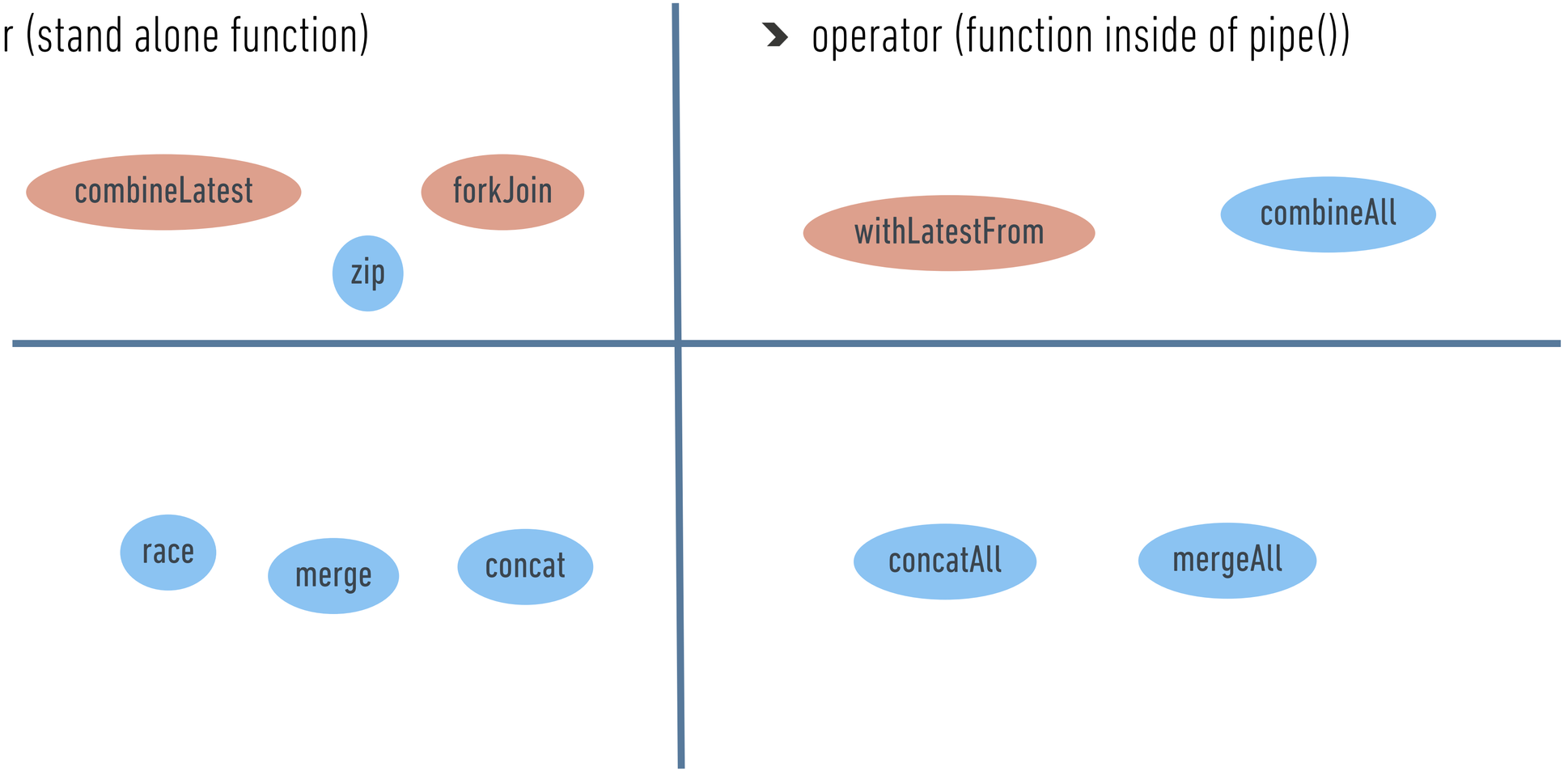
- combine into array
- combine into sequential stream



# Reactive combining

- generator (stand alone function)
- operator (function inside of pipe())

- combine into array
- combine into sequential stream



# Reactive combining

- combineLatest
  - wait until all source observables emit at least one value, then emit whenever any of them change with the latest of each source observables
- forkJoin
  - wait until all source observables emitted and completed, then emit this one single array of values
  - wonderfully simple
- withLatestFrom
  - in a pipe/stream, on every value take the latest value from another observable and add it to the stream

```
// emits 0,1,2 with 10ms delay, like 10ms-0-20ms-1-30ms-2
const obs1 = interval(10).pipe(take(3));
// emits 0,1 with 20ms delay, like 20ms-0-40ms-1
const obs2 = interval(20).pipe(take(2));

combineLatest([obs1, obs2]).subscribe(([val1, val2]) => {
  console.log([val1, val2]);
});
```

```
const obs3 = of('Hello');
const obs4 = of('World');

forkJoin([obs3, obs4]).subscribe(([val1, val2]) => {
  console.log([val1, val2]);
});
```

```
// emits 0,1,2 with 10ms delay, like 10ms-0-20ms-1-30ms-2
const obs5 = interval(10).pipe(take(3));
// emits 0,1 with 6ms delay, like 6ms-0-12ms-1
const obs6 = interval(6).pipe(take(2));

obs5.pipe(withLatestFrom(obs6)).subscribe(([val1, val2]) => {
  console.log([val1, val2]);
});
```

# Hands-on 3: Reactive Combining

- Branch: git switch hands-on-3-combining
- Choose appropriate types of combiners to...
  - ...load price on reload button click (the subject that emits on every click is provided)
  - ...additionally load price from Xetra (in parallel)
  - add some logic to always show lowest price
  - Tip: only consider the three combiners we checked in detail
- Expected:
  - You see the price for your selection after some short delay
  - You see that button click reloads data
  - (you probably cant see that price is always lowest, add some logging if you want to check this)
- Extra:
  - why does the reloadSubject have to be a BehaviorSubject?

# Agenda & Schedule

## Beginner 9h-12h

- Observable vs Promise
- The 3 kinds of Subjects
- Async Mappers
- Reactive combining
- Hands-on using `ReplaySubject`, `take`, `switchMap`, `forkJoin`, `combineLatest`

## Advanced 13h - 15h

- Theory: Paradigm & Classifications
- Side Effects, 3-Channel-Model, Error Handling
- Hands-on using `tap`, `catchError`, `finalize`

## Expert 15h-17h

- RxJS clean code
- Multicast vs Unicast
- Hot vs Cold Observables
- Writing custom pipe operators
- Hands-on using `share`, `shareReplay`

# The Dependency View

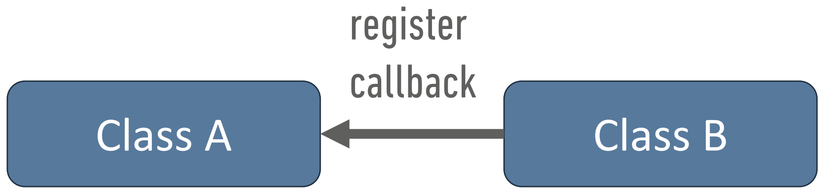
- Normal Programming: something happens in Class A
  - call function in Class B
  - dependency: Class A knows Class B


- Reactive Programming: something happens in Class A
  - react by executing “some function” that B provides
  - dependency: Class B know Class A

# The Dependency View

- Normal Programming: something happens in Class A
  - call function in Class B
  - **by means of method call**
  - dependency: Class A knows Class B

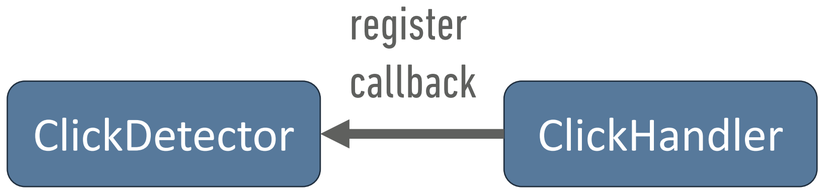

- Reactive Programming: something happens in Class A
  - react by executing “some function” that B provides
  - **by means of previously registering a callback**
  - dependency: Class B knows Class A



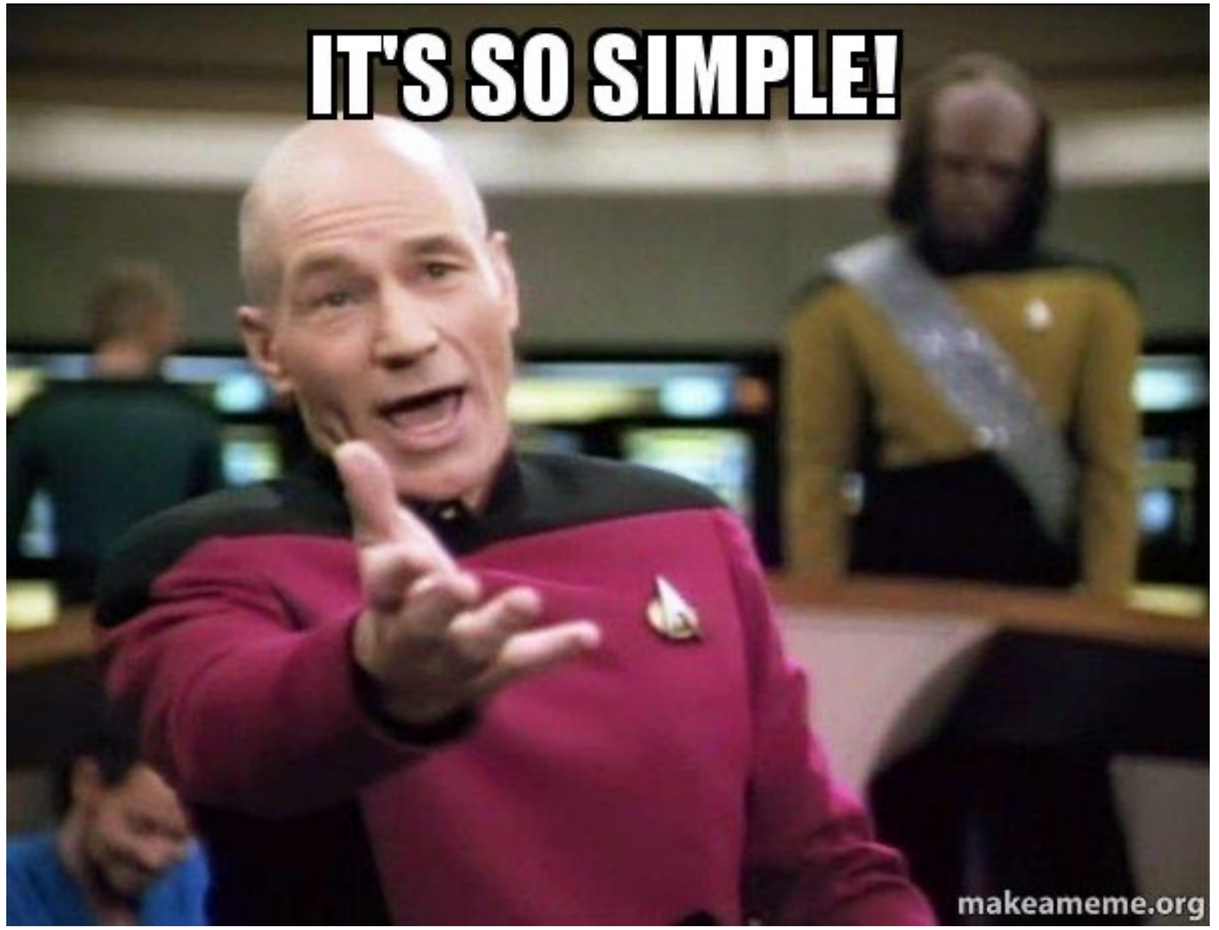
# The Dependency View

- Normal Programming: ClickDetector detects a mouse click
  - call method “onMouseClicked” of the ClickHandler
  - ClickDetector needs an instance of ClickHandler


- Reactive Programming: ClickDetector detects a mouse click
  - execute a function
  - the function comes from a “registerCallback” method of ClickDetector, which the ClickHandler has previously called
  - ClickHandler needs an instance of ClickDetector



# So.. Reactive Programming === Callbacks?



# Hm.. maybe there is more...

## Reactive programming

10 languages

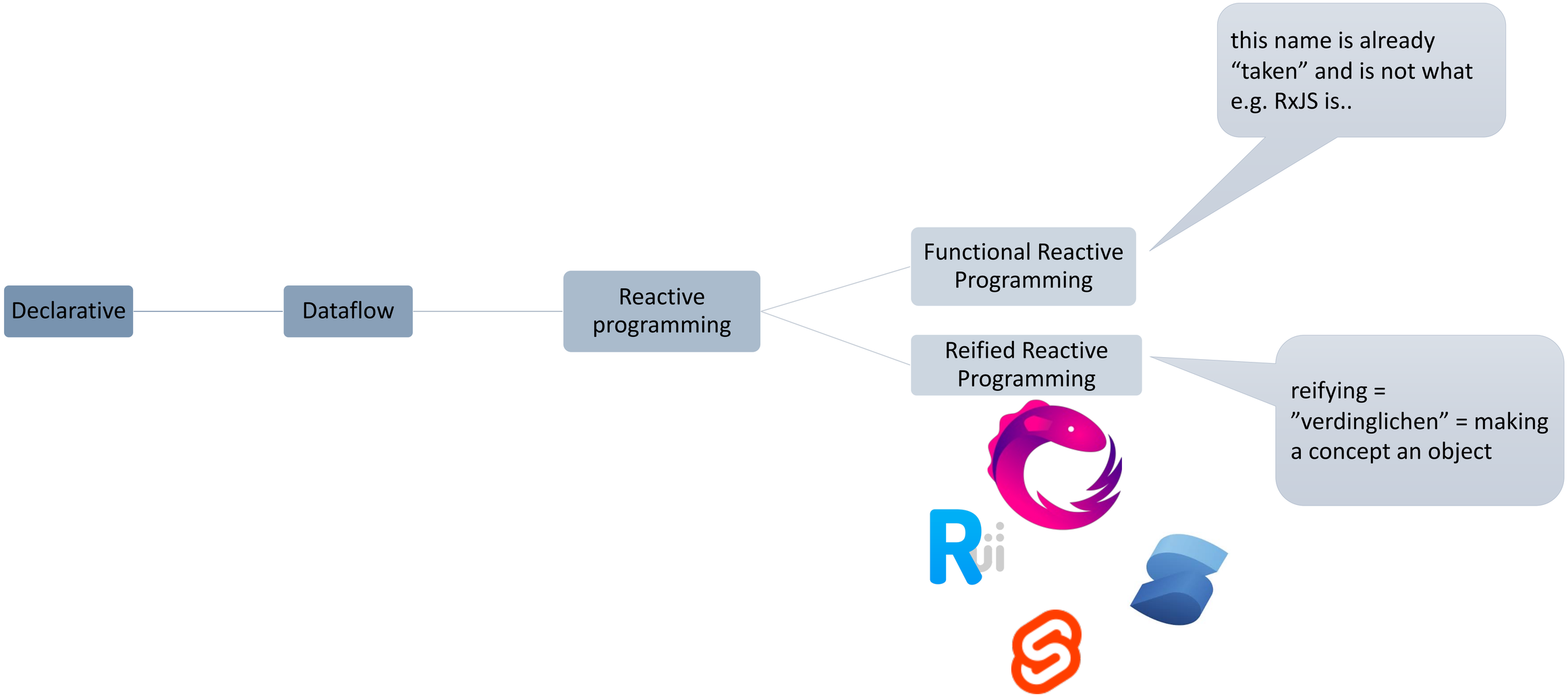
Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

**This article has multiple issues.** Please help **improve it** or discuss these issues on the **talk page**. *(Learn how and when to remove these template messages)* [hide]

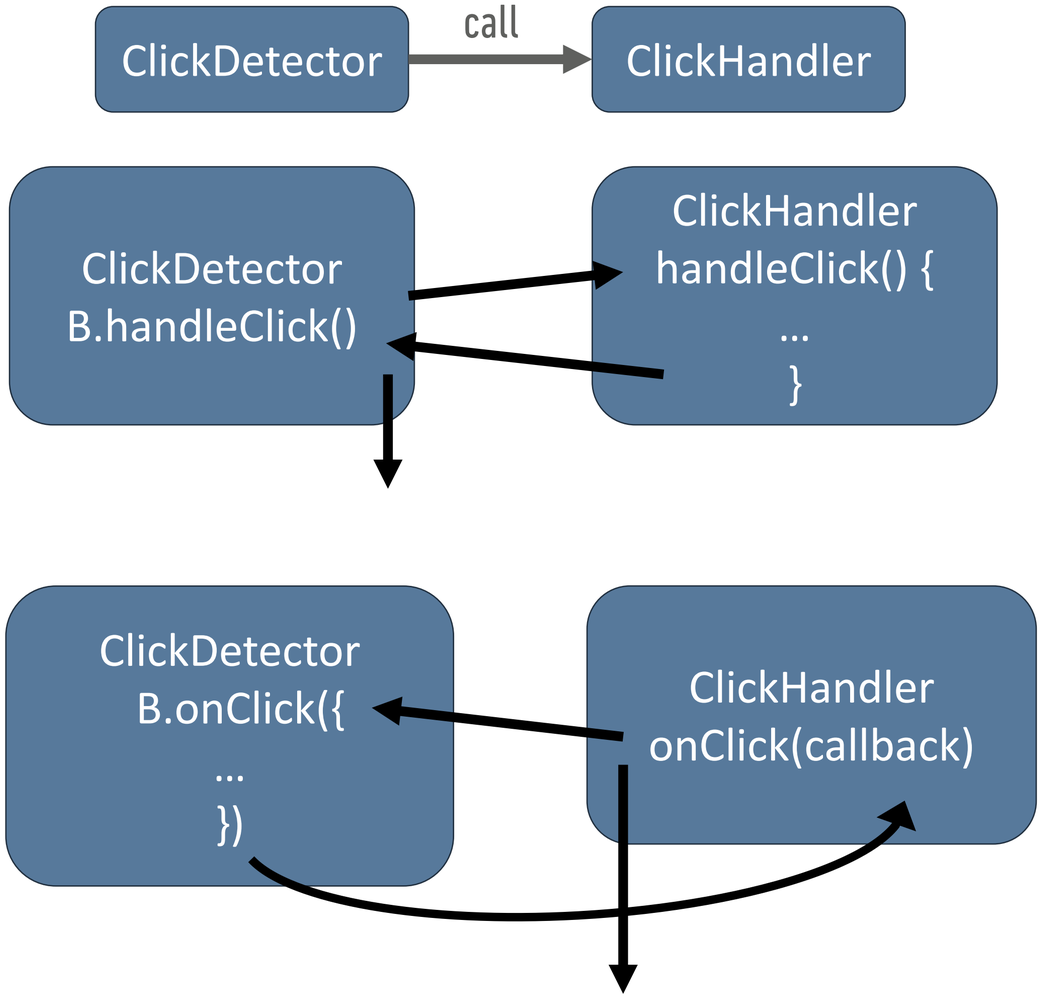
- This article may **require cleanup** to meet Wikipedia's quality standards. The specific problem is: **Some text is too verbose or poorly written. Further, the terminology in these cases is not defined for the reader.** *(November 2018)*
- This article includes a list of general references, but **it lacks sufficient corresponding inline citations**. *(October 2016)*

# Taxonomy of Reactive Programming



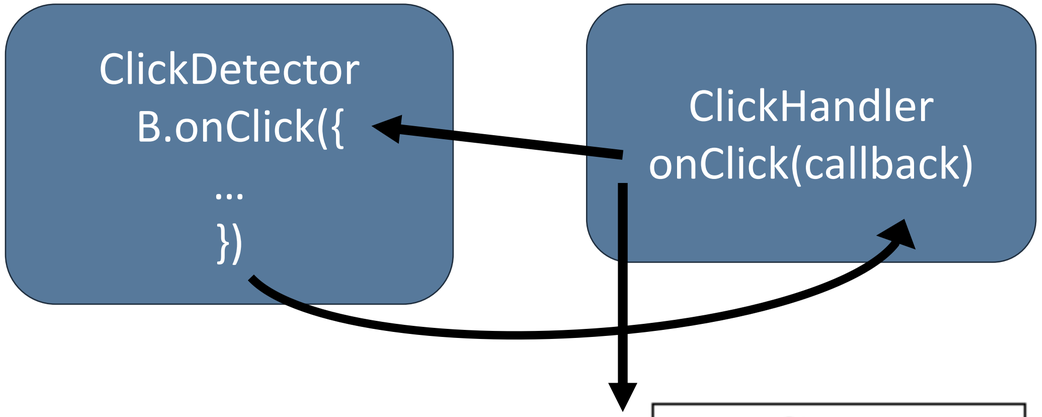
# The Program Flow View

- Normal Programming: ClickDetector detects a mouse click
  - call method in ClickHandler
  - **synchronous**
- Reactive Programming: ClickDetector detects a mouse click
  - execute callback of ClickHandler
  - which is “some time” after ClickHandler registered the callback
  - resulting (usually) for ClickHandler in an **asynchronous program flow!**
- Results in concurrency
- Multiple ”happenings” and multiple callbacks -> “stream”

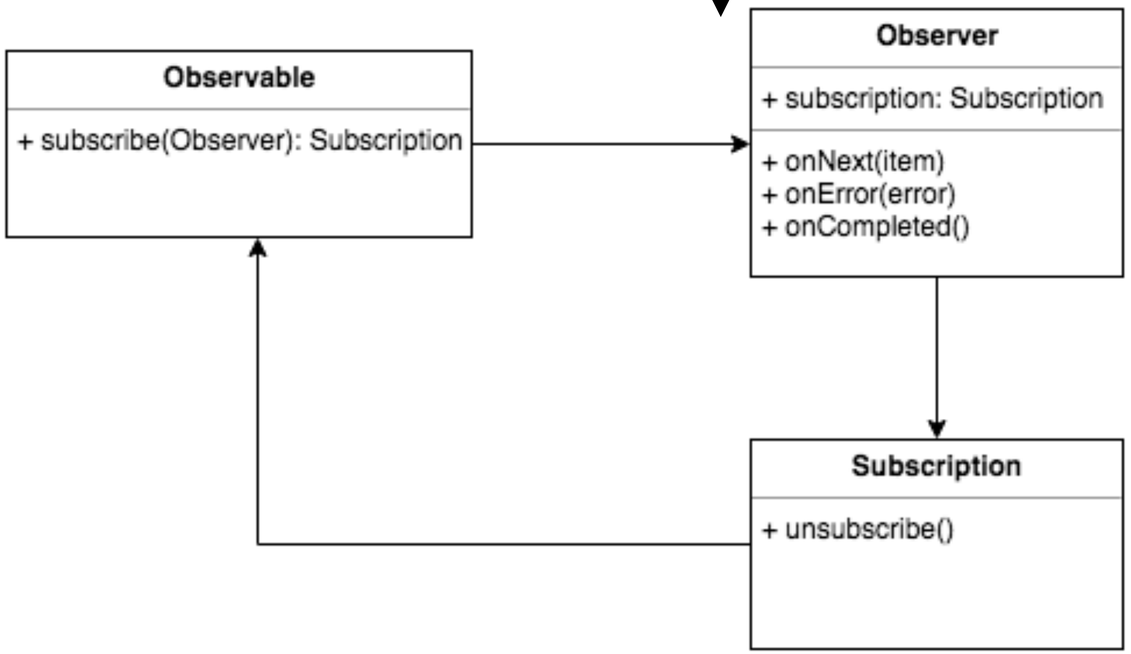


# Observable Pattern

- notice: the async call “onClick(callback) returns nothing




- Observable pattern solves this problem
  - async call returns “Observable”
  - Observable has method “subscribe()” to register an Observer
    - which are several callbacks
  - which returns a Subscription
  - which can be used to un-register the callbacks



# Reactivity on variable level

- Normal Programming: re-assigning property *b*
  - does not change any previous evaluations
  - what is *a* now?

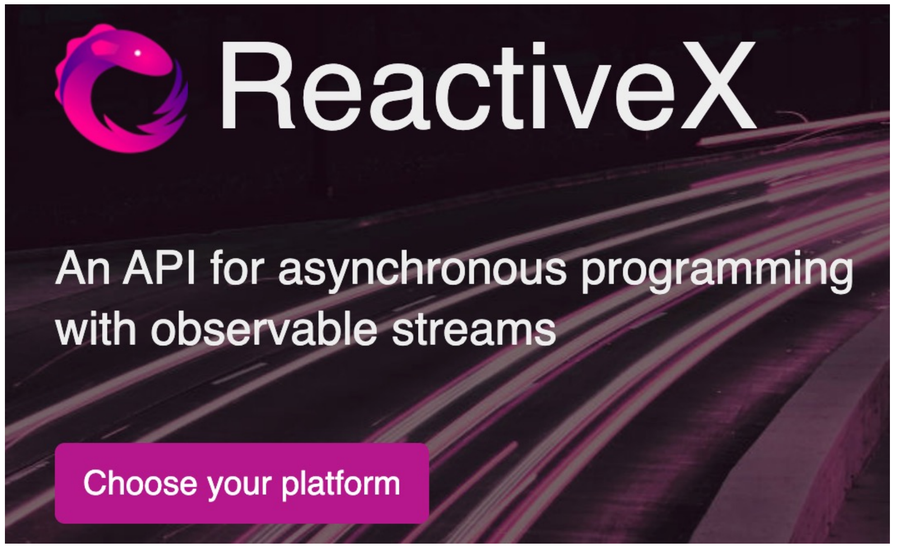
```
var b = 1
var c = 2
var a = b + c
b = 10
```

- Reactive Programming: re-assigning property *b*
  - changes previous evaluation
  - “propagation of change”
  - ‘$=‘ as “reactive assignment”
  - what is *a* now?

```
var b = 1
var c = 2
var a $= b + c
b = 10
```

# Reactive Extensions

- is a “library blueprint” of “reified reactive programming”
- implemented in several languages
- created by Microsoft, released 2011




Popular repositories

**RxJava** Public
RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM.
Java 47.2k 7.7k

**rxjs** Public
A reactive programming library for JavaScript
TypeScript 29.4k 3k

**RxSwift** Public
Reactive Programming in Swift
Swift 23.6k 4.1k

**RxAndroid** Public
RxJava bindings for Android
Java 19.8k 3k

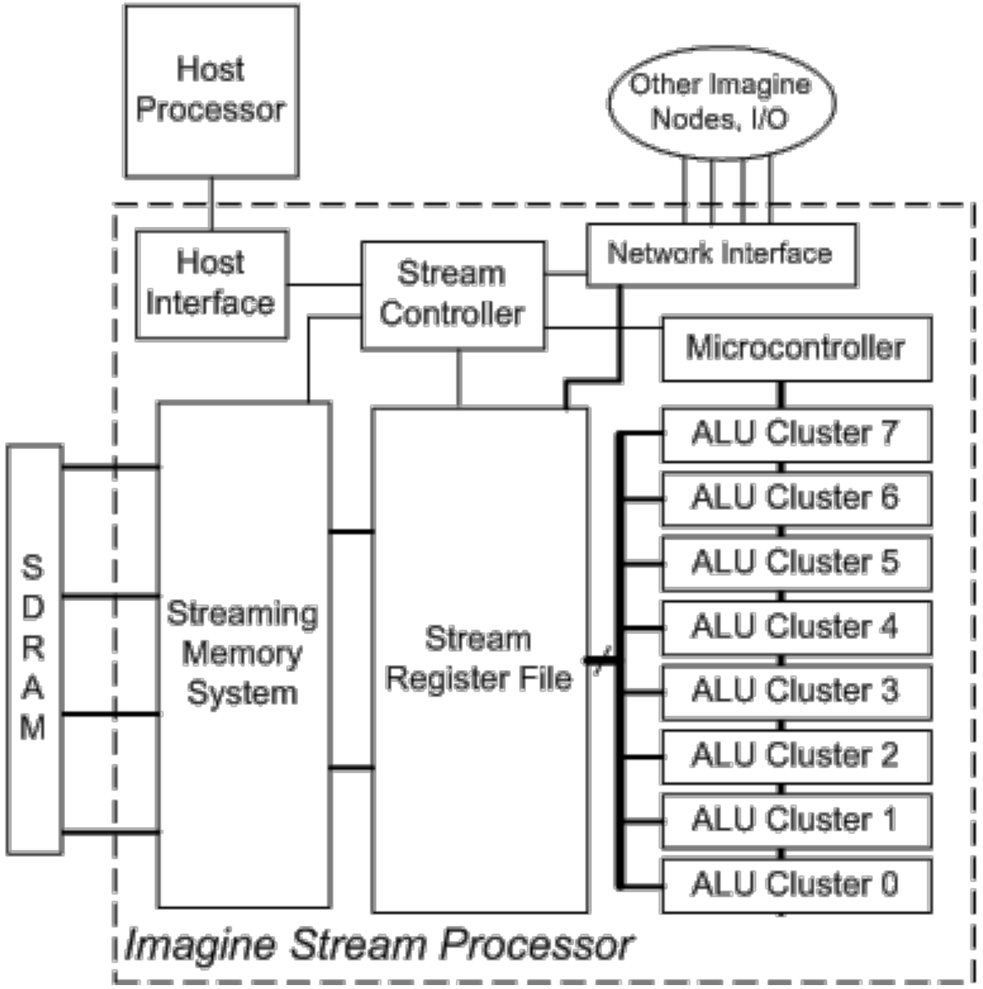
**RxKotlin** Public
RxJava bindings for Kotlin
Kotlin 7k 471

**RxGo** Public
Reactive Extensions for the Go language.
Go 4.7k 341

# Summary: Reactive Programming...

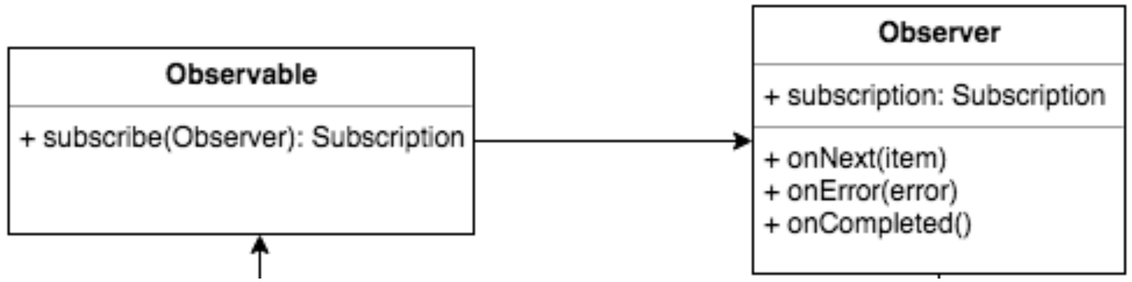
- uses **callbacks** to handle async program flow
- **inverts the dependencies** of components in the program
- is aware of multiple value changes, i.e. **data streams**
- when its “reified reactive programming”, additionally there are
  - objects that represent ”changeable” or “reactive” variables, e.g. via **Observables**
  - means to express a “propagation of change” through multiple possibly **combined reactive variables**

# RxJS – “The stream model”



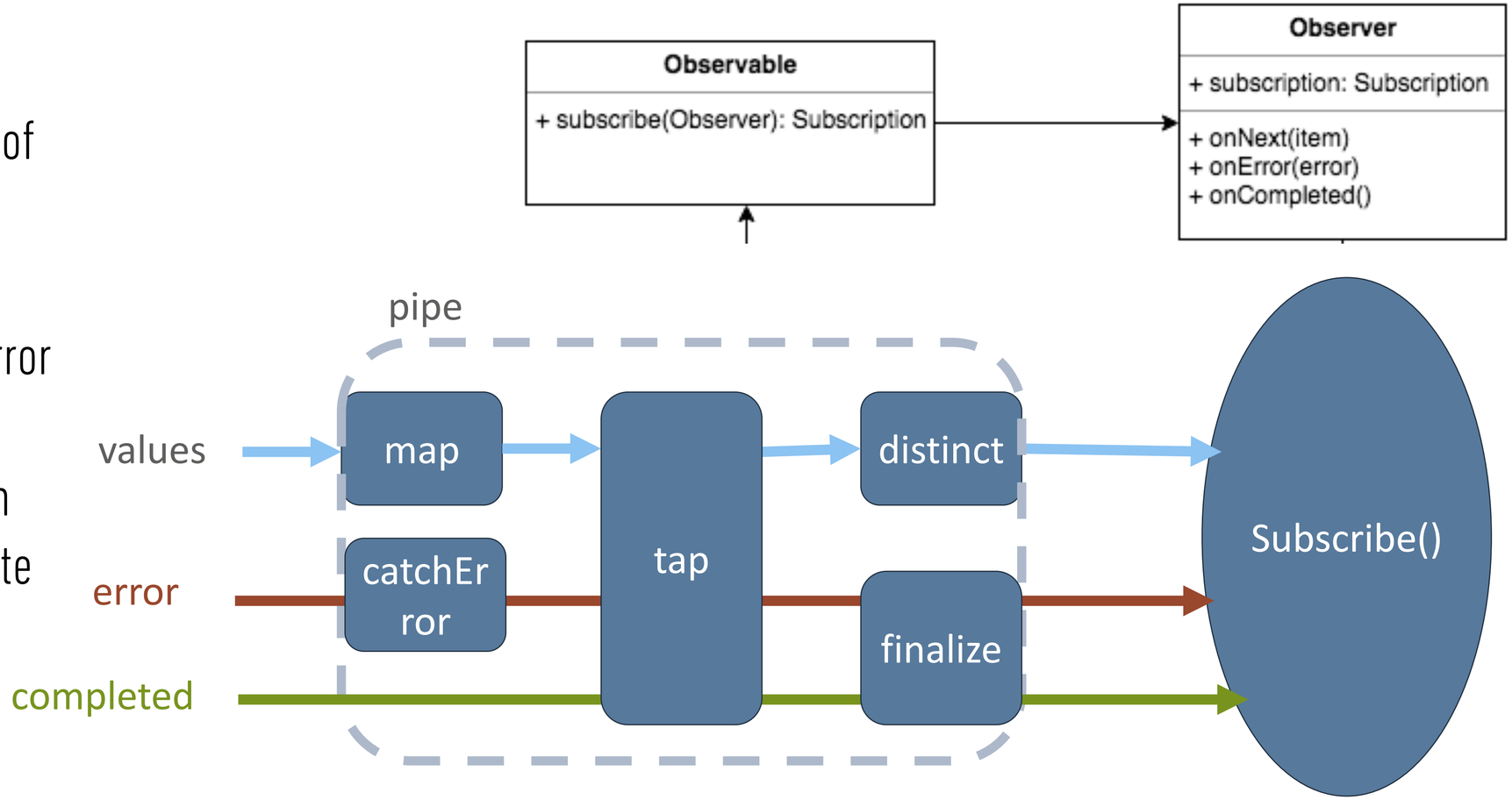
# Rx – “The stream model”

- 3 types of possible items
  - value/next
  - error
  - complete
- The observer has 3 optional callbacks to react on each type



# RxJS – “The stream model”

- operators in the pipe can react on some or also all of them
  - tap() on all of them
  - catchError() only on error
  - finalize() has one callback that reacts on both error and complete



# RxJS – “The stream model”

```
of('something')
  .pipe(
    tap({
      next: val => {
        console.log('next in tap: ', val);
      },
      error: err => {
        console.log('error in tap: ', err);
      },
      complete: () => {
        console.log('completed in tap');
      },
    }),
    finalize(() => {
      console.log('finalize');
    }),
    catchError(err => {
      console.log('catchError');
      return of('fallback');
    })
  )
```

```
  .subscribe({
    next: val => {
      console.log('next in subscribe: ', val);
    },
    error: err => {
      console.log('error in subscribe: ', err);
    },
    complete: () => {
      console.log('completed in subscribe');
    },
  });
```

# RxJS – “The stream model”

- How to insert in the channels?
  - for Subjects:
    - next()
    - error()
    - complete()
  - for Observables:
    - of(value)
    - throwError()
    - EMPTY (== of())

```
const someSubject = new Subject<string>();
someSubject.next('some value');
someSubject.error('some error');
someSubject.complete();
```

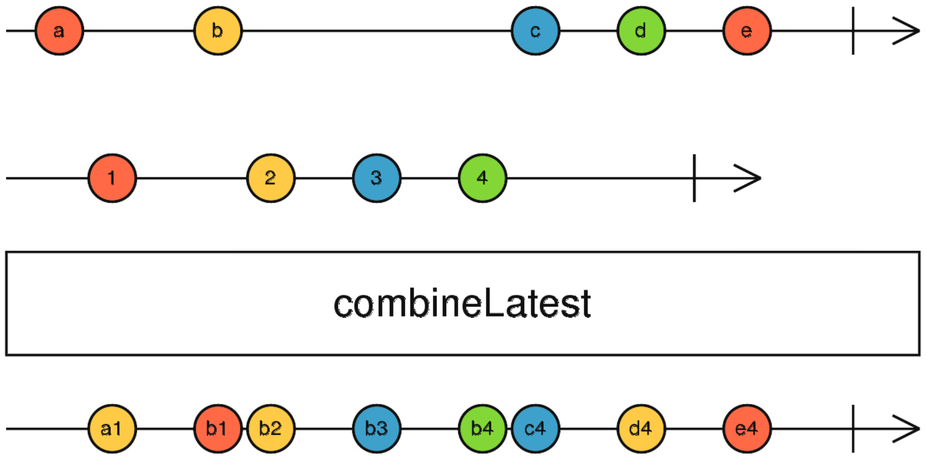
```
of('some value').subscribe({
  next: val => {
    console.log('next: ', val);
  },
  error: err => {
    console.log('error: ', err);
  },
  complete: () => {
    console.log('completed');
  },
});
```

```
throwError(() => 'some error').subscribe({
  next: val => {
    console.log('next: ', val);
  },
  error: err => {
    console.log('error: ', err);
  },
  complete: () => {
    console.log('completed');
  },
});
```

```
EMPTY.subscribe({
  next: val => {
    console.log('next: ', val);
  },
  error: err => {
    console.log('error: ', err);
  },
  complete: () => {
    console.log('completed');
  },
});
```

# RxJS – “The stream model”

- The Rx stream model is also often depected as "marbles”
  - e.g. in “marbles diagram”
  - or in testing libraries




```
// This test runs synchronously.
it('generates the stream correctly', () => {
  testScheduler.run((helpers) => {
    const { cold, time, expectObservable, expectSubscriptions } = helpers;
    const e1 = cold(' -a--b--c---|');
    const e1subs = '  ^----------!';
    const t = time('   ---|       '); // t = 3
    const expected = '-a-----c---|';

    expectObservable(e1.pipe(throttleTime(t))).toBe(expected);
    expectSubscriptions(e1.subscriptions).toBe(e1subs);
  });
});
```

# Catching errors

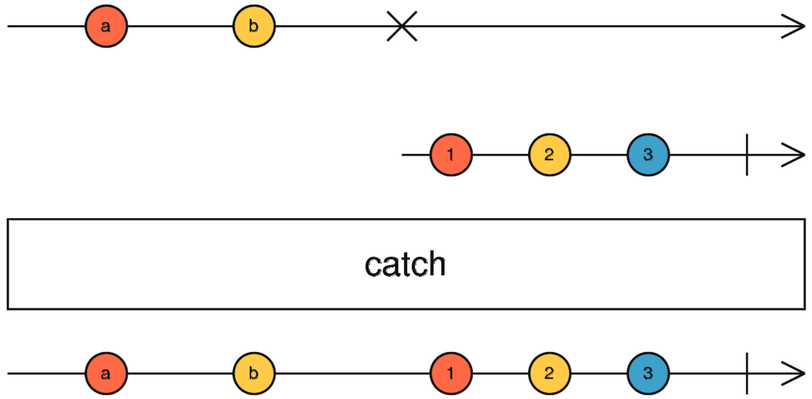
- the Observable might want to communicate to the observer (i.e. the callbacks in the subscribe) that an error occurred
  - e.g. http error
  - or in the pipe() an error occurred
- in this case, this error is “transported” in the “error channel”, and the error callback in the observer (e.g. in the callback in the subscribe) is executed
- **Attention! After an error, an observable always completes! There is no built-in recovery!**

```
type Person = {
  name: string;
  address: { city: string };
};
const max: Person = { name: 'Max' } as Person;

of(max)
  .pipe(map(person => person.address.city))
  .subscribe({
    next: city => console.log('Max lives in', city),
    error: err => console.log('shoot, error!', err),
  });
```

# Catching errors

- catchError() does 2 things
  - executing callback on error
  - replacing the downstream observable with a new one
    - in practice, often of(null)
    - returning EMPTY might also be a good idea




```
type Person = {
  name: string;
  address: { city: string };
};
const max: Person = { name: 'Max' } as Person;

of(max)
  .pipe(
    map(person => person.address.city),
    catchError(err => {
      console.log('shoot, error, gotta handle this...');
      return of(null);
    })
  )
  .subscribe({
    next: city => console.log('Max lives in', city ?? 'unknown city'),
    error: err => console.log('shoot, error!', err),
  });
```

# Side effects

- all operators can do side effects, not only “tap()”
  - because the operator functions are lamda/arrow-function/closures
  - i.e. in JS, they have access to the ‘this’ of the class
  - every write access to ‘this’ is a side effect
- this is a great opportunity, but also destroys our dream of “a value flows throw the async stream”
- opinionated advice:
  - side effects only in  tap() (and switchMap?)
  - use tap() as rare as possible
  - stick to the dream ”flow of values”!

```
class UserInfoComponent implements OnInit {
  superHeroInfos: SuperHeroInfo[] = [];
  hasGodBadge = false;
  numOfSuperHeroBadges: number | null = null;

  constructor(private readonly userBadgesService: UserBadgesService) {}

  ngOnInit() {
    this.userBadgesService
      .loadBadges()
      .pipe(
        tap(badges => {
          this.hasGodBadge = badges.includes('god');
        }),
        map(badges => {
          const superHeroBadges = this.getSuperHeroBadges(badges);
          this.numOfSuperHeroBadges = superHeroBadges.length;
          return superHeroBadges;
        }),
        switchMap(superHeroBadges => {
          return this.userBadgesService.loadSuperHeroInfo(superHeroBadges);
        })
      )
      .subscribe(superHeroInfos => {
        this.superHeroInfos = superHeroInfos;
      });
  }
```

# Hands-on 4: Side-effects and errors

- Branch: git switch hands-on-4-side-effects
- Add loading logic (just set the variable isPriceLoading variable, the rest is coded in the template)
- Add error handling (no variable, think about what would make sense)
  - when the DataService calls fail (induce by e.g. calling getPriceFromSIXWithError, 50% chance of failing)
  - Extra: when something else failes
  - Extra: think about how the stream could easily recover from price loading errors
  - Tip: Log the error in the console
- Don't forget to think about loading logic in case an error happens!
- Extra (w/ Angular knowledge): catch and pass loading error as string literal 'loadingError' through the stream, make template handle it and show message

# Agenda & Schedule

## Beginner 9h-12h

- Observable vs Promise
- The 3 kinds of Subjects
- Async Mappers
- Reactive combining
- Hands-on using `ReplaySubject`, `take`, `switchMap`, `forkJoin`, `combineLatest`

## Advanced 13h - 15h

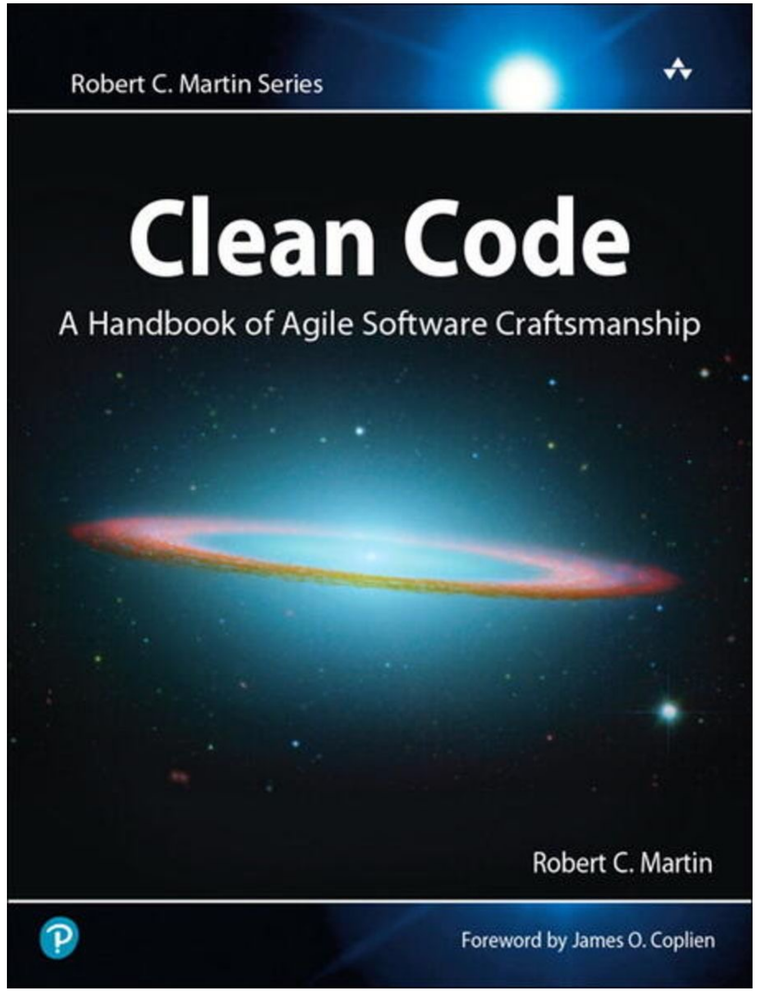
- Theory: Paradigm & Classifications
- Side Effects, 3-Channel-Model, Error Handling
- Hands-on using `tap`, `catchError`, `finalize`

## Expert 15h-17h

- RxJS clean code
- Multicast vs Unicast
- Hot vs Cold Observables
- Writing custom pipe operators
- Hands-on using `share`, `shareReplay`

# Clean Code

- Recap:
  - Good naming/readability
  - DRY
  - small/well-designed units/functions
    - Single Responsibility
  - avoid deep nesting
  - ...
- 90% of clean code for Reactive Programming is the same
- but one extra question: How to design “small units”?



# Clean Code

- What is “dirty code” here?

```
combineLatest([
  selectedItemSubject.pipe(
    map(item => item.trim()),
    filter(item => item != '')
  ),
  currentViewModeSubject.pipe(
    switchMap(currentView =>
      viewModeManagerService.loadCurrentViewModePref(currentView)
    ),
    map(viewMode => ({ ...viewMode, currentDevice: getCurrentDeviceType() }))
  ),
])
  .pipe(
    tap(([item, viewMode]) => {
      showNoPadding = viewMode.type === 'xs';
    }),
    switchMap(([item, viewMode]) =>
      forkJoin([
        configService.loadConfig(item),
        configService.loadConfig2(viewMode.type),
      ]).pipe(
        tap(([loadedItemConfig, loadedViewModeConfig]) => {
          itemConfig = loadedItemConfig;
          viewModeConfig = loadedViewModeConfig;
        }),
        switchMap(() => {
          return detailService.loadData(item, itemConfig, viewModeConfig);
        })
      )
    )
  )
  .subscribe(data => {
    itemDetail = data;
  });
```

# Observables as variables

- typical RxJS problem: “pipes” tend to become big
  - hard to read
  - hard to maintain
  - hard to extend
  - no Single Responsibility
- solution:
  - break into observables-as-variables
  - treat them like functions
  - avoid tap somewhere in the middle

```
const selectedItem$ = selectedItemSubject.pipe(
  map(item => item.trim()),
  filter(item => item != '')
);
const currentViewMode$ = currentViewModeSubject.pipe(
  switchMap(currentView =>
    viewModeManagerService.loadCurrentViewModePref(currentView)
  ),
  map(viewMode => ({ ...viewMode, currentDevice: getCurrentDeviceType() }))
);

combineLatest([selectedItem$, currentViewMode$])
  .pipe(
    tap(([item, viewMode]) => {
      showNoPadding = viewMode.type === 'xs';
    }),
    switchMap(([item, viewMode]) =>
      forkJoin([
        configService.loadConfig(item),
        configService.loadConfig2(viewMode.type),
      ]).pipe(
        tap(([loadedItemConfig, loadedViewModeConfig]) => {
          itemConfig = loadedItemConfig;
          viewModeConfig = loadedViewModeConfig;
        }),
        switchMap(() => {
          return detailService.loadData(item, itemConfig, viewModeConfig);
        })
      )
    )
  )
  .subscribe(data => {
    itemDetail = data;
  });
```

# Observables as variables

- Extreme form
  - create all your observables you need
  - in the end, only “observable.subscribe”
  - Example for “reified reactive programming”

```
currentConfigsForItemAndViewMode$.subscribe(
  ([loadedItemConfig, loadedViewModeConfig]) => {
    itemConfig = loadedItemConfig;
    viewModeConfig = loadedViewModeConfig;
  }
);
currentViewMode$.subscribe(viewMode => {
  showNoPadding = viewMode.type === 'xs';
});
currentItemDetail.subscribe(data => {
  itemDetail = data;
});
```

```
const selectedItem$ = selectedItemSubject.pipe(
  map(item => item.trim()),
  filter(item => item != '')
);

const currentViewMode$ = currentViewModeSubject.pipe(
  switchMap(currentView =>
    viewModeManagerService.loadCurrentViewModePref(currentView)
  ),
  map(viewMode => ({ ...viewMode, currentDevice: getCurrentDeviceType() }))
);

const currentConfigsForItemAndViewMode$ = combineLatest([
  selectedItem$,
  currentViewMode$,
]).pipe(
  switchMap(([item, viewMode]) =>
    forkJoin([
      configService.loadConfig(item),
      configService.loadConfig2(viewMode.type),
    ])
  )
);

const currentItemDetail$ = currentConfigsForItemAndViewMode$.pipe(
  withLatestFrom(selectedItem$),
  switchMap(([[itemConfig, viewConfig], item]) => {
    return detailService.loadData(item, itemConfig, viewConfig);
  })
);
```

# The Unicast Problem

- Notice how often we subscribe on “currentViewMode$”
  - Subscribing is like calling a function
  - loadCurrentViewModePref will be called every time!

```
currentConfigsForItemAndViewMode$.subscribe(
  ([loadedItemConfig, loadedViewModeConfig]) => {
    itemConfig = loadedItemConfig;
    viewModeConfig = loadedViewModeConfig;
  }
);
currentViewMode$.subscribe(viewMode => {
  showNoPadding = viewMode.type === 'xs';
});
currentItemDetail.subscribe(data => {
  itemDetail = data;
});
```

```
const selectedItem$ = selectedItemSubject.pipe(
  map(item => item.trim()),
  filter(item => item != '')
);

const currentViewMode$ = currentViewModeSubject.pipe(
  switchMap(currentView =>
    viewModeManagerService.loadCurrentViewModePref(currentView)
  ),
  map(viewMode => ({ ...viewMode, currentDevice: getCurrentDeviceType() }))
);

const currentConfigsForItemAndViewMode$ = combineLatest([
  selectedItem$,
  currentViewMode$,
]).pipe(
  switchMap(([item, viewMode]) =>
    forkJoin([
      configService.loadConfig(item),
      configService.loadConfig2(viewMode.type),
    ])
  )
);

const currentItemDetail$ = currentConfigsForItemAndViewMode$.pipe(
  withLatestFrom(selectedItem$),
  switchMap(([[itemConfig, viewConfig], item]) => {
    return detailService.loadData(item, itemConfig, viewConfig);
  })
);
```

# Multicast vs Unicast

- Simple:
  - all subjects are multicast
  - all observables are unicast
- Complicated:
  - we can easily cast a subject to an observable – still multicast
  - all action in the pipe is unicast
  - we can make an observable multicast

```
// multicast because it's a Subject
someSubject.pipe(
  // multicast because above share()
  tap(() => executeStupidSideEffect()),
  switchMap(() => callSomeApi()),
  share(),
  // not multicast because below share()
  tap(() => executeStupidSideEffect())
);
```

# Making Multicast

- share vs shareReplay
  - share(): make an observable/pipe multicast

```
let shared$ = notShared$.pipe(
    share()
)
```

```
shared$ = notShared$.pipe(
  share({
    connector: () => new ReplaySubject(1),
  })
);
```

  - shareReplay(): a specifically configured share()

```
shared$ = notShared$.pipe(
  shareReplay(1)
);
```

```
shared$ = notShared$.pipe(
  share({
    connector: () => new Subject(),
    resetOnError: true,
    resetOnComplete: true,
    resetOnRefCountZero: true,
  })
);
```

```
export function shareReplay<T>(
  configOrBufferSize?: ShareReplayConfig | number,
  windowTime?: number,
  scheduler?: SchedulerLike
): MonoTypeOperatorFunction<T> {
  let bufferSize: number;
  let refCount = false;
  if (configOrBufferSize && typeof configOrBufferSize === 'object') {
    ({ bufferSize = Infinity, windowTime = Infinity, refCount = false, scheduler } = configOrBufferSize);
  } else {
    bufferSize = (configOrBufferSize ?? Infinity) as number;
  }
  return share<T>({
    connector: () => new ReplaySubject(bufferSize, windowTime, scheduler),
    resetOnError: true,
    resetOnComplete: false,
    resetOnRefCountZero: refCount,
  });
}
```

# 🥵 vs 🥶 Observable

- The question: when is the value of the observable “computed” (possibly async, of course)?
  - a: when there is a subscriber (with an observer)
  - b: when there is someone calling “next”
  - c: when the observable/variable is created
  - examples (all methods return an observable):
    - dataService.loadFromServer()
    - mouseClickDetector.mouseClicked()
    - datePicker.dateSelected()
    - reduxStore.getCarsInShoppingCart()
    - timerService.everySec(60, ‘sec)
    - of(Date.now())

# 🥵 vs 🥶 Observable

- The question: when is the value of the observable “computed” (possibly async, of course)?
  - a: when there is a subscriber (with an observer) 🥶
  - b: when there is someone calling “next” 🥵
  - c: when the observable/variable is created 🥶 🤔
  - examples (all methods return an observable):
    - dataService.loadFromServer() 🥶
    - mouseClickDetector.mouseClicked() 🥵
    - datePicker.dateSelected() 🥵
    - reduxStore.getCarsInShoppingCart() 🥵
    - timerService.every(60, ‘sec’) 🥶
    - of(Date.now()) 🥶 🤔

**Cold**: the value gets computed inside the observable, “because” there is a subscriber
**Hot**: the value gets computed outside of the observable and then inserted, “regardless” whether there is a subscriber

# 🥵 vs 🥶 Observable

- Simple:
  - all subjects are hot
  - all observables are cold
- Complicated:
  - share()/shareReplay() makes a cold observable semi-hot
    - cold until first one subscribed, for later subscribers its hot
    - but everything below the share() is cold
  - ReplaySubject with a pipe is semi-cold
    - a subscriber will “trigger” the computations/side effects in the pipe, but then every ‘next’ will also trigger it

**“Observables are like function calls”**
**WRONG!**
**Only cold ones are.**

# 🥶 observable's execution time

- subscribing is not only listening, its also triggering!
  - observables are lazy (like functions)
  - creation time != execution time
- Implications
  - be aware of of() and the operator startWith()
  - use defer() to create when subscribed
  - also tap can execute callbacks on subscribe

```
const notWorkingNow$ = of(new Date());
const now$ = defer(() => of(new Date()));

notWorkingNow$.subscribe(time => console.log('not working: ', time));
now$.subscribe(time => console.log('working: ', time));

setTimeout(() => {
  notWorkingNow$.subscribe(time => console.log('2s later not working: ', time));
  now$.subscribe(time => console.log('2s later working: ', time));
}, 2000);
```

```
.pipe(
  tap({ subscribe: () => console.log('yay, someone finally listens to me!') })
)
```

# Hands-on 5: clean code & shared observables

- Branch: git switch hands-on-5-clean-code-and-shared-observables
- Add a new feature
  - as soon as price is loaded, call this.accountService.isAmountAvailable(amount) to check if this amount is available
  - if not, show warning (set the variable showAmountNotAvailableWarning)
  - handle loading logic for amount availability checking (set variable isAmountAvailbilityLoading)
- Avoid a huge pipe! Adhere to clean code!
  - at some point, you should need to use share(). Keep an eye on the price loading loggings!
- Extra (w/ Angular knowledge): get rid of all subscribes!

# Build custom pipe operators

- lets check “filter” in RxJS source code:

```
function filter<T>(predicate: (value: T, index: number) => boolean, thisArg?: any): MonoTypeOperatorFunction<T>
```

- MonoTypeOperatorFunction is just an OperatorFunction with same type in generics

```
export interface MonoTypeOperatorFunction<T> extends OperatorFunction<T, T> {}
```

- OperatorFunction is just a function that takes an observable and returns an observable
- so a custom operator has to return a function that takes an observable and returns an observable
- For example, a noop operator:

```
function noop<T>(): MonoTypeOperatorFunction<T> {
  return (source: Observable<T>) => source;
}
of('some value')
  .pipe(noop())
  .subscribe(val => console.log(val));
```

# Build custom pipe operators

- another example: stringify()

```
function stringify<T>(): OperatorFunction<T, string> {
  return (source: Observable<T>): Observable<string> =>
    source.pipe(map(val => JSON.stringify(val)));
}
of('some value')
  .pipe(stringify())
  .subscribe(val => console.log(val));
```

# Hands-on 6: custom operator

- Branch: git switch hands-on-6-custom-operator
- Add a new operator
  - switchMapKeep()
  - should do a switchMap but keep also the input and emit an array [input, output]
  - Notice:
    - Code is not working at this branch
    - When task finished, code should work, but don't expect different behaviour of the app

# Time Booking

- Book your time on
  - **Project Number: 3010000-11**
  - **Task: Nn-Blb**
  - (applies for colleagues with Swiss Contract only)

# Further topics

- retry logic in RxJS
- scheduler API in RxJS
- testing with rxjs/testing or jasmine-marbles
- the dreaded “unsubscribe”/memory-leak discussion

# Summery

## Practitioner

- Subject < ReplaySubject < BehaviourSubject
- map to observables: switchMap is best
- combine with forkJoin (easy), combineLatest (harder)

## Engineer

- RxJS = JS implementation of Reactive Extensions, implementing the Observer Pattern (reified)
- 3 message types and error completes the observable
- side effects are bad but necessary evil

## Lead Engineer

- Clean Code: Avoid large pipes, split the streams in observables as variable
- sharing is caring
- Implementing custom pipe operators is surprisingly easy