Reactive programming with RxJs

Streams FTW

Streams of values

User triggered events

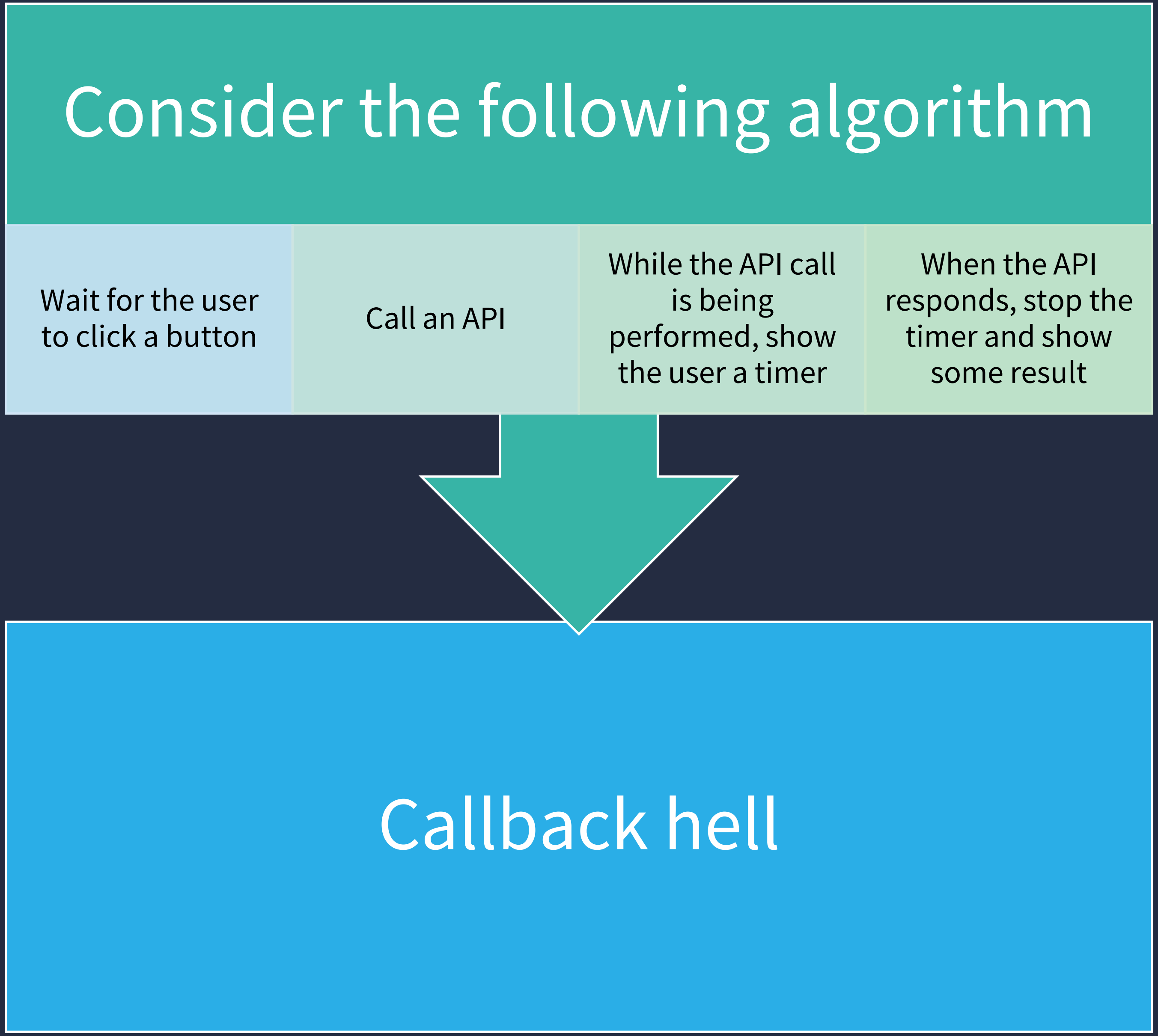
Programmatic intervals (setInterval)

Timeouts (setTimeout)

An HTTP request

Basically any asynchronous event can be thought as a stream emitting a value

Combining streams of values can be painful



Consider the following algorithm

Wait for the user to click a button

Call an API

While the API call is being performed, show the user a timer

When the API responds, stop the timer and show some result

Callback hell

Reactive extensions for JavaScript (RxJs)

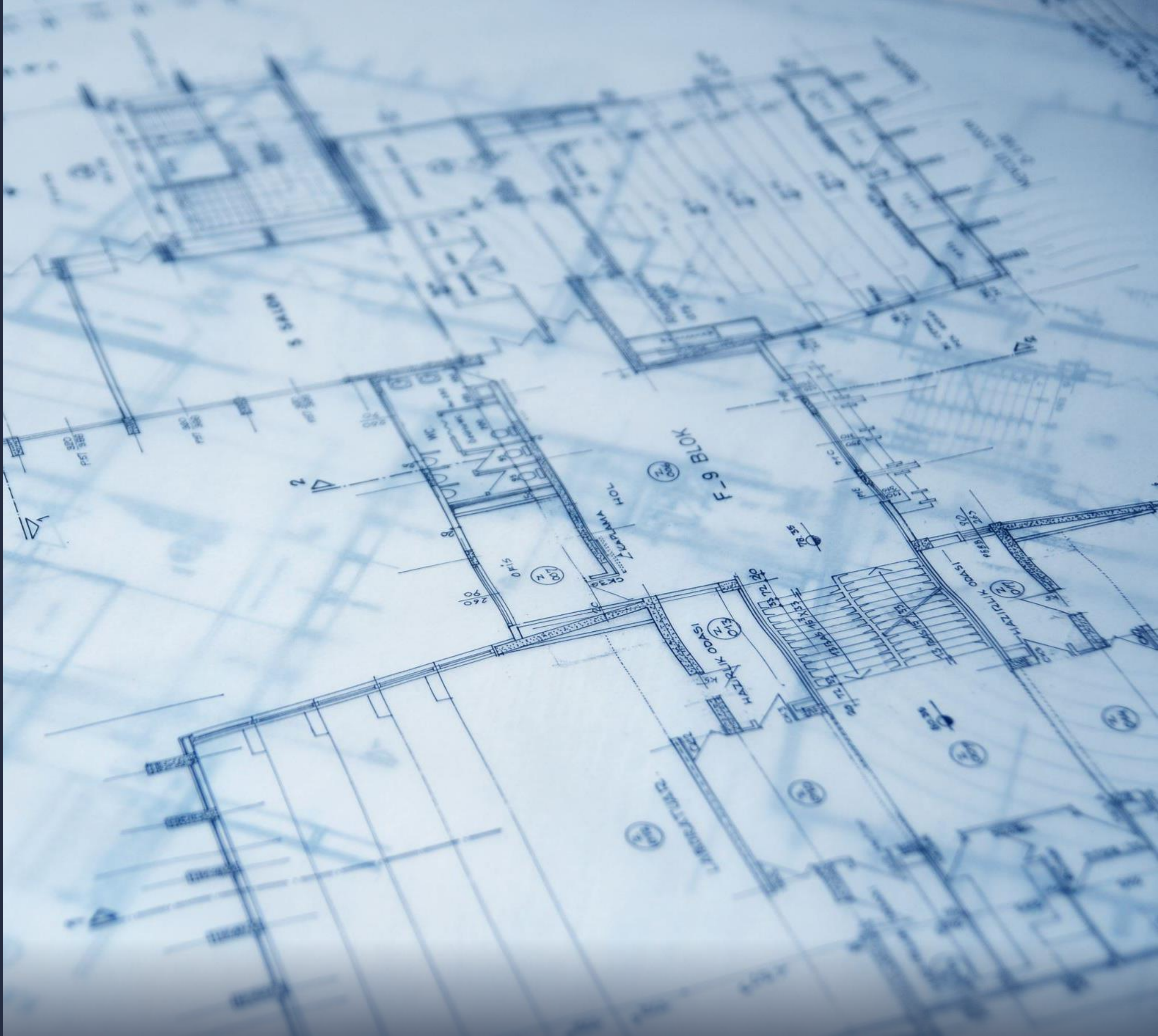
EXPOSES JAVASCRIPT API TO DEFINE AND CONSUME STREAMS

ALLOWS EASILY WORKING WITH VALUE STREAMS, EVEN IN COMPLEX SCENARIOS

REVOLVES AROUND BLUEPRINTS RATHER THAN IMPERATIVE FLOWS

Observables

- Are blueprints for value streams
- Actually become streams only when subscribed
- Some basic examples of stream blueprints
    - timer
    - fromEvent



Observables – core rules

- The «subscribe» method accepts three callbacks
  - OnNext: (value) => {…}
  - OnError: (error) => {…}
  - OnCompleted: () => {…}
- These callbacks implement the Observable contract (ReactiveX - The Observable Contract)

Observables – the lifecycle of an observable

An Observable may make zero or more OnNext notifications

Those notifications can be followed by either an OnCompleted or an OnError notification, but not both

An Observable may emit no items at all

An Observable may also never terminate with either an OnCompleted or an OnError notification

Observables must issue notifications to observers serially (not in parallel)

When an Observable does issue an OnCompleted or OnError notification, the Observable may release its resources and terminate

Its observers should not attempt to communicate with it any further

An OnError notification must contain the cause of the error

Observables – subscription rules

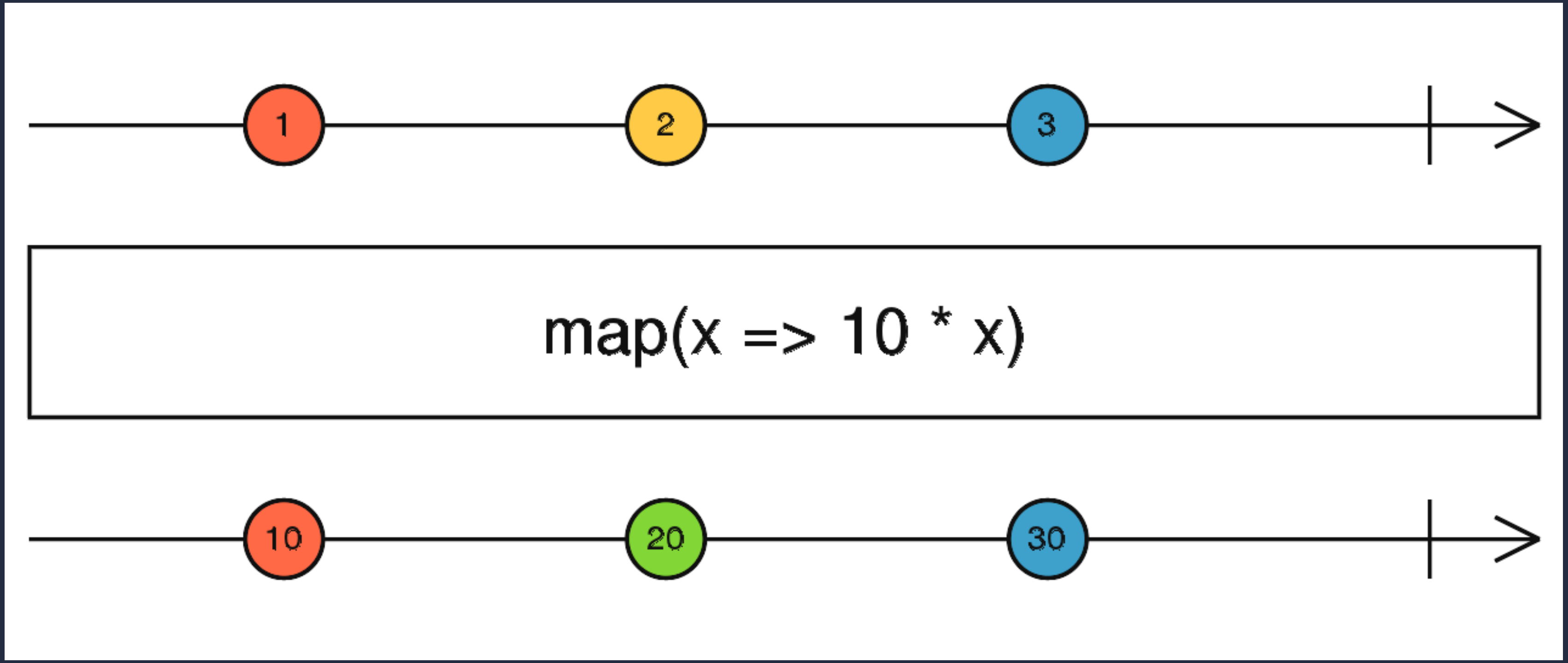
- An Observable may begin issuing notifications to an observer immediately after the Observable receives a Subscribe notification from the observer
- When an observer issues an Unsubscribe notification to an Observable, the Observable will attempt to stop issuing notifications to the observer
- When an Observable issues an OnError or OnComplete notification to its observers, this ends the subscription
- Observers do not need to issue an Unsubscribe notification to end subscriptions that are ended by the Observable in this way

Manipulating streams with RxJs

- RxJs exposes a wide variety of operators
- Operators are the tools RxJs provides to
  - Manipulate an existing observable (pipeable operators)
  - Create new observables from scratch (creation operators)
- Piping is the practice to define a readable blueprint for data stream manipulation by concatenating operators
- Operators behavior is best described by marble diagrams (RxJS - RxJS Operators)

Simple data manipulation

- Transform the item emitted by an observable by applying a function to it
- Not to be confused with the array's map method
- RxJS - map



Interlude – imperative vs reactive design

Adding a lot of logic to the subscribe method might be tempting, but does not scale well in complexity

Derive different data from the same original stream

Perform different manipulations on the same original stream

Perform different actions in response to data

Rules of thumb of reactive design

More observable blueprints over subscribe logic

Never nest subscriptions

Observable concatenation

- RxJs operators that tackle the data stream concatenation problem: one stream after another
    - mergeMap (RxJS - mergeMap)
    - switchMap (RxJS - switchMap)
    - concatMap (RxJS - concatMap)
    - exhaustMap (RxJS - exhaustMap)

Observable combination

- RxJs operators that tackle the data stream combination problem: more streams in parallel
  - forkJoin (RxJS - forkJoin)
  - combineLatest (RxJS - combineLatest)
  - withLatestFrom (RxJS – withLatestFrom)

Forcing completion of long running observables

- Operators that deal with long running observables, or simply solve scenarios where we only need a certain number of values from the stream
    - take (RxJS - take)
    - takeWhile (RxJS - takeWhile)
    - first (RxJS - first)

Data filtering and timing

- Let’s discuss the following requirements for a search typeahead
  - Should capture the user input from an <input> tag
  - Should trim spaces from the input string
  - Should call some search API only if the input is 3 characters or more
  - Should call the API at most once every second and only if the input has been changed
  - Should print the number of received items and info about all the results

Data filtering and timing

- The fromEvent operator creates an observable from the user input event
- The filter operator can filter out input values shorter than the minimum input length
- The distinctUntilChanged operator prevents duplicate search calls for identical user inputs
- The debounceTime oeprator ensures the API gets called at most once per second
- The switchMap operator actually subscribes to the search API call
    - Bonus: it cancels the previous call if still ongoing and the input changes

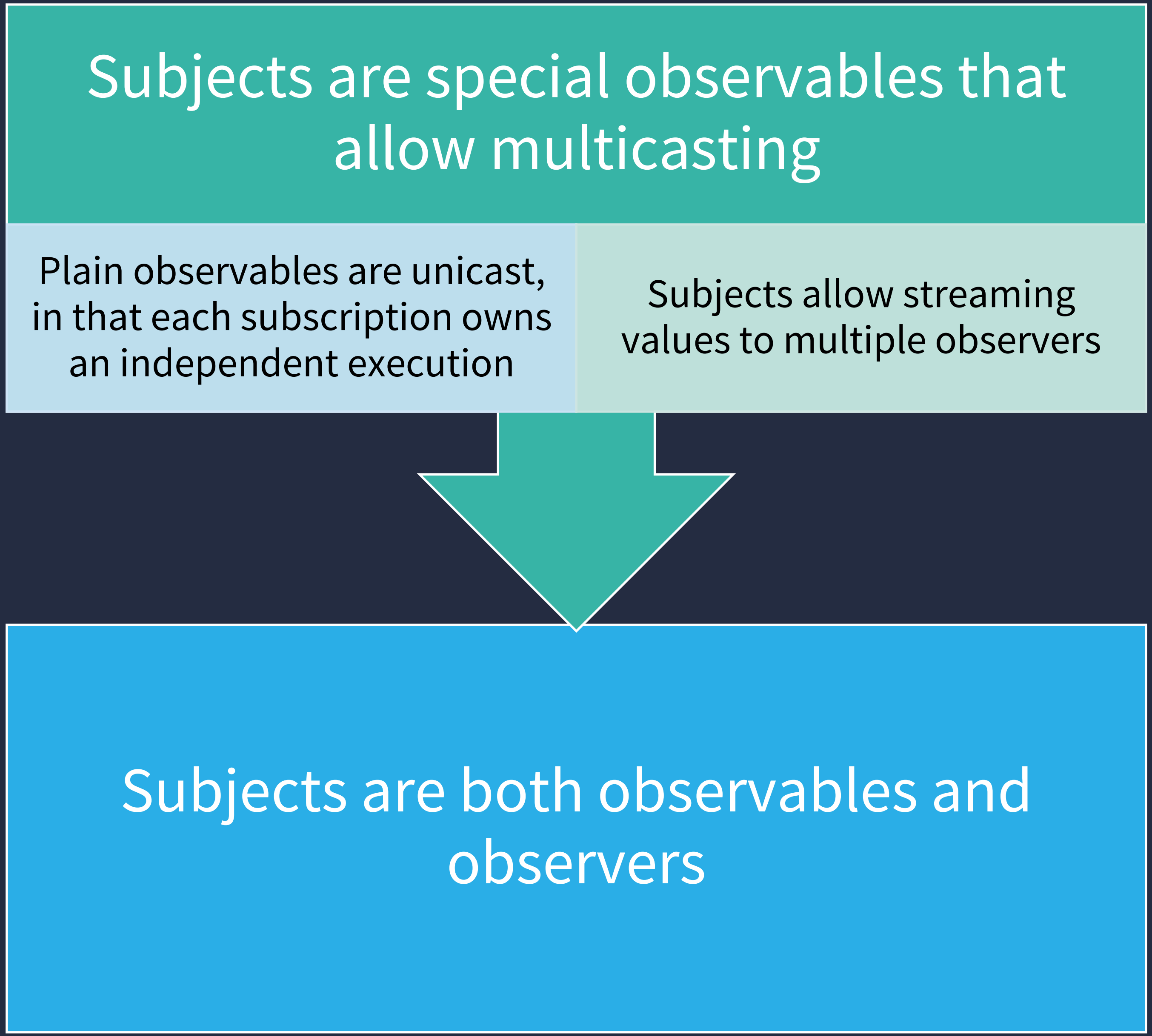
Data filtering and timing

- Throttling: discard data emitted from the input stream for a while
    - Emit a value from the input stream
    - Wait for some time to pass before emitting the next one
- Debouncing: waiting for the input stream to be «stable»
    - Wait for the input stream to not emit for a given amount of time
    - Emit the last emitted value

Error handling in RxJs

- Error handling strategies
  - Catch and replace (catchError)
  - Catch and rethrow (catchError -> throwError -> finalize)
  - Retry (retry)

RxJs subjects and the store pattern



Subjects are special observables that allow multicasting

Plain observables are unicast, in that each subscription owns an independent execution

Subjects allow streaming values to multiple observers

Subjects are both observables and observers

RxJs BehaviorSubject

Subjects that support late subscriptions

When a late subscription is performed, a BehaviorSubject emits its last value to the new subscriber

If it hasn’t completed before

«BehaviorSubject»s can be assumed to have memory

Best candidates to implement the store pattern

Emit the current value to each new subscriber

Emit every subsequent value when the value changes