academeez

Operators

RXJS super strong toolbox!

Operators

- With Operators you can easily create Observables
- With Operators you can easily manipulate Observable from one data stream to the other
- Think of the operators as a toolkit for solving common observable problems

```
import { interval, Observable } from 'rxjs';
import { map } from 'rxjs/operators';

const helloObservable: Observable<string> = interval(1000).pipe(
  map((index: number) => 'hello!')
)
```

Operators as functions

academeez

- Some **Operators** returns an Observable
  - Those are usually used for creating a new Observable
- Some **Operators** will return a function that returns an Observable
  - The function returned will usually get as argument an Observable
  - Those are used to manipulate the data stream

Types of operators

- We can classify the operators to 2 groups
  - Creating operators - those operators will help you create a new Observable
  - Manipulation operators - those operators will help you transform an Observable
- Let’s learn about those types…

Creating Operators

- These are functions that return a new Observable
- The **interval** will create an Observable that emits every period of time a counter

```
// used for create an Observable
// emits counter every second
export declare function interval(period?: number, scheduler?: SchedulerLike): Observable<number>;
```

Manipulation Operators

academeez

- These operators will help you transform your observable to another observable
- They do not effect the original observable
- Subscribing to the result observable will subscribe to both the observables

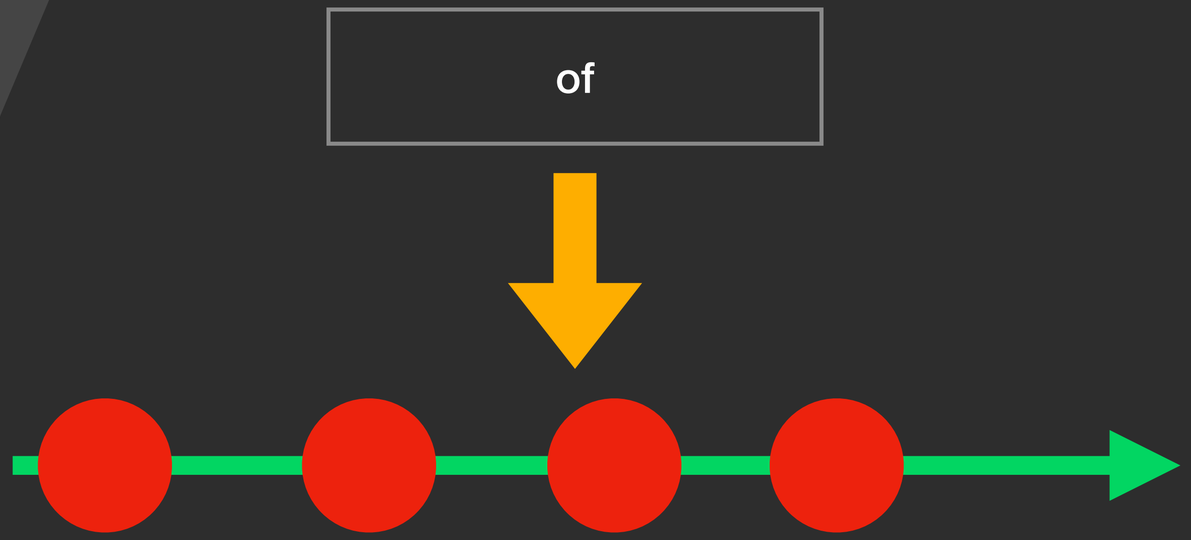
```
// operator for transforming the datastream pulses
export declare function map<T, R>(project: (value: T, index: number) => R, thisArg?: any):
OperatorFunction<T, R>;

// the manipulation operators return this type
export interface OperatorFunction<T, R> extends UnaryFunction<Observable<T>, Observable<R>> {

// the base type that manipulation operators are returning
export interface UnaryFunction<T, R> {
  (source: T): R;
}
```

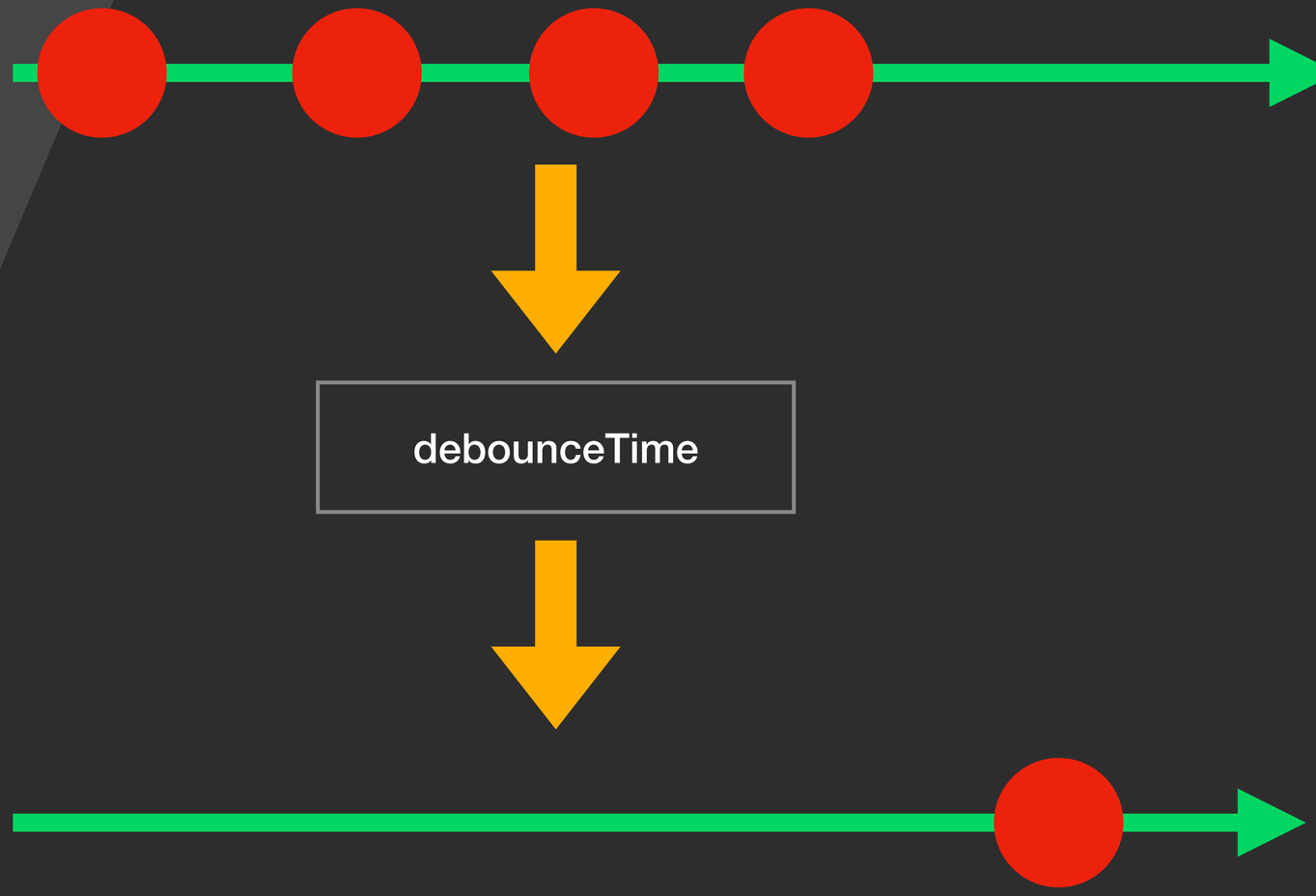
Marble Diagram

- To understand how to use operators, it helps to look at what they give you in a marble diagram
- A creating operator is a function that produce an Observable:



Marble Diagram - Manipulating operators

- The manipulation operators will transform one data stream to another



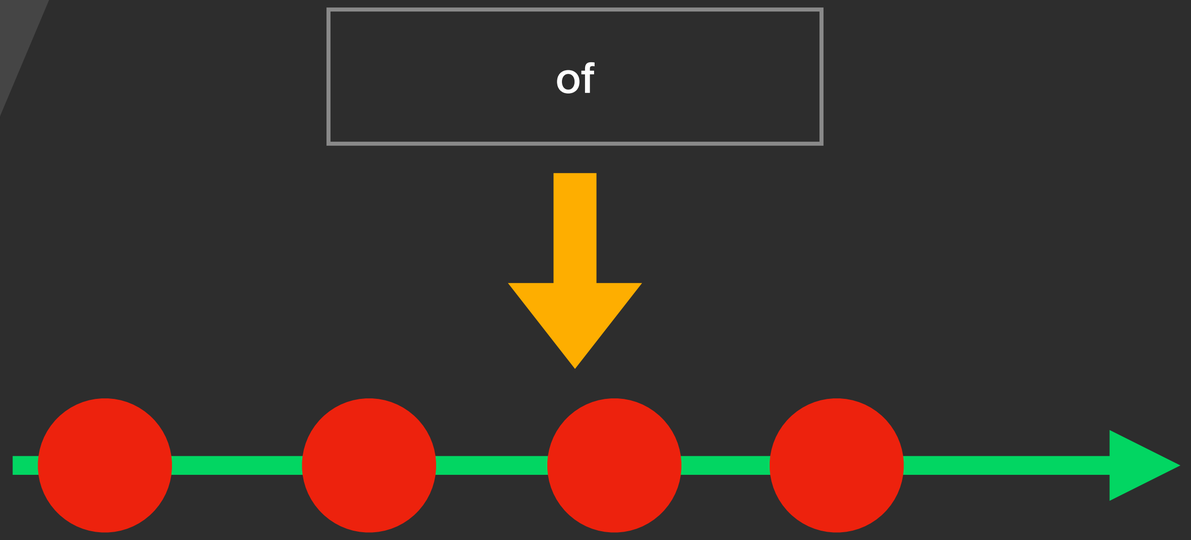
Operators category

- When you start using operators you will be overwhelmed by the amount of operators you can use
- It will be hard to familiarise yourself with all the operators.
- It will be easier to understand the different groups of operators you have and
- Each group does a certain manipulation to your data stream
- If I know the manipulation I need and the manipulation of the groups it will be easier to locate the operator we need

Creating operators

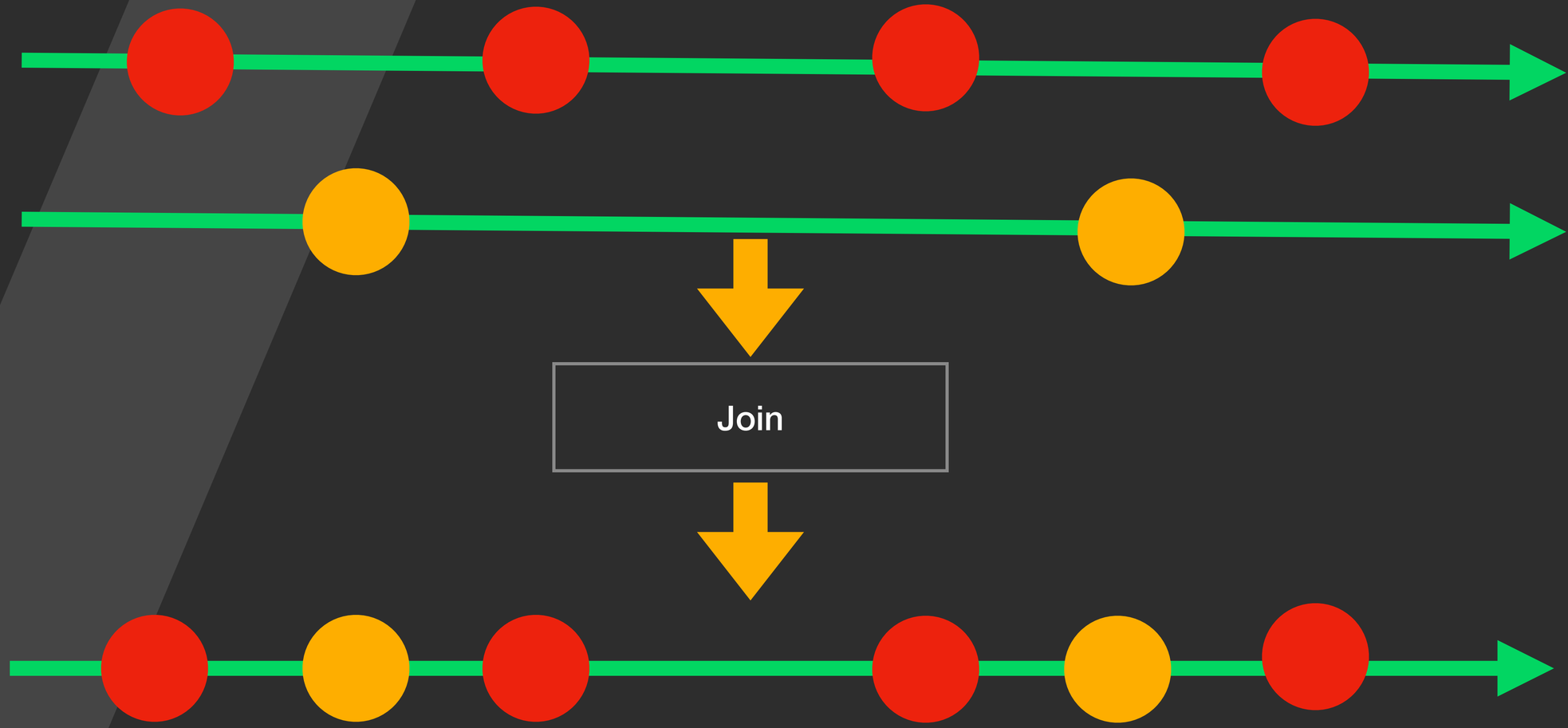
academeez

- We've already seen those, they will be operators that helps us create an Observable



Join operators

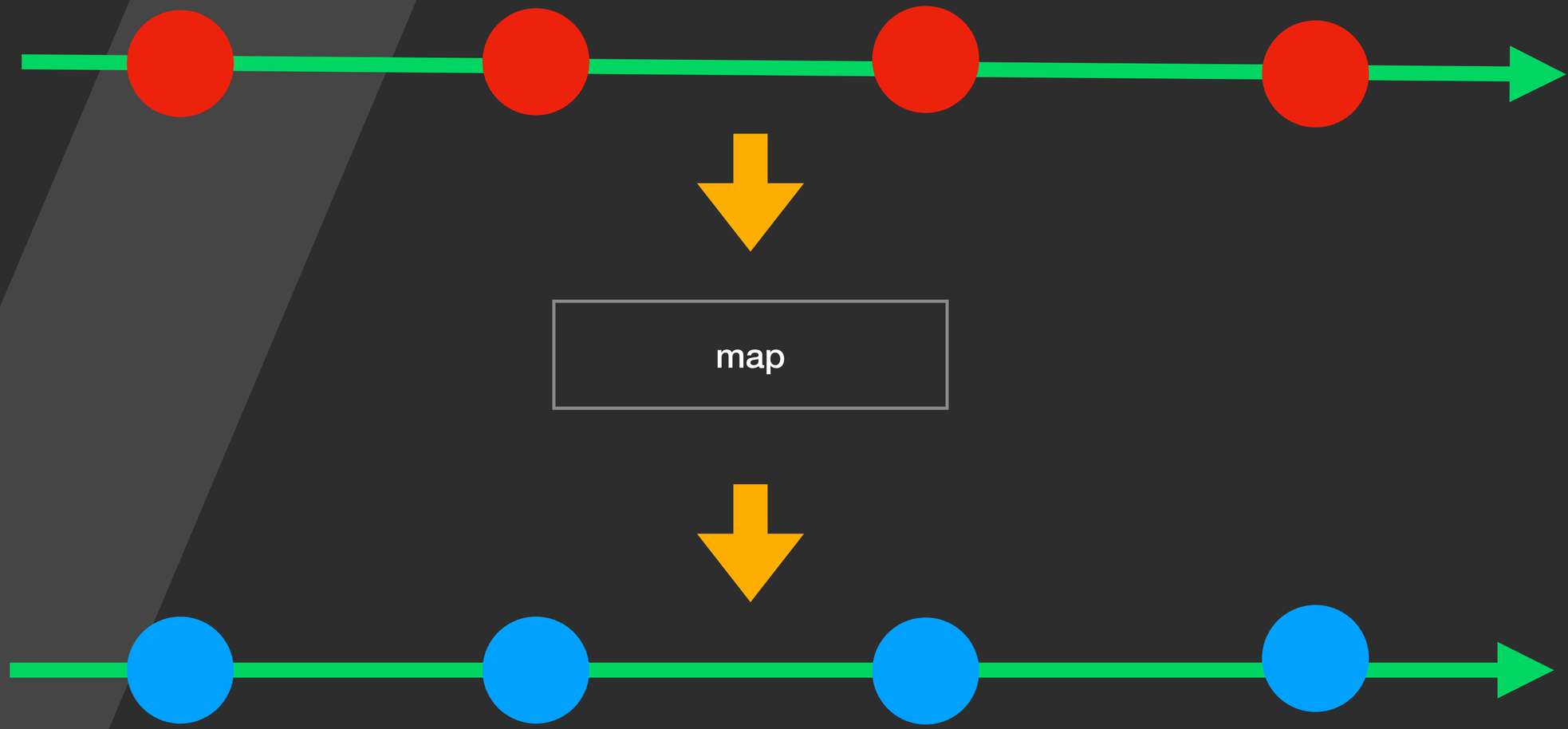
- Create a new Observable by joining multiple Observables



Transformation operators

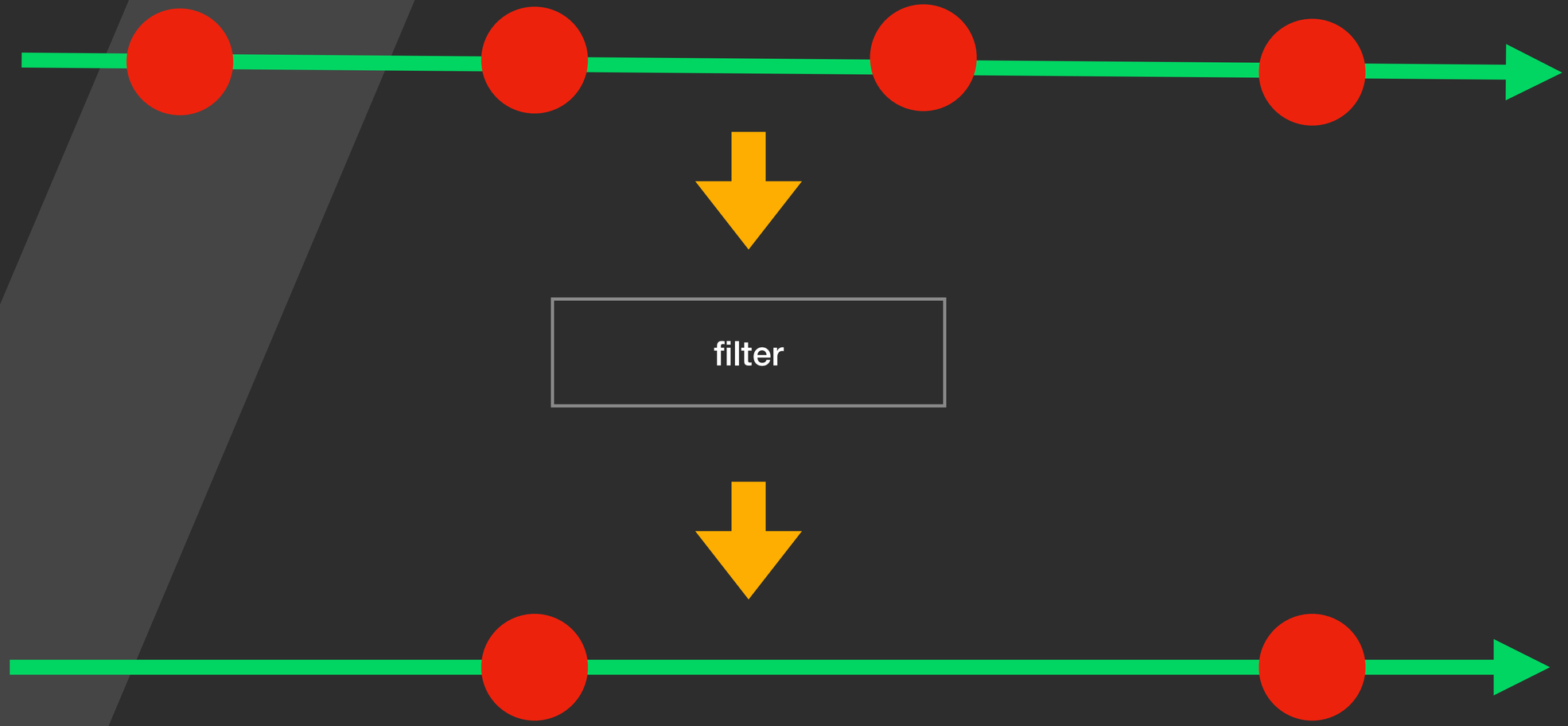
academeez

- Transforming a pulse in the data stream



Filtering operators

- Reducing the number of pulses



Using the operators

- Let’s go over some common operators you have to know
- To use the creating operators we simply call their function with whatever arguments they require
- To use the manipulating operators we chain them in the **pipe** method

```
// creates an observable with one pulse of an array
of([1,2,3]);

// transforms an observable
someObs.pipe(
  operator1,
  operator2,
  ...
)
```

Popular operators

- Let’s go over some popular operators you need to know and you will probably need to use…

of

- The operator function will get as an argument data for the data stream
- It will create a sync observable which pulse the data

```
import { of } from 'rxjs';

// creates an observable with one pulse of an array
of([1,2,3]);
```

from, fromEvent

academeez

- The operator function will get as an argument data for the data stream
- It will create a sync observable which pulse the data

```
import {from, fromEvent} from 'rxjs';

// creates an observable which emits 3 sync pulses
// 1,2,3
from([1,2,3]);

// creates an observable which emits every click event a pulse
// the pulse will contain the Event object
fromEvent(document.getElementById('stam-button'), 'click');
```

interval

- Creates an observable which starts to count 0..1..2
- The count will be according to the milliseconds

```
import { interval } from 'rxjs';

// creates an observable that emits 0..1..2..
// every second
interval(1000)
```

map

- Transforming a pulse to another pulse
- In the following example we are transforming an Observable with a string to another Observable that counts the string number of characters

```
import { interval } from 'rxjs';
import { map } from 'rxjs/operators';

// creates an observable that emits hello0..hello1..
// every second
interval(1000).pipe(
  map((count: number) => 'hello' + count)
)
```

mergeMap

academeez

- Similar to map only we return an Observable from the function
- The Observable in the mergeMap function can use the original observable value
- The result will be a flatten Observable

```
import { interval } from 'rxjs';
import { mergeMap } from 'rxjs/operators';

// creates an observable that sends a request
// every second
interval(1000).pipe(
  mergeMap((count: number) => httpClient.get('https://academeez.com/api' + count))
)
```

debounceTime

academeez

- Filters the number of pulse based on time passed since pulse emitted
- If a pulse emitted and a time period pass without another pulse than that pulse is emitted
- Good for throttling

```
import { Subject } from 'rxjs';
import { debounceTime } from 'rxjs/operators';

const throttlingSubject: Subject<string> = new Subject()

// this will send only hello pulse
// after 1 sec
throttlingSubject.pipe(
  debounceTime(1000)
).subscribe((greeting: string) => console.log(greeting));

throttlingSubject.next('hi');
throttlingSubject.next('hello');
```

filter

academeez

- Will emit pulse based on a predicate function

```
import { interval } from 'rxjs';
import { filter } from 'rxjs/operators';

// will emit every 2 seconds
// the numbers divided by 2
interval(1000).pipe(
  filter((num) => num % 2 === 0)
);
```

first, take

- first will take the first value
- take - will take number of values

```
import { interval } from 'rxjs';
import { take, first } from 'rxjs/operators';

// will emit a pulse every second until 4 is reached
interval(1000).pipe(
  take(5)
);

// will emit one pulse 0 after 1 sec
interval(1000).pipe(
  first()
);
```

tap

- This operator is quite useful in debugging
- It allows you to examine the context of the data stream without making any changes

```
import { interval } from 'rxjs';
import { take, tap } from 'rxjs/operators';

interval(1000).pipe(
  take(5),
  tap((num: number) => {
    // i can examine the current content of each pulse
    console.log(num);
  })
);
```

Operators - EX

academeez

- Let’s try and practice the usage of operators in an Angular application
- Create an angular component which display a list of ul li
- The li items should be taken from a server in this URL:
  - https://nztodo.herokuapp.com/api/task/?format=json&search=<search string>
- Above the list there will be a search box every time the user types a string a request is sent.
- Implement throttling which means only after 1 sec passed and the user did not type anything a request is sent and the list will be populated

Summary

academeez

- Operators is what makes RXJS such an awesome library
- The fact that you have a toolbox for dealing with common async and sync problems is very useful and helps you elegantly solve complex problems with very little lines of codes
- Know which operator to choose by understanding the operator categories
- You can use the decision tree to help you find your operator

academeez

Thank You

Next Lesson: Custom Operators