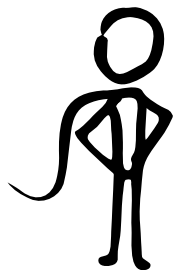


# Препарируем RxJS операторы



# Галушко Кирилл



# Тинькофф

[t.me/qeireal](https://t.me/qeireal)

[medium.com/@qeireal](https://medium.com/@qeireal)

[github.com/qeireal](https://github.com/qeireal)

[twitter.com/qeireal](https://twitter.com/qeireal)



# О чем поговорим

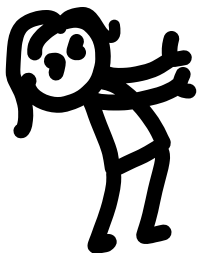
1. Что у Subscribable сущностей под капотом?
2. Что внутри у стандартных операторов?
3. Чего не хватает?
4. Как это можно реализовать?
5. И чем можно воспользоваться?

[github.com/ReactiveX/rxjs/](https://github.com/ReactiveX/rxjs/) 

# Subscribable



```
export interface Subscribable<T> {  
  subscribe(observer?: PartialObserver<T>): Unsubscribable;  
}
```



```
export interface Unsubscribable {  
  unsubscribe(): void;  
}
```

# subscribe()



```
// ...  
const { operator } = this;  
const sink = toSubscriber(observerOrNext, error, complete);  
  
if (operator) {  
    operator.call(sink, this.source);  
}  
// ...
```

# lift()



```
lift<R>(operator: Operator<T, R>): Observable<R> {  
    const observable = new Observable<R>();  
    observable.source = this;  
    observable.operator = operator;  
    return observable;  
}
```



# pipe()

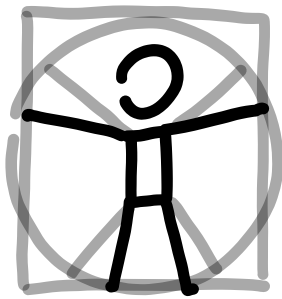


```
pipe( ... operations: OperatorFunction<any, any>[]): Observable<any> {  
    if (operations.length === 0) {  
        return this as any;  
    }  
  
    return pipeFromArray(operations)(this);  
}
```

# pipeFromArray()



```
function pipeFromArray<T, R>(fns: Array<UnaryFunction<T, R>>): UnaryFunction<T, R> {  
  // ...  
  return function piped(input: T): R {  
    return fns.reduce((prev: any, fn: UnaryFunction<T, R>) => fn(prev), input as any);  
  };  
}
```



# Операторы



# take()

13



```
class TakeSubscriber<T> extends Subscriber<T> {  
    // ...
```

```
    protected _next(value: T): void {  
        const total = this.total;  
        const count = ++this.count;  
        if (count ≤ total) {  
            this.destination.next(value);  
            if (count ≡ total) {  
                this.destination.complete();  
                this.unsubscribe();
```

```
        }
```

```
    }
```

```
}
```

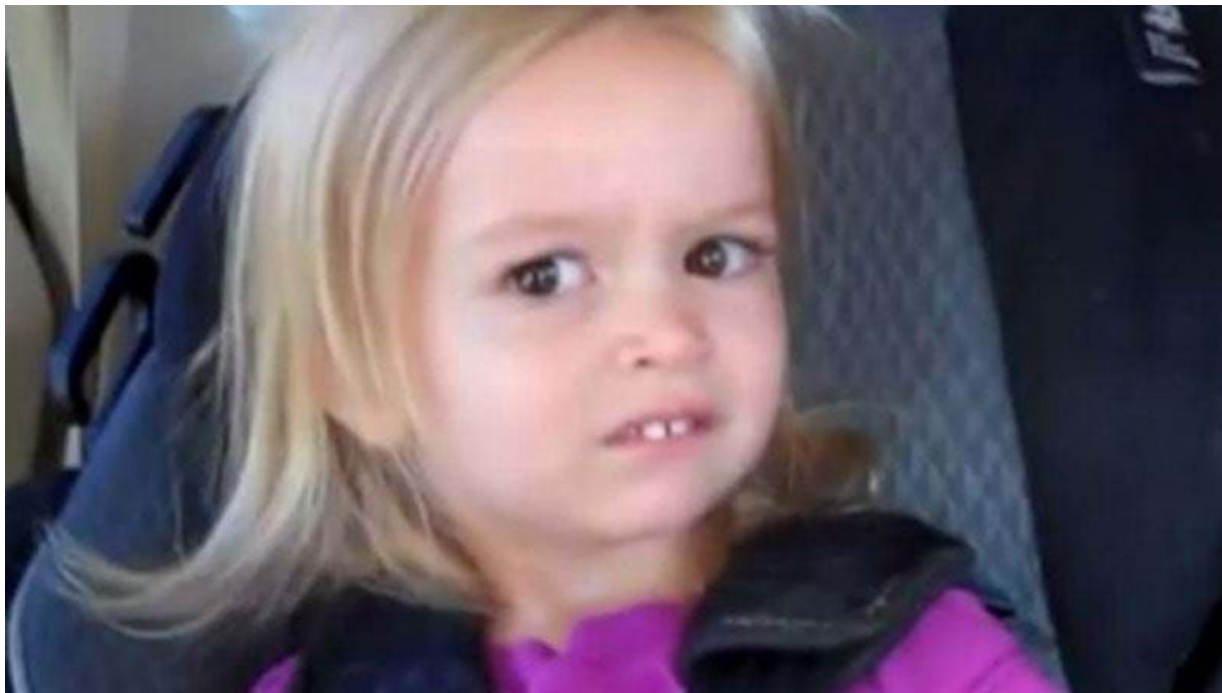
```
}
```



```
class TakeSubscriber<T> extends Subscriber<T> {  
    constructor(destination: Subscriber<T>) {  
        if (destination === this) {  
            throw new TypeError('self-subscription not allowed');  
        }  
        call(subscribeTo(this, destination), this);  
        return this;  
    }  
}
```

```
total));
```

Теперь, когда мы разобрались...









# Кастомный оператор

## Часть 1: Создание

# Правила создания

- Операторы должны всегда возвращать Observable
- Оператор должен иметь подписку на исходный Observable и не внутри возвращаемого
- Обработать все исключения
- Не допускать утечки памяти

# truthyFilter()



```
function truthyFilter<T>(): MonotypeOperatorFunction<T> {  
  return input$ => input$.pipe(filter(Boolean))  
}
```

# truthyFilter()



```
function truthyFilter<T>(): MonoTypeOperatorFunction<T> {  
  return input$ => input$.pipe(filter(Boolean))  
}
```

# Типы возвращаемых значений

“

***MonoTypeOperatorFunction*** - тип не меняется.

“

***OperatorFunction*** - тип меняется.

# Кастомный оператор

## Часть 2: Использование

# Использование

# Кастомный оператор

## Часть 3: Тестирование



# Тестирование

# Marbles тестирование



```
describe('truthyFilter оператор', () => {  
  it('Должен оставить только истинные значения', () => {  
    const source = from([0, 1, null, '', 'foo', undefined, 'bar', [], {}])  
      .pipe(truthyFilter());  
  
    const expected = cold('abcde', { a: 1, b: 'foo', c: 'bar', d: [], e: {} });  
  
    expect(source).toBeObservable(expected);  
  });  
});
```

# cdrChecking()



```
function cdrChecking<T>(  
    cd: ChangeDetectorRef,  
    rightNow?: boolean  
): MonoTypeOperatorFunction<T> {  
    return tap(() => rightNow ? cd.detectChanges() : cd.markForCheck());  
}
```

# consoleLog()

0
1
null
foo
undefined
bar

# Преобразование lodash в rxjs операторы

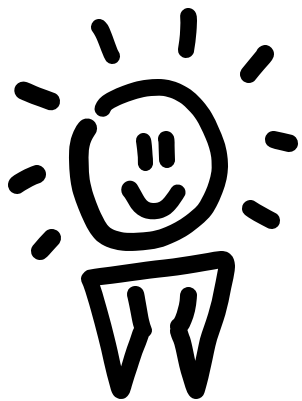


```
const pickNumbers = () => map(x =>
  _.pickBy(x, _.isNumber)
);

const source$ = of( { 'foo': 1, 'bar': 'str', 'baz': 3 } );





source$.pipe(pickNumbers()).subscribe();

// Output
// { 'foo': 1, 'baz': 3 }
```






# Готовые решения

## "rxjs-etc" Nicholas Jamieson

-  RxJS Core team maintainer
-  Observables и операторы, которые не попали в rxjs
-  Или попадут (например, endWith)
-  Много узконаправленных операторов

## "ngx-take-until-destroy" Netanel Basal

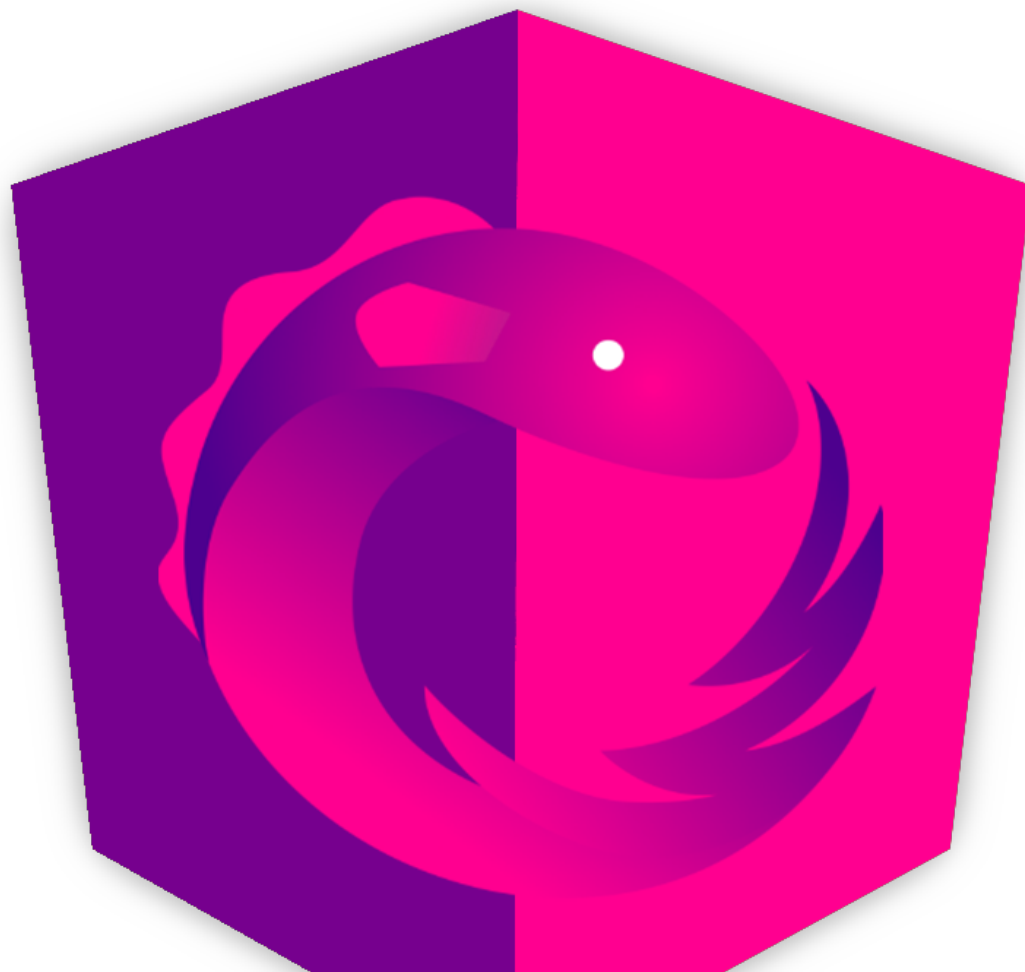
-  "Guy that you obviously know" maintainer
-  Простота
-  Нужно всегда инициализировать ngOnDestroy



```
import { untilDestroyed } from 'ngx-take-until-destroy';

@Component({
  selector: 'app-inbox',
  templateUrl: './inbox.component.html',
})
export class InboxComponent implements OnInit, OnDestroy {
  ngOnInit() {
    interval(1000)
      .pipe(untilDestroyed(this))
      .subscribe(val => console.log(val));
  }

  // This method must be present, even if empty.
  ngOnDestroy() {
    // To protect you, we'll throw an error if it doesn't exist.
  }
}
```



# Q&A

