# Knowledge share on rxjs

by Andrey Lukyanenko

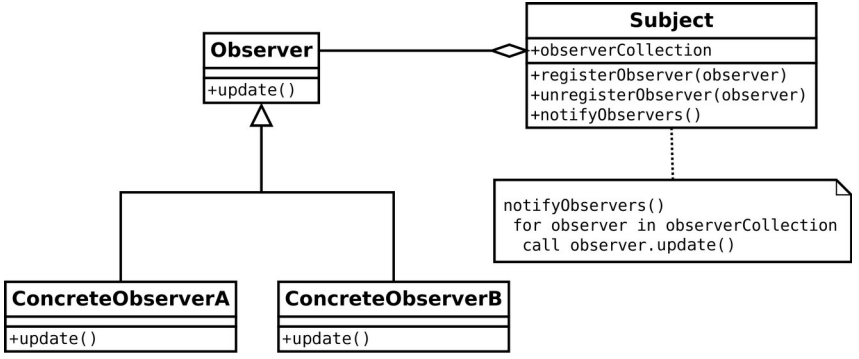
# ReactiveX family

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Lua: RxLua
- Ruby: Rx.rb
- Python: RxPY
- Go: RxGo
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP
- Elixir: reaxive
- Dart: RxDart

# Observables

# Observer pattern

- Classical pattern
- Well described in Gang of Four (GoF) book
- May be each of us implemented it at least once



# Observer pattern

Design:

```
export interface Observer {
   update : (value : string) => void;
}

export class Subject {

   observers : Observer[] = [];

   addObserver(observer : Observer)
   {
      this.observers.push(observer);
   }

   removeObserver(observer : Observer)
   {
      const index = this.observers.indexOf(observer);
      if (index !== -1) this.observers.splice(index, deleteCount: 1);
   }

   notify(value : string)
   {
      this.observers.forEach(observer => observer.update(value));
   }
}
```

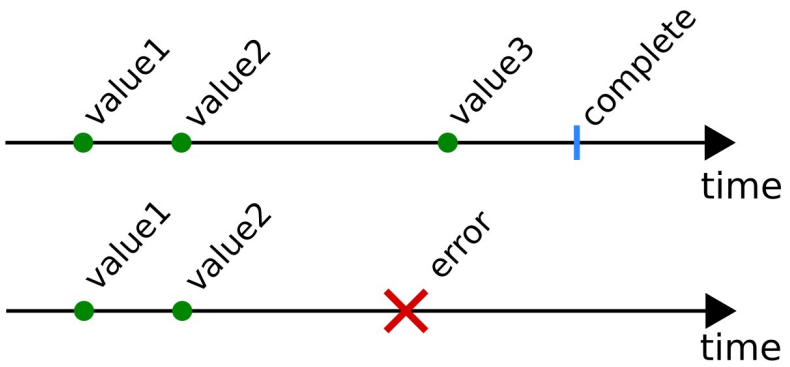
Usage:

```
const subj = new Subject();
const obs1 : Observer = {update : value => console.log('obs1: ' + value)};
const obs2 : Observer = {update : value => console.log('obs2: ' + value)};

subj.addObserver(obs1);
subj.addObserver(obs2);
subj.notify( value: 'First event for both');
subj.notify( value: 'Second event for both');

subj.removeObserver(obs1);
console.log('\nobs1 removed');
subj.notify( value: 'Third event only for one');
```

```
obs1: First event for both
obs2: First event for both
obs1: Second event for both
obs2: Second event for both

obs1 removed
obs2: Third event only for one
```

# rxjs observer pattern

- rxjs observer has 3 callback function instead one
  - **next(value)** – value notification
    - *update* in observer pattern
  - **error**(**message**) – error notification
  - **complete**() – observable is finalized notification
- Observable, is an array of events



# Creating observable

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

function createObservable(n : number)
{
   return new Observable( subscribe: observer => {
      for (let i = 0; i < 10; i++) {
         if (i === n) observer.error('Something went wrong');

         observer.next(i);
      }
      observer.complete();
   });
}

createObservable( n: 12).subscribe(
    next: value => console.log('Observer1 got value: ' + value),
    error: error => console.log('Observer1 got error: ' + error),
    complete: () => console.log('Observer1 completed!\n')
);

createObservable( n: 2).subscribe(
    next: value => console.log('Observer2 got value: ' + value),
    error: error => console.log('Observer2 got error: ' + error),
    complete: () => console.log('Observer2 completed!\n')
);
```

```
Observer1 got value: 0
Observer1 got value: 1
Observer1 got value: 2
Observer1 got value: 3
Observer1 got value: 4
Observer1 got value: 5
Observer1 got value: 6
Observer1 got value: 7
Observer1 got value: 8
Observer1 got value: 9
Observer1 completed!

Observer2 got value: 0
Observer2 got value: 1
Observer2 got error: Something went wrong
```

# Promise vs Observable

**Promises**

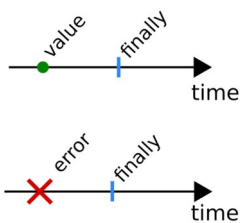
- Single value
- Eager executed
- Always async

**Observables**

- Multiple values
- Lazy executed
- Can be sync/async
- Streams open to modifications
- Hot / cold

# Single vs Multiple values

```ts
export function createPromise(n : number) : Promise<number>
{
    return new Promise( executor: resolve => {
        for (let i = 0; i < 10; i++) {
            if (i === n) throw 'Something went wrong';
            resolve(i);
        }
    });
}

createPromise( n: 12).then(
    value => console.log('Promise1 got value: ' + value),
    error => console.log('Promise1 got error: ' + error)
).finally( onfinally: () => console.log('Promise1 completed!\n'));

createPromise( n: 0).then(
    value => console.log('Promise2 got value: ' + value),
    error => console.log('Promise2 got error: ' + error)
).finally( onfinally: () => console.log('Promise2 completed!\n'));
```

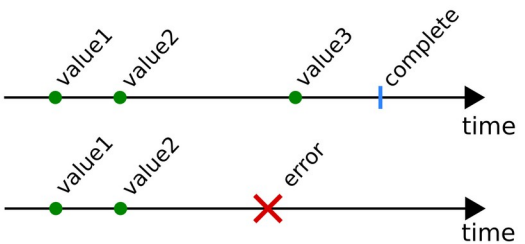



```
Promise1 got value: 0
Promise2 got error: Something went wrong
Promise1 completed!

Promise2 completed!
```

## Compare to observable:

```ts
export function createObservable(n : number)
{
    return new Observable( subscribe: observer => {
        for (let i = 0; i < 10; i++) {
            if (i === n) observer.error('Something went wrong');

            observer.next(i);
        }
        observer.complete();
    });
}
```

```
Observer1 got value: 0
Observer1 got value: 1
Observer1 got value: 2
Observer1 got value: 3
Observer1 got value: 4
Observer1 got value: 5
Observer1 got value: 6
Observer1 got value: 7
Observer1 got value: 8
Observer1 got value: 9
Observer1 completed!

Observer2 got value: 0
Observer2 got value: 1
Observer2 got error: Something went wrong
```



# Sync / Async

## Sync:

```
export function createObservable(n : number)
{
   return new Observable( subscribe: observer => {
      for (let i = 0; i < 10; i++) {
         if (i === n) observer.error('Something went wrong');

         observer.next(i);
      }
      observer.complete();
   });
}
```

```
Observer1 got value: 0
Observer1 got value: 1
Observer1 got value: 2
Observer1 got value: 3
Observer1 got value: 4
Observer1 got value: 5
Observer1 got value: 6
Observer1 got value: 7
Observer1 got value: 8
Observer1 got value: 9
Observer1 completed!

Observer2 got value: 0
Observer2 got value: 1
Observer2 got error: Something went wrong
```

## Async:

```
export function createObservable(n : number)
{
   return new Observable( subscribe: observer => {
      for (let i = 0; i < 10; i++) {
         if (i === n) setTimeout( callback: () => observer.error('Something went wrong'), i);

         setTimeout( callback: () => observer.next(i), i);
      }
      setTimeout( callback: () => observer.complete(),  ms: 10);
   });
}
```

```
Observer1 got value: 0
Observer1 got value: 1
Observer2 got value: 0
Observer2 got value: 1
Observer1 got value: 2
Observer2 got error: Something went wrong
Observer1 got value: 3
Observer1 got value: 4
Observer1 got value: 5
Observer1 got value: 6
Observer1 got value: 7
Observer1 got value: 8
Observer1 got value: 9
Observer1 completed!
```

# Lazy vs Eager execution

```
export function createPromise() : Promise<string>
{
   return new Promise( executor: resolve => {
      console.log('Started');
      resolve( value: 'Reply');
   });
}

export const promise = createPromise();

console.log('Attaching to promises');

promise.then(
   value => console.log('Promise1 got value: ' + value),
   error => console.log('Promise1 got error: ' + error)
).finally( onfinally: () => console.log('Promise1 completed!\n'));

promise.then(
   value => console.log('Promise2 got value: ' + value),
   error => console.log('Promise2 got error: ' + error)
).finally( onfinally: () => console.log('Promise2 completed!\n'));
```

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

export function createObservable()
{
   return new Observable( subscribe: observer => {
      console.log('Started');
      observer.next( value: 'Reply');
      observer.complete();
   });
}

const obs$ = createObservable();

console.log('Attaching to observables');

obs$.subscribe(
    next: value => console.log('Observer1 got value: ' + value),
    error: error => console.log('Observer1 got error: ' + error),
    complete: () => console.log('Observer1 completed!\n')
);

obs$.subscribe(
    next: value => console.log('Observer2 got value: ' + value),
    error: error => console.log('Observer2 got error: ' + error),
    complete: () => console.log('Observer2 completed!\n')
);
```

# Lazy vs Eager execution

```ts
export function createPromise() : Promise<string>
{
   return new Promise( executor: resolve => {
      console.log('Started');
      resolve( value: 'Reply');
   });
}

export const promise = createPromise();

console.log('Attaching to promises');

promise.then(
   value => console.log('Promise1 got value: ' + value),
   error => console.log('Promise1 got error: ' + error)
).finally( onfinally: () => console.log('Promise1 completed!\n'));

promise.then(
   value => console.log('Promise2 got value: ' + value),
   error => console.log('Promise2 got error: ' + error)
).finally( onfinally: () => console.log('Promise2 completed!\n'));
```

```
Started
Attaching to promises
Promise1 got value: Reply
Promise2 got value: Reply
Promise1 completed!

Promise2 completed!
```

```ts
import { Observable } from 'rxjs';

export function createObservable()
{
   return new Observable( subscribe: observer => {
      console.log('Started');
      observer.next( value: 'Reply');
      observer.complete();
   });
}

const obs$ = createObservable();

console.log('Attaching to observables');

obs$.subscribe(
    next: value => console.log('Observer1 got value: ' + value),
    error: error => console.log('Observer1 got error: ' + error),
    complete: () => console.log('Observer1 completed!\n')
);

obs$.subscribe(
    next: value => console.log('Observer2 got value: ' + value),
    error: error => console.log('Observer2 got error: ' + error),
    complete: () => console.log('Observer2 completed!\n')
);
```

```
Attaching to observables
Started
Observer1 got value: Reply
Observer1 completed!

Started
Observer2 got value: Reply
Observer2 completed!
```

# Operators

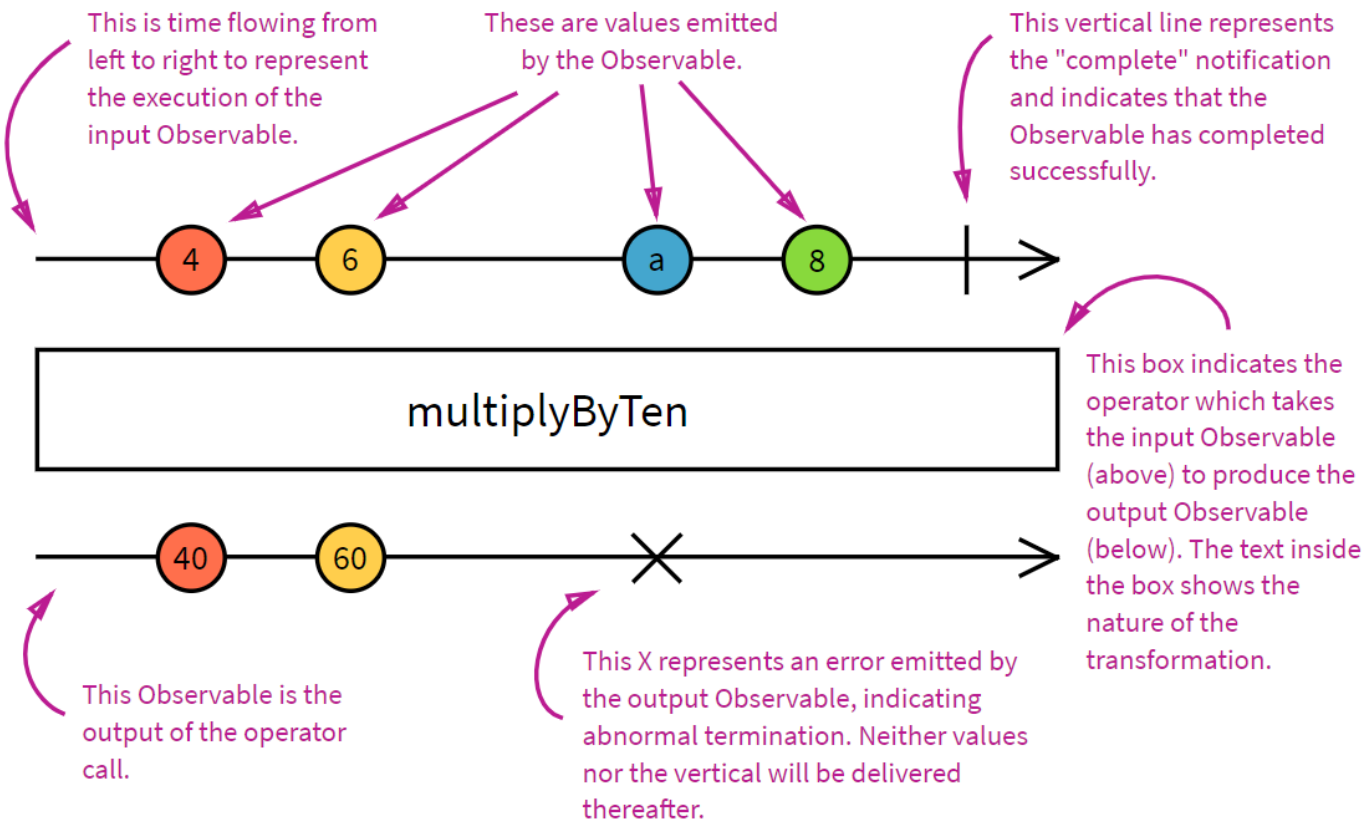
# Operators

*“Think of RxJS as Lodash for events”* (c) rxjs.dev

- Operators – are pure function on observables.
- With observable *obs$*, and operators *op1()*, op2(), op3(), op4(), we can:
  - *op1*()(*obs$*)
  - op2()(*op1*()(*obs$*))
  - op4()(op3()(op2()(*op1*()(*obs$*))))
- But we can simplify with pipe():

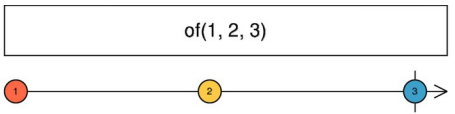
```
obs$.pipe(
    op1(),
    op2(),
    op3(),
    op4()
);
```

# Marble diagram

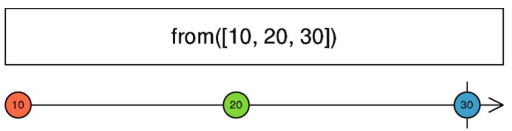


# Basic operators

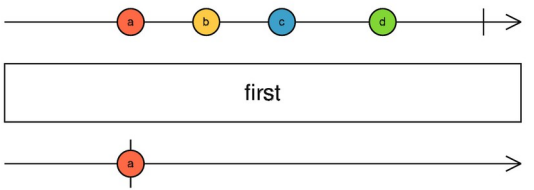
- *of*
  - returns observable from value
  - values, comma separated emits different values,
  - Note difference: of(1, 2, 3) vs of([1, 2, 3])



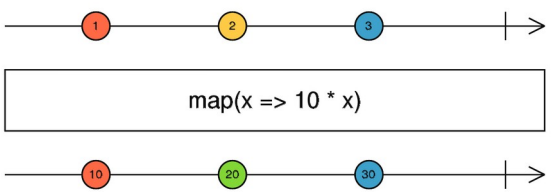

- from: produce observables from
  - Promise
  - Array




- *first*
  - takes first value from observables and complete it




- *tap*
  - perform some function over observable value, without modification to stream
- *map*
  - maps emited values to new ones



# Basic operators

```
import { from, of } from 'rxjs';
import { first, map, tap } from 'rxjs/operators';

const obs1$ = of( a: 'a', b: 'b', c: 'c').pipe(
   tap( next: alpha => console.log(alpha)),
   map( project: alpha => alpha.repeat( count: 10)),
   tap( next: message => console.log(message)),
   first()
);

const obs2$ = from( input: [1, 2, 3]).pipe(
   tap( next: value => console.log(value)),
   map( project: value => value * value * value),
   tap( next: value => console.log(value))
);


console.log('Observer 1:');
obs1$.subscribe( next: value => console.log('Observer1 says: ' + value));

console.log('\nObserver 2:');
obs2$.subscribe( next: value => console.log('Observer2 says: ' + value));
```

# Basic operators

```
import { from, of } from 'rxjs';
import { first, map, tap } from 'rxjs/operators';

const obs1$ = of( a: 'a', b: 'b', c: 'c').pipe(
   tap( next: alpha => console.log(alpha)),
   map( project: alpha => alpha.repeat( count: 10)),
   tap( next: message => console.log(message)),
   first()
);

const obs2$ = from( input: [1, 2, 3]).pipe(
   tap( next: value => console.log(value)),
   map( project: value => value * value * value),
   tap( next: value => console.log(value))
);
```

```
Observer 1:
a
aaaaaaaaaa
Observer1 says: aaaaaaaaaa

Observer 2:
1
1
Observer2 says: 1
2
8
Observer2 says: 8
3
27
Observer2 says: 27
```

```
console.log('Observer 1:');
obs1$.subscribe( next: value => console.log('Observer1 says: ' + value));

console.log('\nObserver 2:');
obs2$.subscribe( next: value => console.log('Observer2 says: ' + value));
```

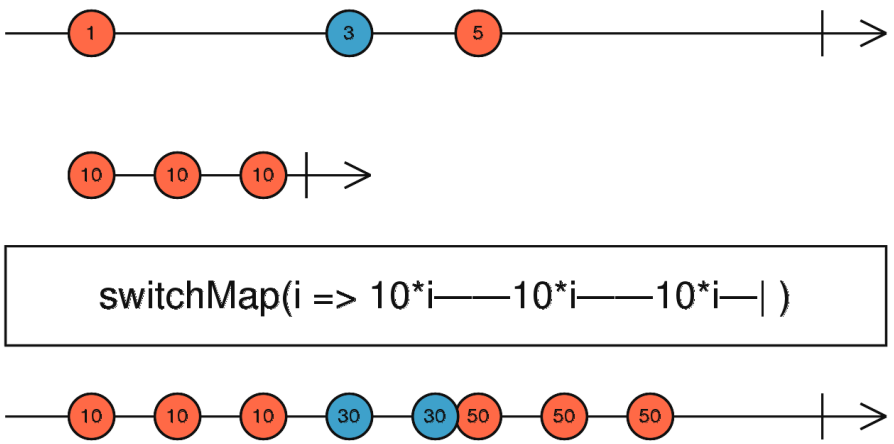
# Simplifying creation

```
import { from, Observable, of, range } from 'rxjs';

export const obs1$ = of( a: 1,  b: 2,  c: 3);
export const obs2$ = from( input: [1, 2, 3]);
export const obs3$ = range( start: 1,  count: 3);


export const subscribe = (obs$ : Observable<number>, name : string) =>
   obs$.subscribe(
       next: value => console.log(`${name} got value: ${value}`),
       error: error => console.log(`${name} got error: ${error}`),
       complete: () => console.log(`${name} completed!\n`)
   );


subscribe(obs1$,  name: 'Observer1');
subscribe(obs2$,  name: 'Observer2');
subscribe(obs3$,  name: 'Observer3');
```

```
Observer1 got value: 1
Observer1 got value: 2
Observer1 got value: 3
Observer1 completed!

Observer2 got value: 1
Observer2 got value: 2
Observer2 got value: 3
Observer2 completed!

Observer3 got value: 1
Observer3 got value: 2
Observer3 got value: 3
Observer3 completed!
```

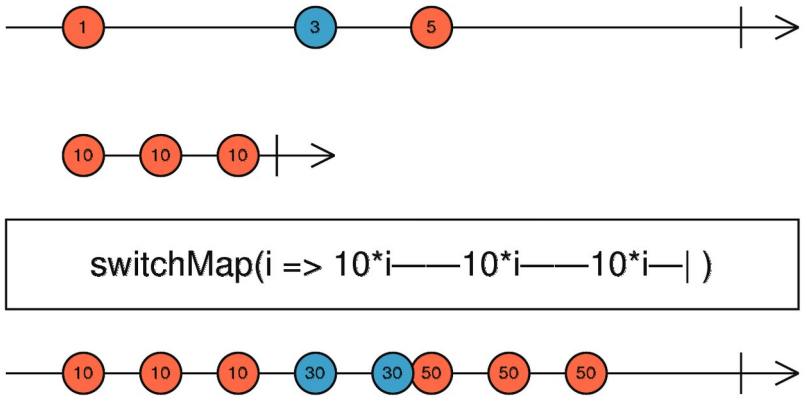
# switchMap

- Previously we have seen some
  - Creation operators (*of*, *from*, *range*, *etc*)
  - Filtering operators (*first*)
  - Utility operators (*tap*)
  - Simple transformation operators (*map*)
- None of them was complex, but there are plenty of complex operators
  - First on the way is *SwitchMap*



# switchMap

```
import { from, of } from 'rxjs';
import { map, switchMap } from 'rxjs/operators';

const obs1$ = from( input: [1, 2, 3]).pipe(
   map( project: value => value * value * value)
);

const obs2$ = from( input: [1, 2, 3]).pipe(
   switchMap( project: value => of( a: value * value * value))
);

obs1$.subscribe( next: value => console.log('Observer1 says: ' + value));
obs2$.subscribe( next: value => console.log('Observer2 says: ' + value));
```




```
Observer1 says: 1
Observer1 says: 8
Observer1 says: 27
Observer2 says: 1
Observer2 says: 8
Observer2 says: 27
```

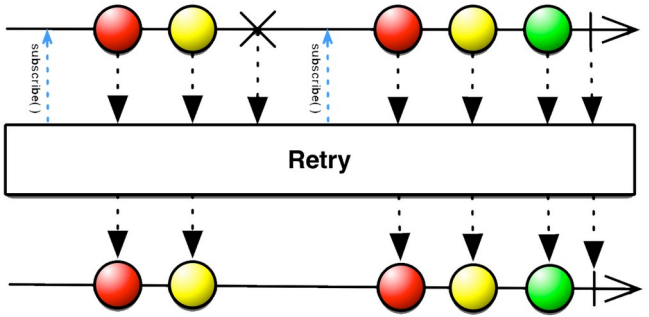
# Error / catchError / throwError

- If error is met observable switches to error stream
- Normal operators on the stream omitted
- CatchError can return observable on main stream
- If error goes to subscriber (observer) the observable becomes closed
- There is example...

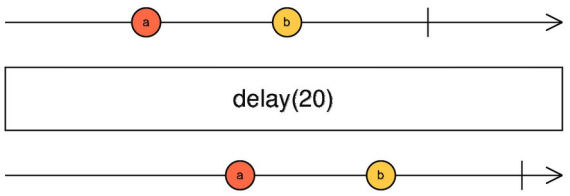
https://medium.com/sodalabs/rxjava-handling-errors-like-a-pro-8e403dd2126a

# retry / delay / retryWhen

- retry
  - attempts to retry observable on error

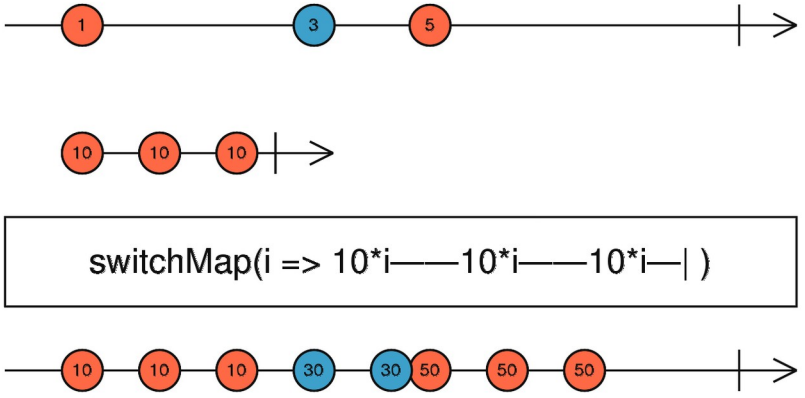


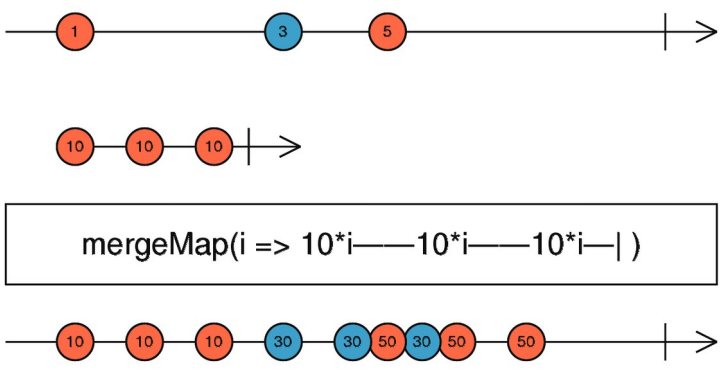

- delay

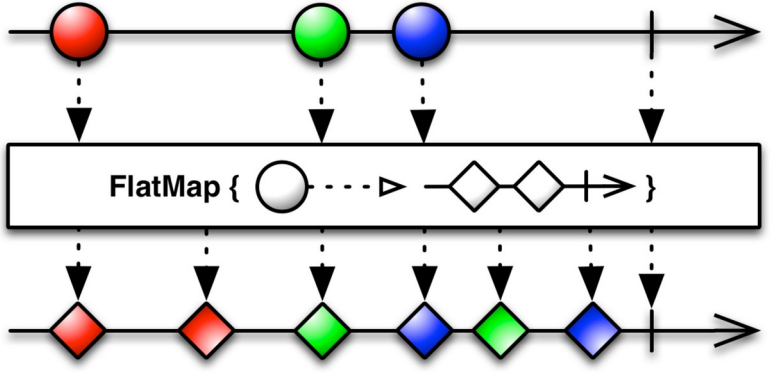



- retryWhen – example of very confusing definition:
  - Returns an Observable that mirrors the source Observable with the exception of an error. If the source Observable calls error, this method will emit the Throwable that caused the error to the Observable returned from notifier. If that Observable calls complete or error then this method will call complete or error on the child subscription. Otherwise this method will resubscribe to the source Observable.
  - better read: https://medium.com/angular-in-depth/retry-failed-http-requests-in-angular-f5959d486294

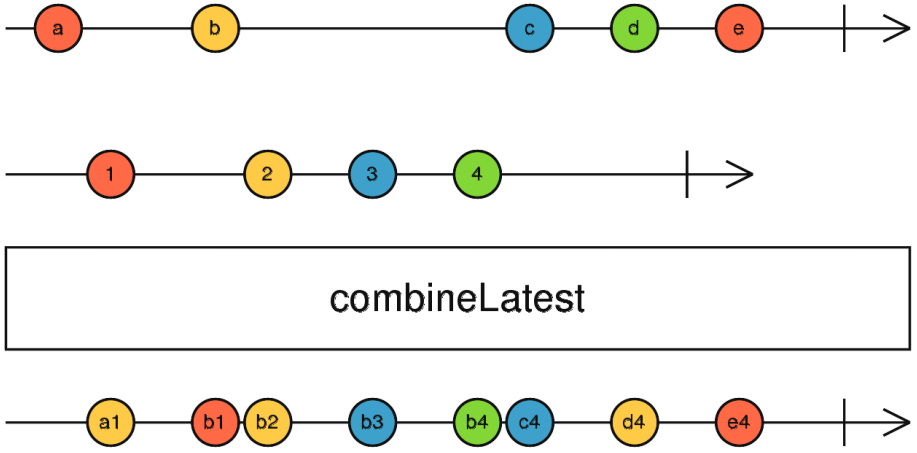
# SwitchMap vs FlatMap (mergeMap)

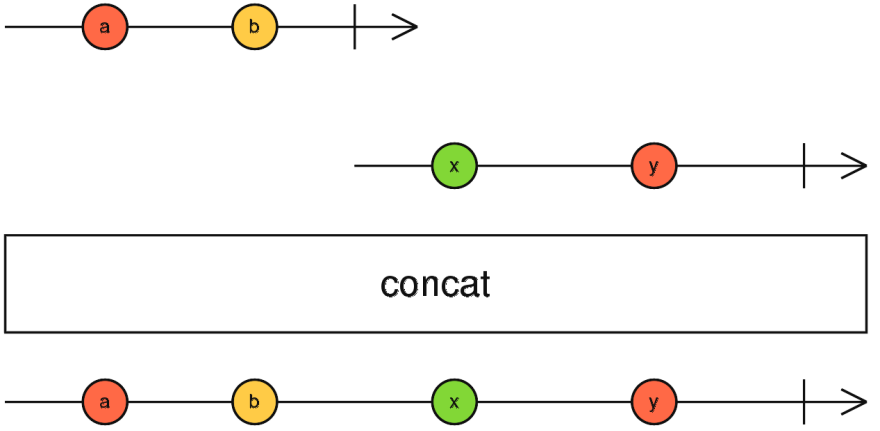


# CombineLatest and concat



# Subject

# Subject

I lied a little bit …

… when I mentioned observer pattern.

# Observer pattern

Design:

```
export interface Observer {
   update : (value : string) => void;
}

export class Subject {

   observers : Observer[] = [];

   addObserver(observer : Observer)
   {
      this.observers.push(observer);
   }

   removeObserver(observer : Observer)
   {
      const index = this.observers.indexOf(observer);
      if (index !== -1) this.observers.splice(index, deleteCount: 1);
   }

   notify(value : string)
   {
      this.observers.forEach(observer => observer.update(value));
   }
}
```

Usage:

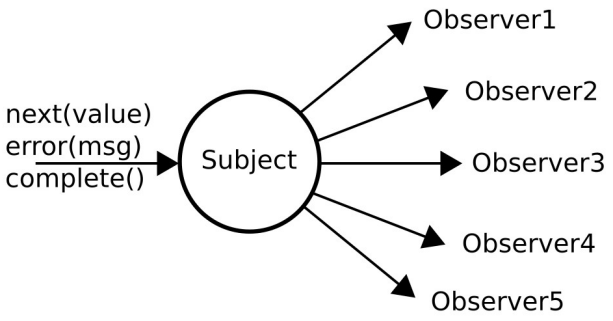
```
const subj = new Subject();
const obs1 : Observer = {update : value => console.log('obs1: ' + value)};
const obs2 : Observer = {update : value => console.log('obs2: ' + value)};

subj.addObserver(obs1);
subj.addObserver(obs2);
subj.notify( value: 'First event for both');
subj.notify( value: 'Second event for both');

subj.removeObserver(obs1);
console.log('\nobs1 removed');
subj.notify( value: 'Third event only for one');
```

```
obs1: First event for both
obs2: First event for both
obs1: Second event for both
obs2: Second event for both

obs1 removed
obs2: Third event only for one
```

# Subject

Subject

- Acts as Observable
- Acts as Observer




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

const subject = new Subject<number>();

const subscription = subject.subscribe( next: value => console.log('obs1: ' + value));

subject.next( value: 1);
subject.next( value: 2);

subject.subscribe( next: value => console.log('obs2: ' + value));

subject.next( value: 3);
subject.next( value: 4);

subscription.unsubscribe();

subject.next( value: 5);

subject.complete();
subject.error('Oops');
```

```
obs1: 1
obs1: 2
obs1: 3
obs2: 3
obs1: 4
obs2: 4
obs2: 5
```

# Subject

Rxjs has following types of Subjects:

- Subject
  - normal subject
- BehaviorSubject
  - stores latest value
  - has “hidden” problem: cannot distinguish initial from stored latest value
- ReplaySubject
  - records multiple values
    - const subject = new ReplaySubject(3);
  - can have time frame
    - const subject = new ReplaySubject(100, 500 /* windowTime */);
- AsyncSubject

# Pros / Cons

Cons:

- No cons for me, but generally...
- Time to study
- Many operators
- Careful memory management
  - in some cases memory leak
- Naming inconsistency
- Big change from rxjs5 to rxjs6

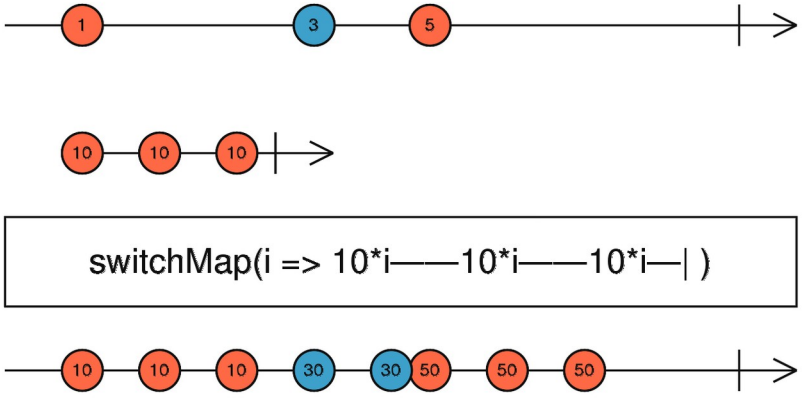
Pros:

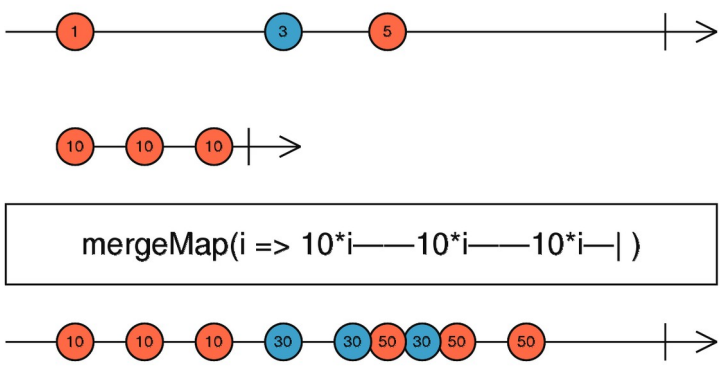
- Easy to control
  - Control of logic not construct helping operations
- Easy to extend
- Many operators
- Lazy execution
- Error handling

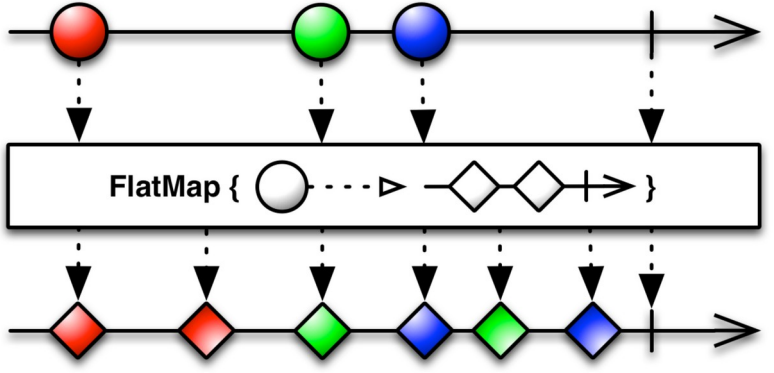
Thank You

# Backup slides

# SwitchMap vs FlatMap (mergeMap)



# Hot / cold observables

- Cold observables, which produce value from observable.
- Hot observable which produces value outside.
  - Hot observable can be shared

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

const getCold$ = () => new Observable( subscribe: observer => {
   observer.next(Math.random());
   observer.complete();
});

getCold$().subscribe( next: value => console.log('Cold observable1: ' + value));
getCold$().subscribe( next: value => console.log('Cold observable2: ' + value));

const hotObs$ = of(Math.random());

hotObs$.subscribe( next: value => console.log('Hot observable1: ' + value));
hotObs$.subscribe( next: value => console.log('Hot observable2: ' + value));
```

```
Cold observable1: 0.4023136180209881
Cold observable2: 0.6075014168244075
Hot observable1: 0.47822946640088704
Hot observable2: 0.47822946640088704
```

https://medium.com/@luukgruijs/understanding-hot-vs-cold-observables-62d04cf92e03

# Memory leakage

- const subscription = obs$.subscribe();
- subscription.unsubscribe();

# Old usage

- Previously (before rxjs6) operators were chained:
  - obs$.op1().op2().op3().op4()
  - Some Internet code still shows examples like that
  - You can still write this style with `rxjs-compat` module.
- It was decided to switch to “more” pure functions and now all operations are not chainable, instead pipe is used

# What is not covered

- Scheduler
- Marble testing
- Full list of operators
  - List: https://rxjs.dev/guide/operators
  - Tool: https://rxjs.dev/operator-decision-tree