ANGULAR

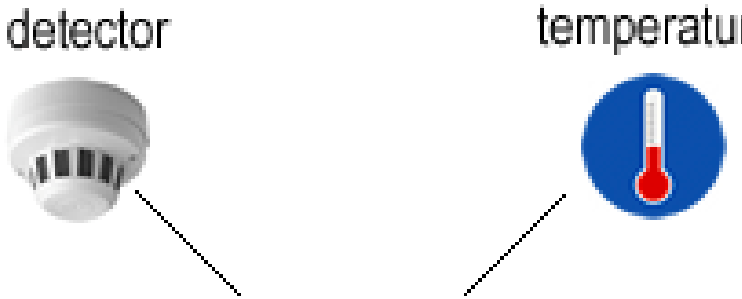
RXJS IN ANGULAR: BASICS

LUXOFT

LXFT LISTED NYSE

RXJS

RXJS Reactive Programming



Observable variables

alarm.active = detector > X && temperature >Y

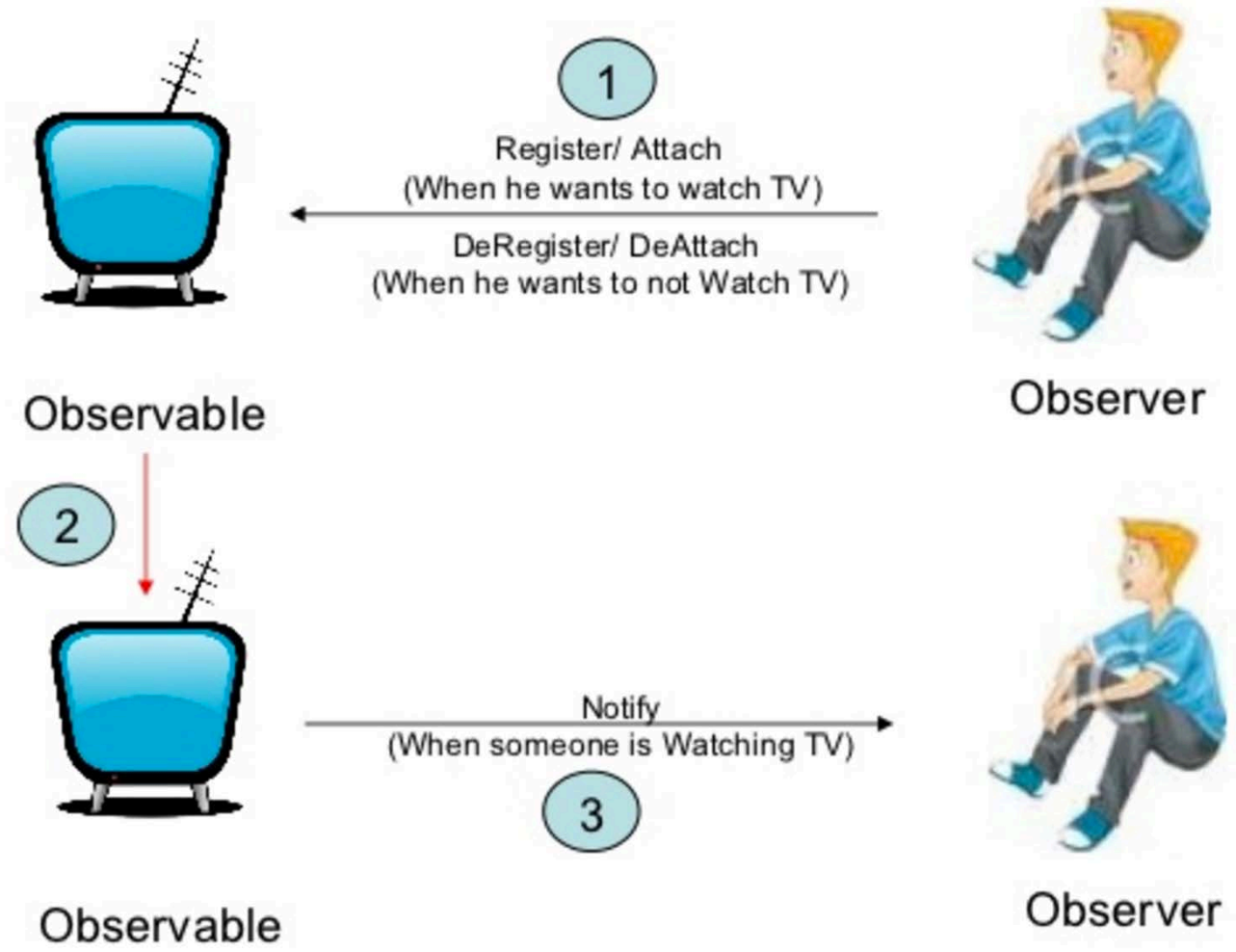
Reactive variable

angulartypescript.com

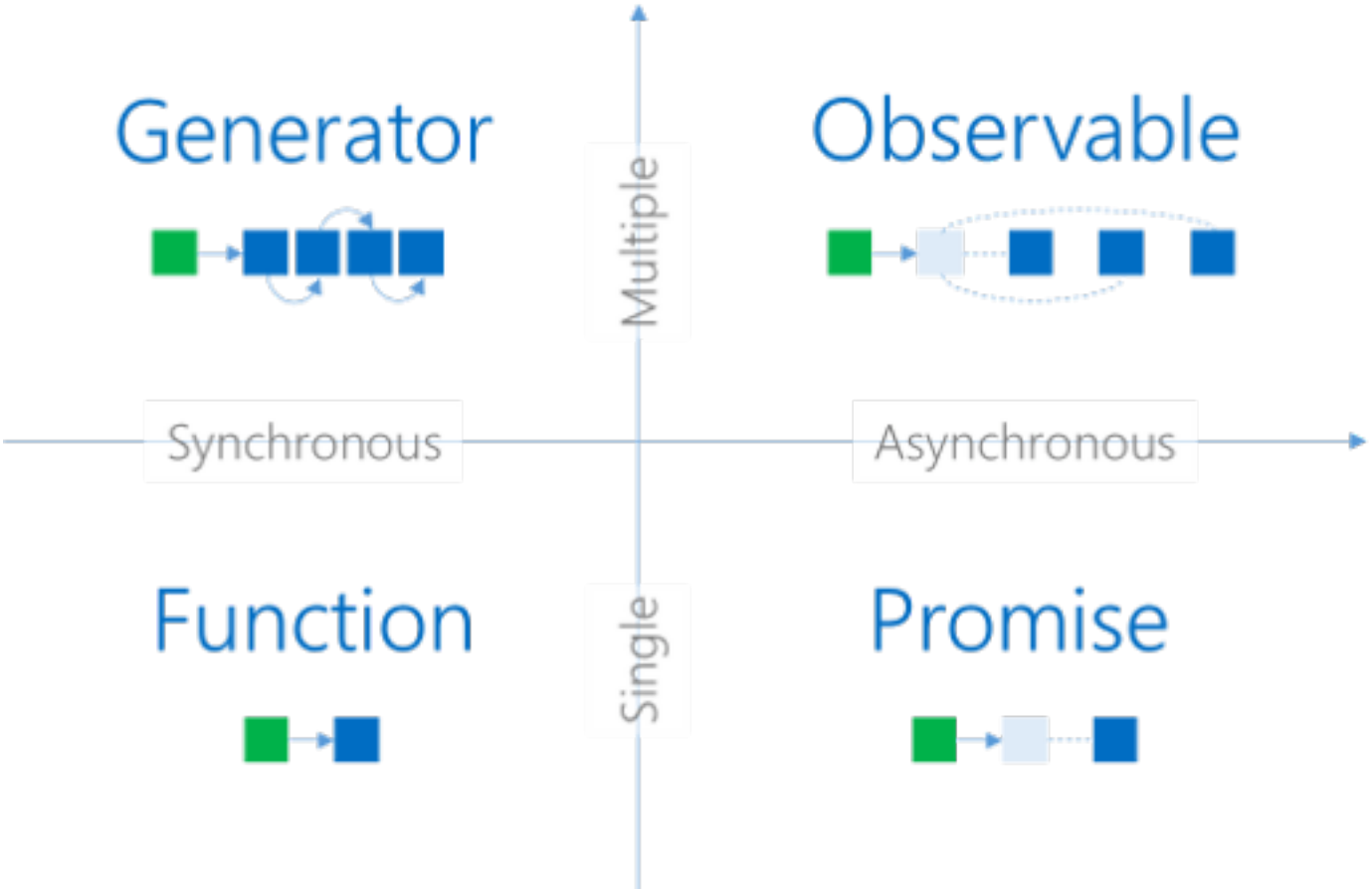
CREATE AND SUBSCRIBE OBSERVABLE

```
var observable = Observable.create(function (observer) {
    observer.next(42);
    observer.next(42);
    observer.complete();
});

var subscription = observable.subscribe(
    function (value) {
        console.log('Next: %s.', value);
    },
    function (ev) {
        console.log('Error: %s!', ev);
    },
    function () {
        console.log('Completed!');
    }
);

subscription.dispose();
```

OBSERVABLE PATTERN



DATA PRODUCERS



HOW TO GET OBSERVABLE?

Observable creation helpers in RxJS

- Observable.of(value, value2, value3, …)

```
Rx.Observable.of(1,2,3);
```

- Observable.from(promise/iterable/observable)

```
Rx.Observable .from([1,1,2,2,3,1,2,3]);
```

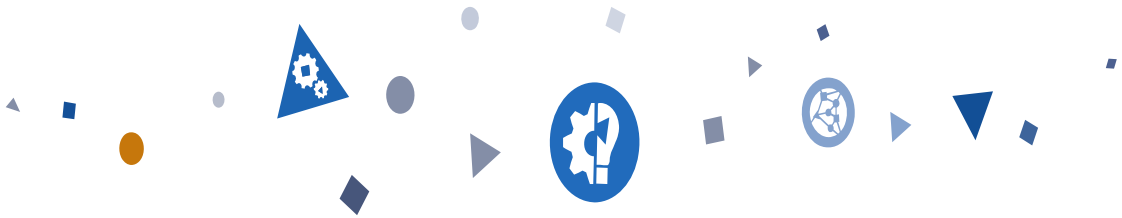
- Observable.fromEvent(item, eventName)

```
Rx.Observable.fromEvent(document, 'click');
```

- Angular’s HTTP and Realtime Data services

```
constructor(private http: Http) {
      this.http.get("server.com/data")
}
```

- Many community-driven RxJS modules and libraries



OBSERVABLES ERROR HANDLING

OBSERVABLE: ERROR HANDLING

```
var observable = Observable.create(function (observer) {
    observer.next(42);
    observer.next(42);
    observer.error("fail!");
    observer.complete();
});

var subscription = observable.subscribe(
    function (value) { console.log('Next: %s.', value); },
    function (ev) { console.log('Error: %s!', ev); },
    function () { console.log('Completed!'); }
);

// Results
Next: 42
Next: 42
Error: fail!
```

```
interface Observer<T> {
    next(value: T) : void
    error(error: Error) : void
    complete() : void
}
```

OBSERVABLE: CATCH

```
var source = Rx.Observable.catch(
    get('url1'),
    get('url2'),
    get('url3'),
    getCachedVersion()
);

var subscription = source.subscribe(
    data => {
        // Display the data as it comes in
    }
);
```

OBSERVABLE: CATCH AS OPERATOR

```
var source = get('url1').catch(e => {
    if (e.status === 500) {
        return cachedVersion();
    } else {
        return get('url2');
    }
});

var subscription = source.subscribe(
    data => {
        // Display the data as it comes in
    }
);
```

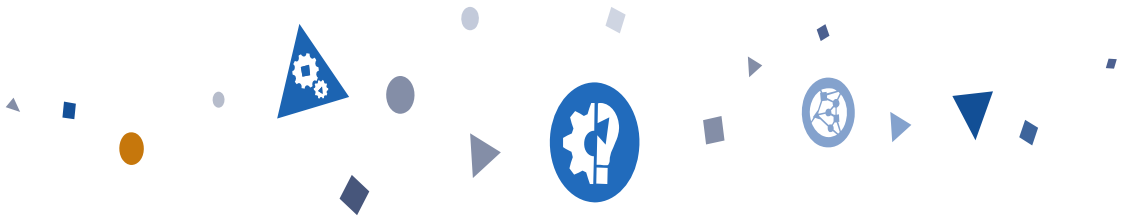
IGNORING ERRORS WITH ONERRORRESUMENEXT

onErrorResumeNext:
when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred, and execution continues from that point

```
var source = Rx.Observable.onErrorResumeNext(
    Rx.Observable.just(42),
    Rx.Observable.throw(new Error()),
    Rx.Observable.just(56),
    Rx.Observable.throw(new Error()),
    Rx.Observable.just(78)
);

var subscription = source.subscribe(
    data => console.log(data)
);
// => 42
// => 56
// => 78
```

RETRYING

```
// Try three times to get the data and then return cached data if still fails
var source = get('url').retry(3).catch(cachedVersion());

var subscription = source.subscribe(
   data => {
      // Displays the data from the URL or cached data
      console.log(data);
   }
);
```



PROMISE AND OBSERVABLE

COMPONENT WITH PROMISES

```
@Component({
   templateUrl: '../html/films.html'
})
export class FilmsComponent  {
   title: string = 'Films';
   filmsPromiseArray: FilmModel[] = new Array();
   errorMessage: string;
   constructor( private _http: Http) { }

   ngOnInit() {
      this.getFilmsPromise().then(items => this.filmsPromiseArray = items);
   }
   getFilmsPromise(): Promise<FilmModel[]> {
      return this._http.get('http://swapi.co/api/films')
         .toPromise()
         .then((response) => response.json().results);
   }
}
```

COMPONENT WITH OBSERVABLE

```
export class FilmsComponent {
    title: string = 'Films';
    films: FilmModel[];
    constructor(private _http: Http) { }
    ngOnInit() {
        this.getFilmsObservable()
            .subscribe(data => this.films = data);
    }
    getFilmsObservable(): Observable<FilmModel[]> {
        return this._http.get('http://swapi.co/api/films')
            .map((response: Response) => response.json() as FilmModel[])
            .do(data => console.log(JSON.stringify(data)))
            .catch(this.handleError);
    }
    private handleError(error: Response) {
        console.error(error);
        return Observable.throw(error.json().error || 'Server error');
    }
}
```

HOT AND COLD OBSERVABLES

- Hot observables are pushing even when we are not subscribed to them (e.g., UI events).
- Cold observables start pushing only when we subscribe. They start over if we subscribe again.

```
var obs = Observable.interval(500).take(5)
   .do(i => console.log("obs value "+ i) );

obs.subscribe(value => console.log(
      "observer 1 received " + value));

obs.subscribe(value => console.log(
      "observer 2 received " + value));
```

```
obs value 0
observer 1 received 0
obs value 0
observer 2 received 0

obs value 1
observer 1 received 1
obs value 1
observer 2 received 1
```

When we create a subscriber, we are setting up a whole new separate processing chain.

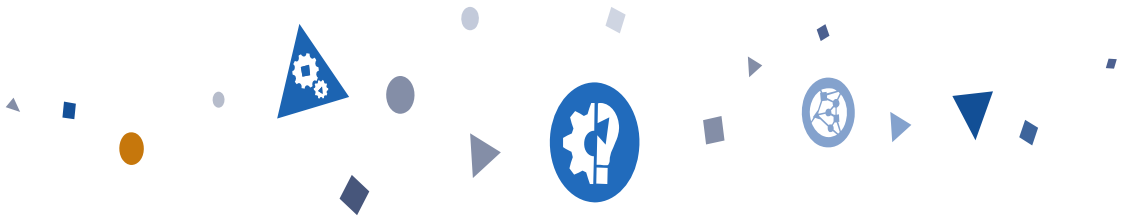
Observable is not shared: each subscriber get its own copy.

SHARE OPERATOR

The share operator allows to share a single subscription of a processing chain with other subscribers.

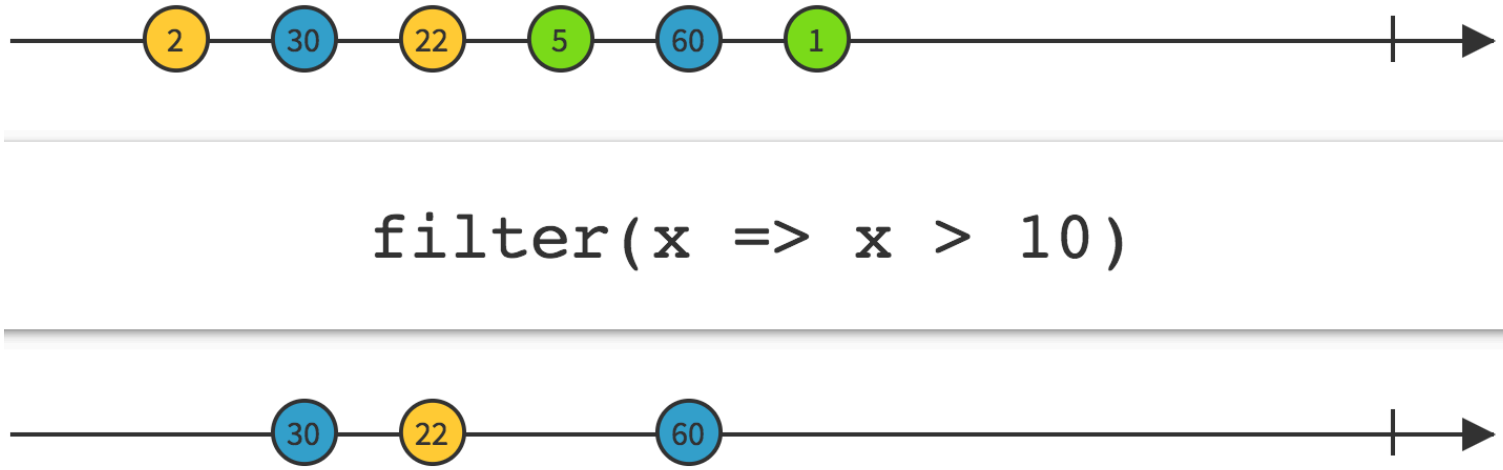
```
var obs = Observable.interval(500).take(5)
    .do(i => console.log("obs value "+ i) )
    .share();

obs.subscribe(value => console.log(
        "observer 1 received " + value));

obs.subscribe(value => console.log(
        "observer 2 received " + value));
```

```
obs value 0
observer 1 received 0
observer 2 received 0


obs value 1
observer 1 received 1
observer 2 received 1
```



POPULAR RXJS OPERATORS

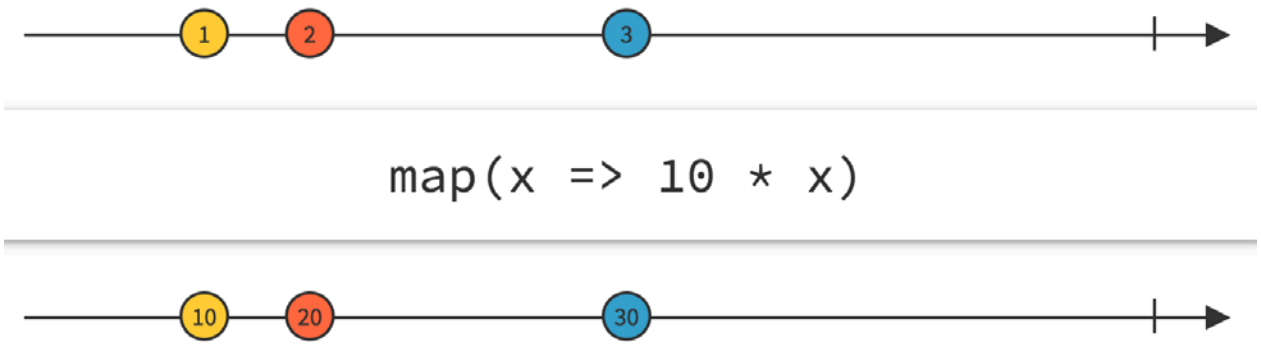
FILTER OPERATOR




```
var source = Observable.range(0, 5)
    .filter(function (x, idx, obs) {
        return x % 2 === 0;
    });

var subscription = source.subscribe(
    function (x) { console.log('Next: %s', x); },
    function (err) { console.log('Error: %s', err); },
    function () { console.log('Completed'); });
```

```
Next: 0
Next: 2
Next: 4
Completed
```

MAP OPERATOR

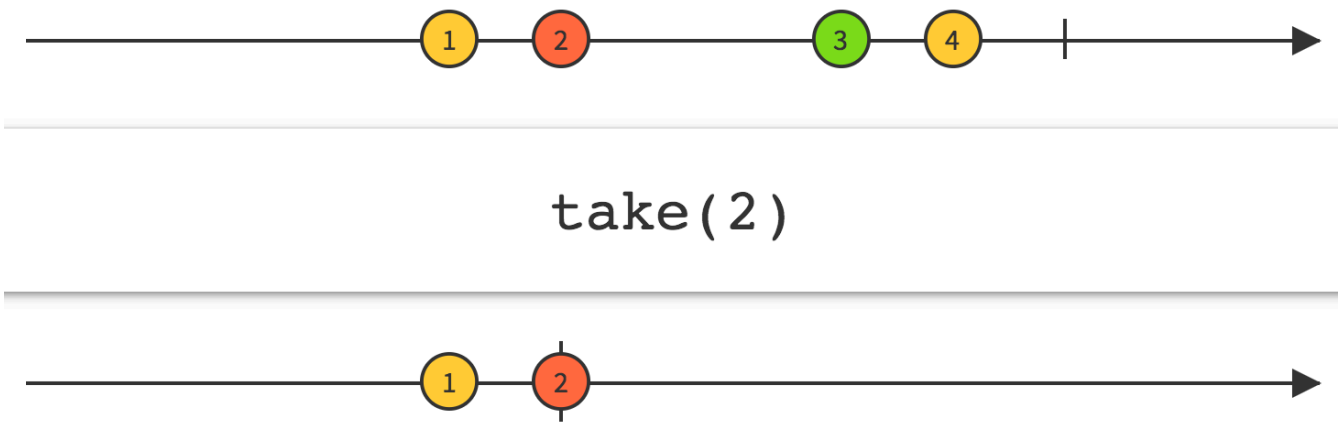



```
// Using a value
var md = Observable.fromEvent(document, 'mousedown')
        .map(e=>{ return { x:e.x, y: e.y}  });

var subscription = source.subscribe(
   function (a) { console.log('Mouseclick at (${a.x},${a,y}) '); },
   function (err) { console.log('Error: ' + err); },
   function () { console.log('Completed'); });
```

EXAMPLE: ANGULAR AJAX

```
export class FilmsComponent {
    title: string = 'Films';
    films: FilmModel[];
    constructor(private _http: Http) { }
    ngOnInit() {
        this.getFilmsObservable().subscribe(data => this.films = data);
    }
    getFilmsObservable(): Observable<FilmModel[]> {
        return this._http.get('http://swapi.co/api/films')
            .map((response: Response) => response.json() as FilmModel[])
            .do(data => console.log(JSON.stringify(data)))
            .catch(this.handleError);
    }
    private handleError(error: Response) {
        console.error(error);
        return Observable.throw(error.json().error || 'Server error');
    }
}
```

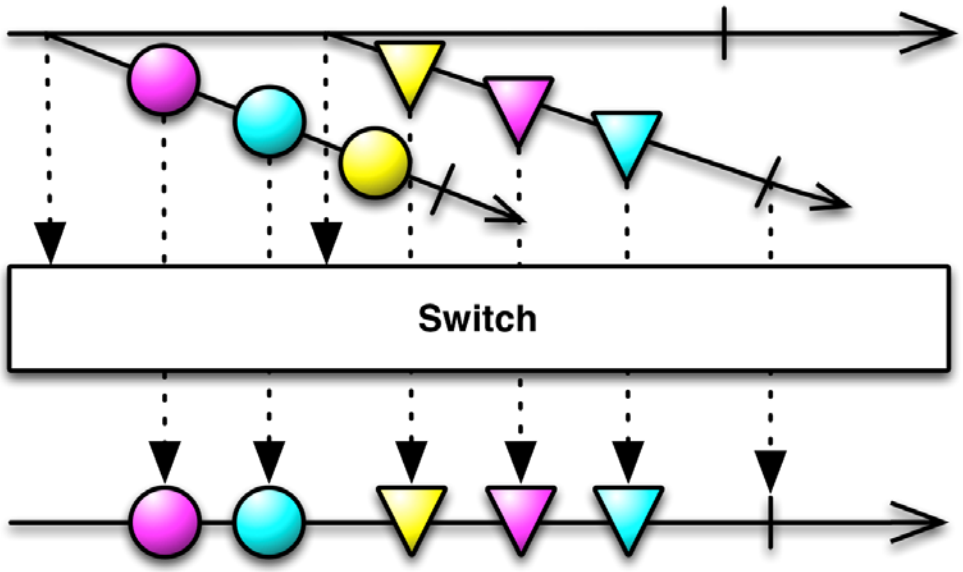
TAKE OPERATOR




```
var source = Observable.range(0, 5).take(3);

var subscription = source.subscribe(
    function (x) { console.log('Next: ' + x); },
    function (err) { console.log('Error: ' + err); },
    function () { console.log('Completed'); });
```

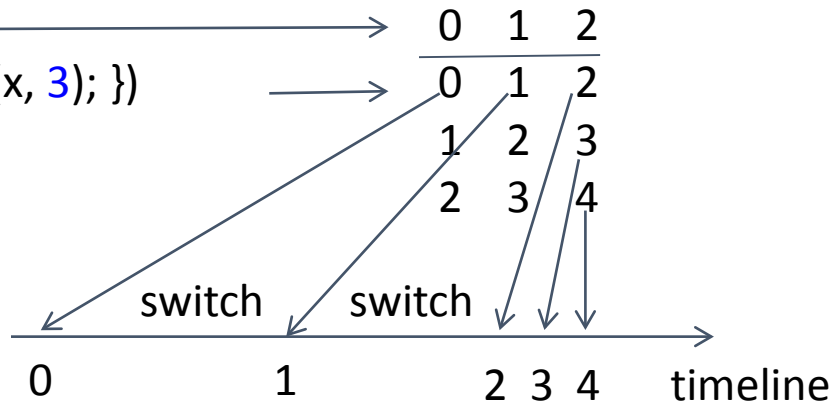
```
Next: 0
Next: 1
Next: 2
Completed
```

SWITCH OPERATOR




```
Next: 0
Next: 1
Next: 2
Next: 3
Next: 4
Completed
```

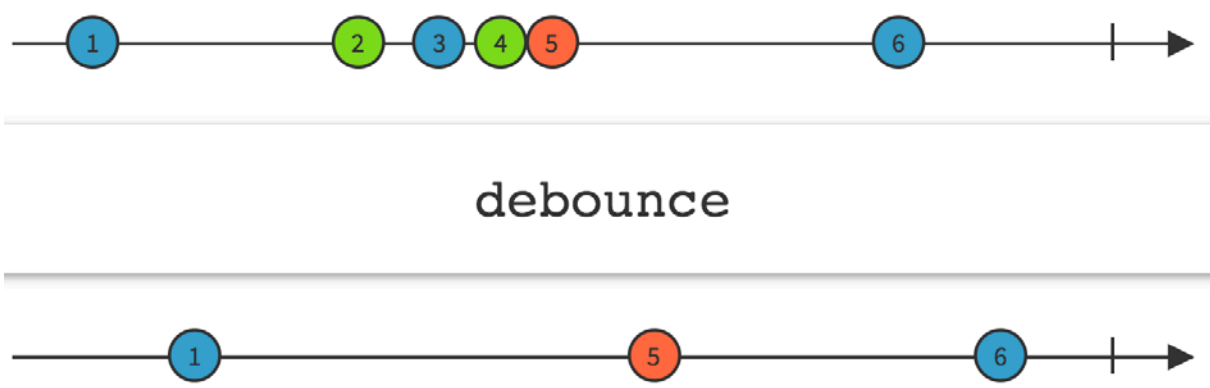
```
var source = Observable.range(0, 3)
    .map(function (x) { return Observable.range(x, 3); })
    .switch();

var subscription = source.subscribe(
    function (x) { console.log('Next: ' + x); },
    function (err) { console.log('Error: ' + err); },
    function () { console.log('Completed'); });
```



DEBOUNCE OPERATOR

The Debounce technique allow us to "group" multiple sequential calls in a single one.

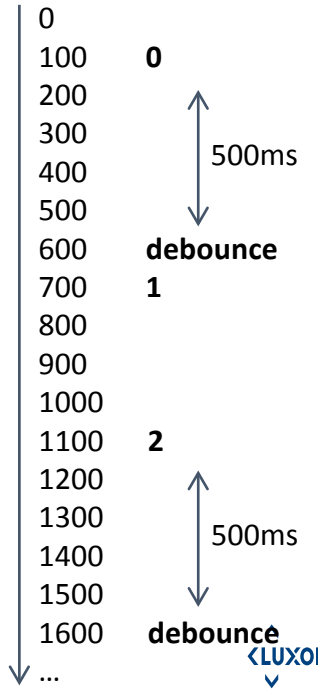
Debouncing enforces that a function not be called again until a certain amount of time has passed without it being called. As in "***execute this function only if 100 milliseconds have passed without it being called***."




```
var times = [
   { value: 0, time: 100 },
   { value: 1, time: 600 },
   { value: 2, time: 400 },
   { value: 3, time: 700 },
   { value: 4, time: 200 }
];
```

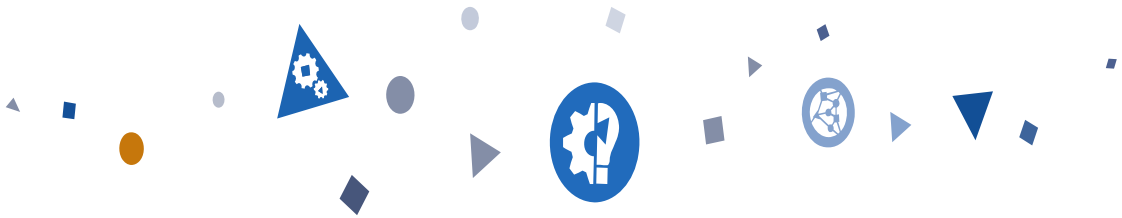
```
Next: 0
Next: 2
Next: 4
Completed
```

```
// Delay each item by time and project value;
var source = Observable.from(times)
   .flatMap(function (item) {
      return Rx.Observable
         .of(item.value)
         .delay(item.time);
   })
   .debounce(500 /* ms */);
var subscription = source.subscribe(
   (x)=>console.log('Next: %s', x));
```



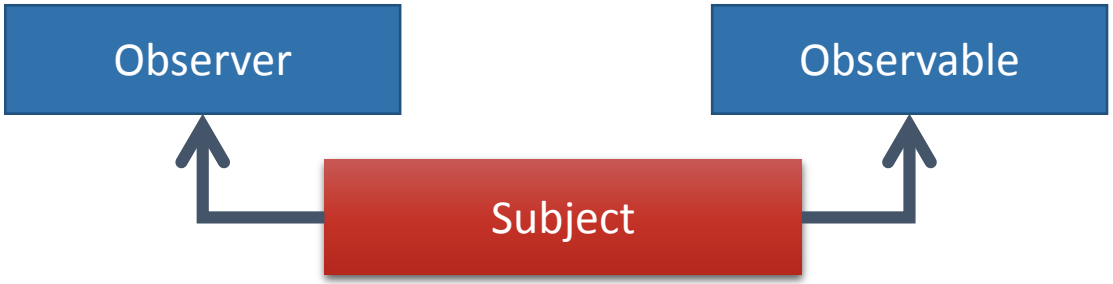
EXAMPLE: PROCESSING INPUT BOX CHANGES WITH RXJS

```
@Component({
   selector: 'my-app',
   template: `<input [formControl]="searchBox" />
             {{searchResults}}
          `
})
export class App {
   searchBox: FormControl = new FormControl();
   searchResults: string;

   constructor(httpService: HttpService) {
      this.searchBox.valueChanges
         .debounceTime(500)
         .distinctUntilChanged()
         .switchMap(data => this.httpService.getListValues(data))
         .subscribe(res=>this.searchResults=res,
               (err: Error) => console.log(err));
   }
}
```



SUBJECTS

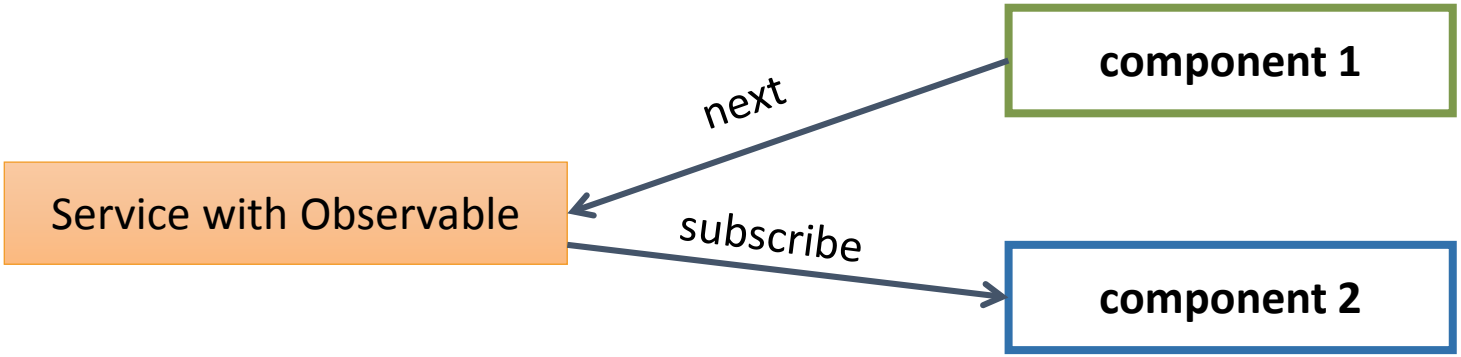
SUBJECT: OBSERVER AND OBSERVABLE




```
let subject = new Rx.Subject();

// acts as Observable
subject.subscribe(val => console.log(val));

// acts like Observer
subject.onNext(1);
subject.onCompleted();
```

Observable: Assume that a professor is an observable. The professor teaches about some topic.
Observer: Assume that a student is an observer. The student observes the topic being taught by the professor.

EXAMPLE: ANGULAR COMPONENT COMMUNICATION WITH RXJS




```
@Injectable()
export class MessageService {
   private messageSource = new Subject<Message>();
   messages$ = this.messageSource.asObservable();

   sendMessage(message: Message) {
      this.messageSource.next(message);
   }
}

export class Message {
   title: string;
   text: string;
}
```

```
constructor(
   private messageService: MessageService)
{
    this.messageService.messages$.subscribe(
        message => this.process(message))
}
```

Think about
how data should flow
instead of
what you do to make it flow