ANGULAR 2

OBSERVABLES

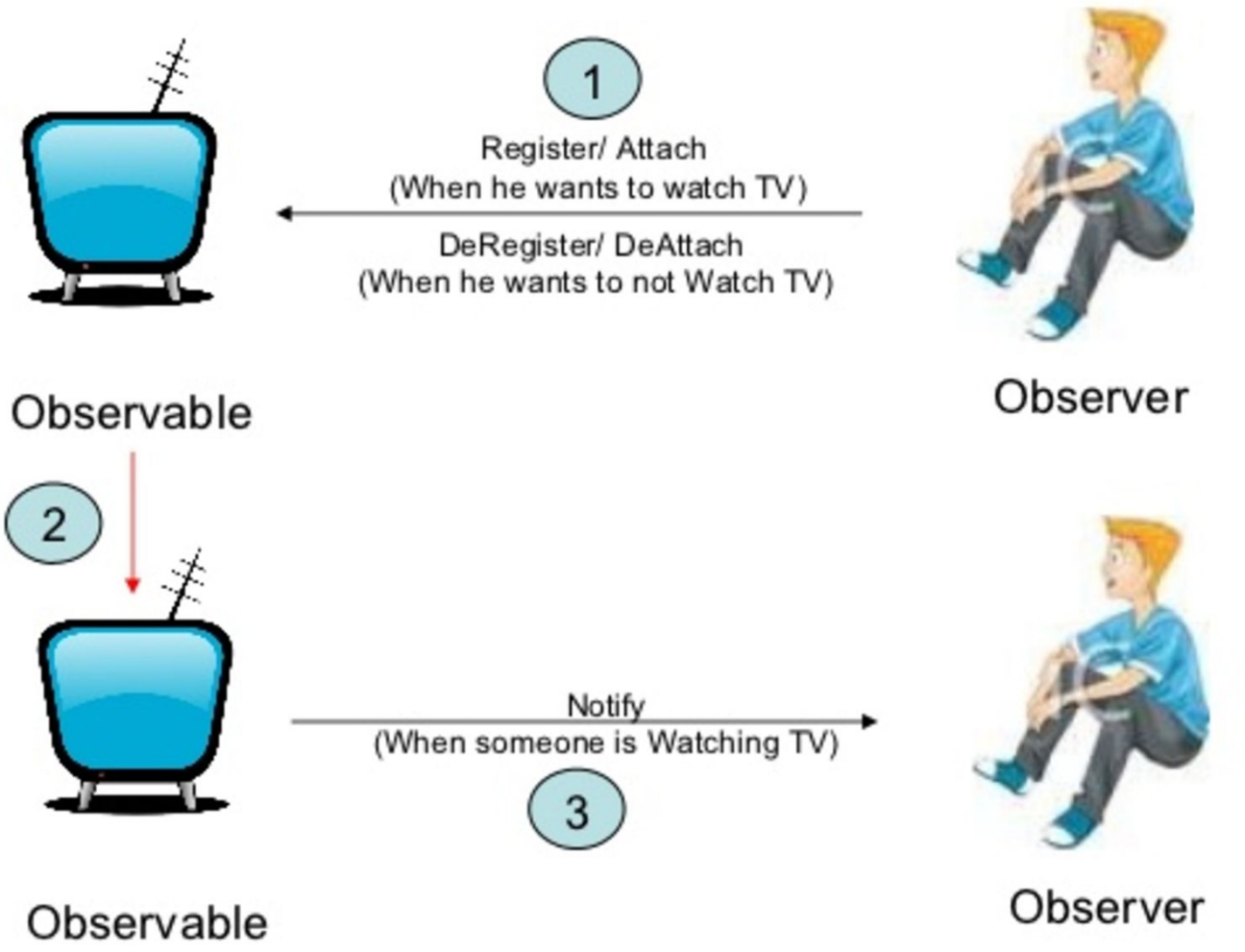
LUXOFT

LXFT LISTED NYSE

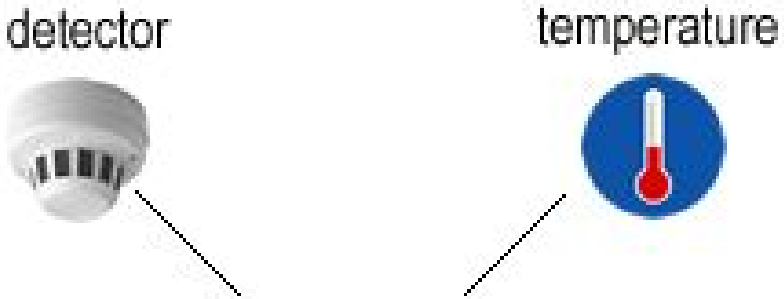
RXJS

RxJS is a library for composing asynchronous and event-based programs by using observable sequences.

OBSERVABLE PATTERN



RXJS Reactive Programming



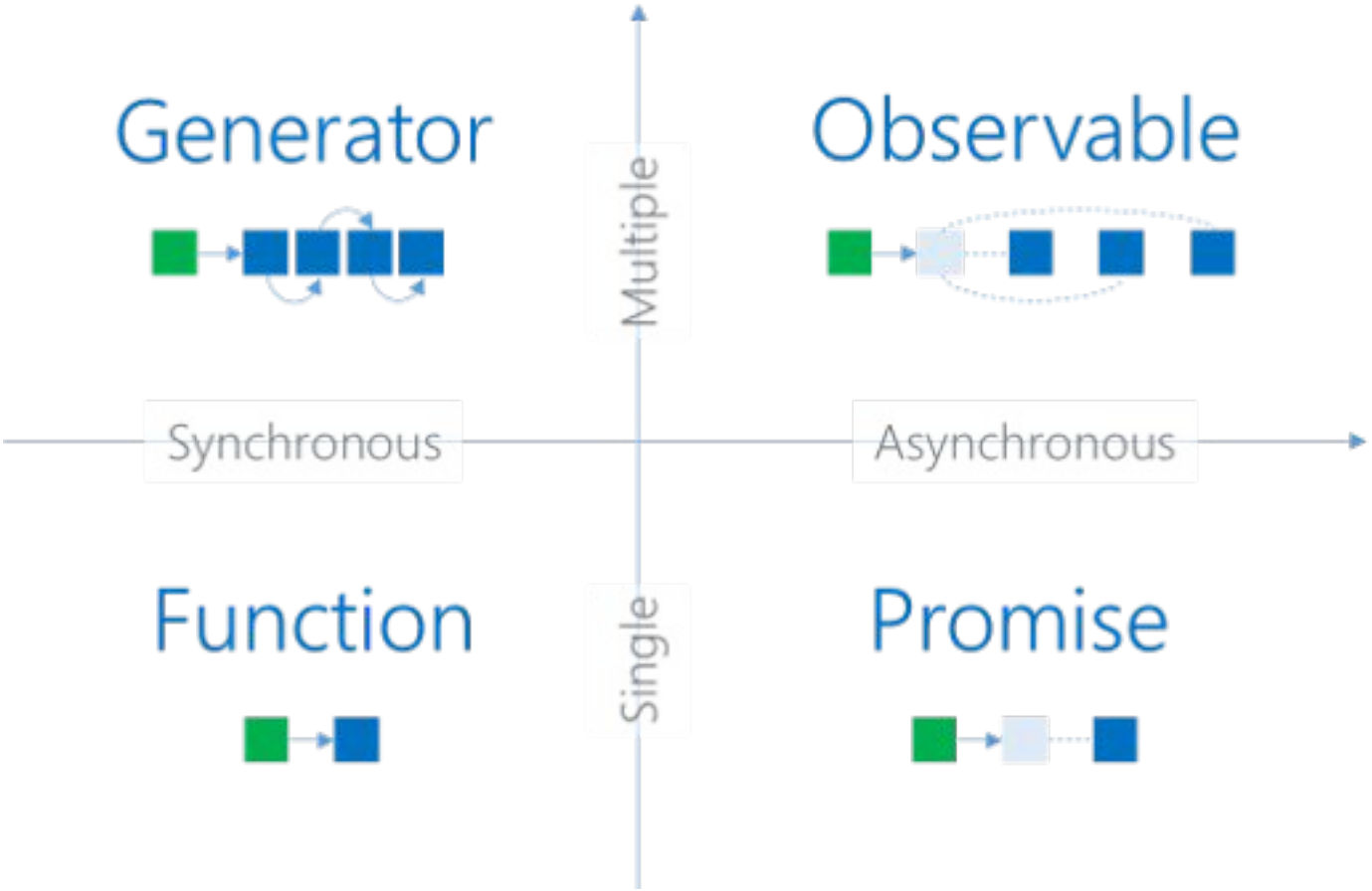
alarm.active = detector > X && temperature >Y

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();
```

DATA PRODUCERS



HOW TO GET OBSERVABLE?

Observable creation helpers in RxJS

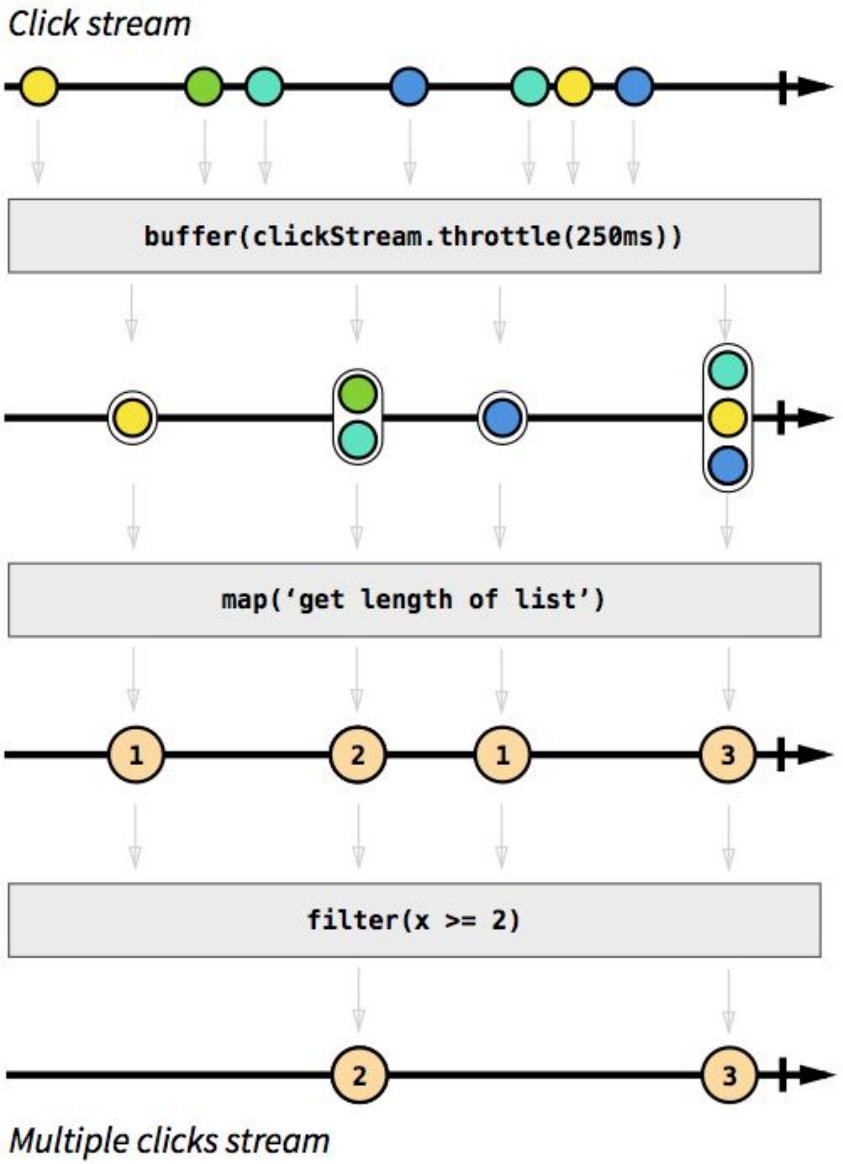
- Observable.of(value, value2, value3, …)
- Observable.from(promise/iterable/observable)
- Observable.fromEvent(item, eventName)
- Angular’s HTTP and Realtime Data services
- Many community-driven RxJS modules and libraries

OBSERVABLE ERROR HANDLING

```
myObservable.catch(error => {
   if (error instanceof OkayError) {
      return Observable.of("okay");
   } else {
      throw error;
   }
});
```

RXJS EXAMPLE

TASK:
Get stream of "double click" events. consider triple clicks as double clicks.



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));
```

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.

```
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
```

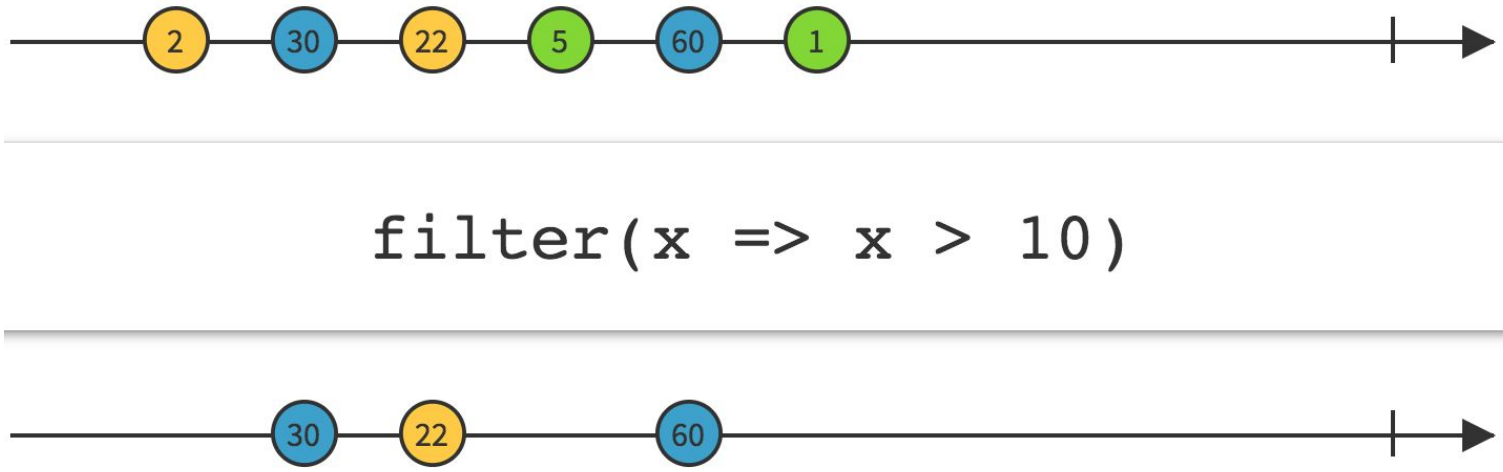
SHARE OPERATOR

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
```

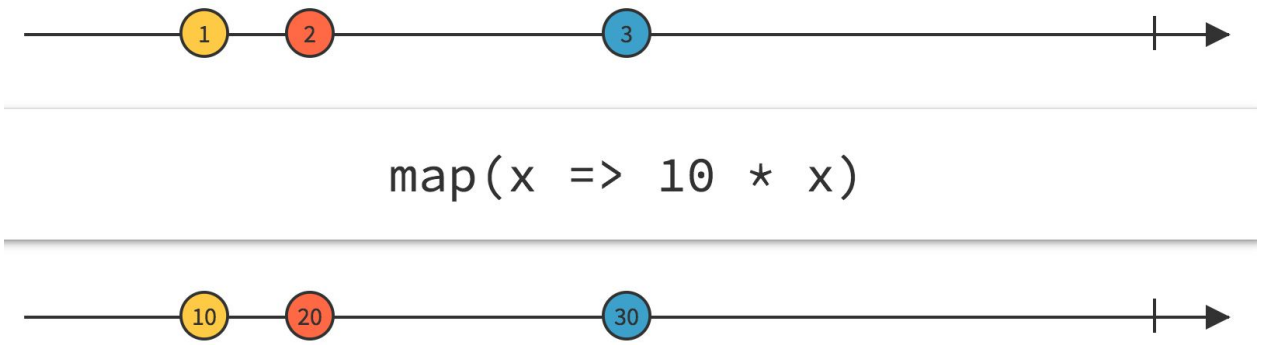
FILTER OPERATOR




```
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'); });
```

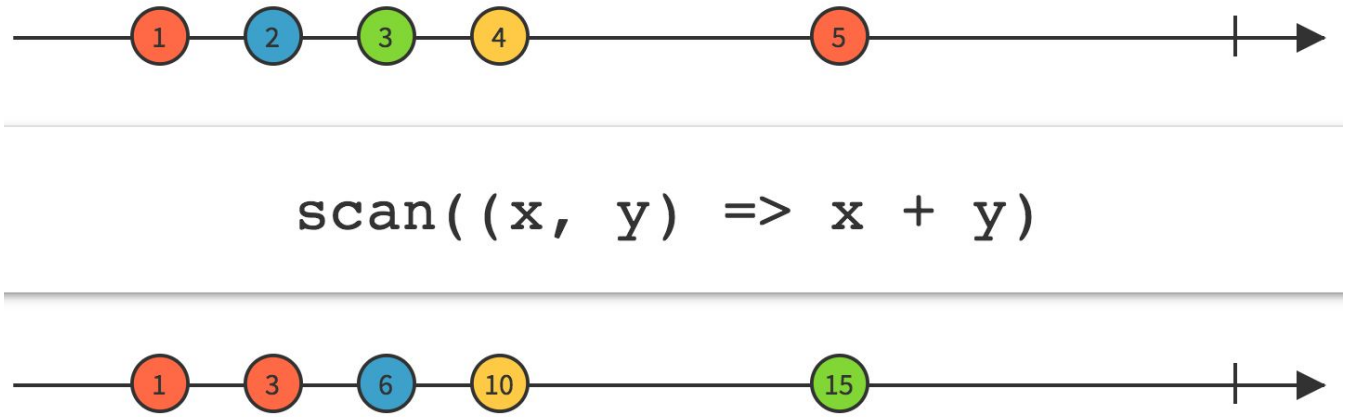
```
Next: 0
Next: 2
Next: 4
Completed
```

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'); });
```

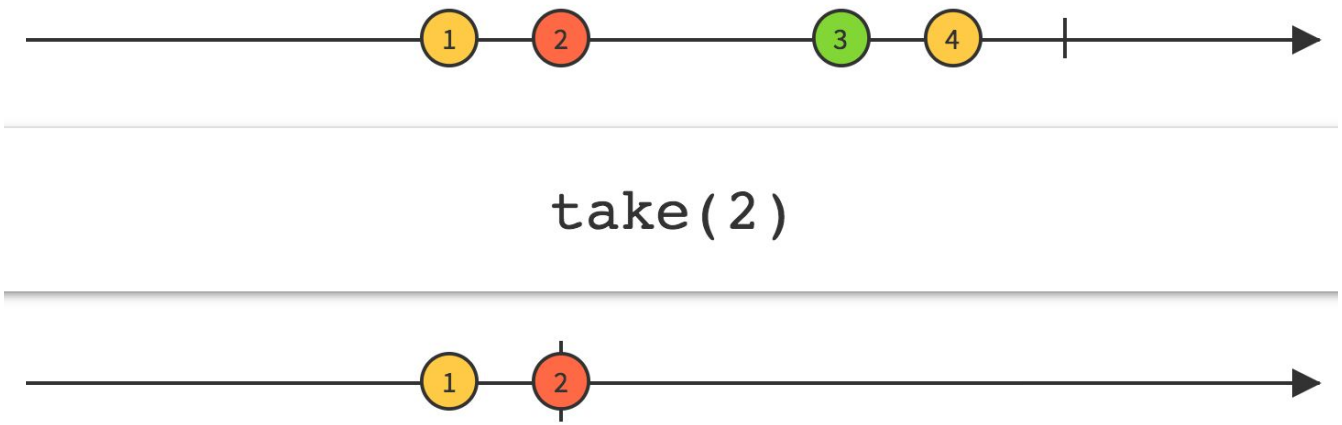
SCAN OPERATOR




```
var source = Observable.range(1, 3)
    .scan(
        function (acc, x) {
            return acc + x;
        });

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

```
Next: 1
Next: 3
Next: 6
Completed
```

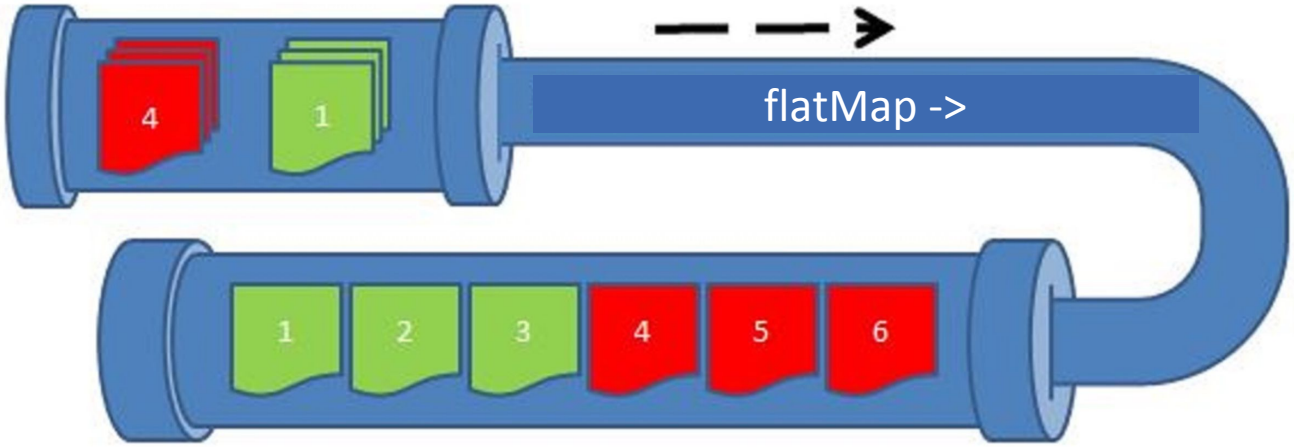
TAKE OPERATOR



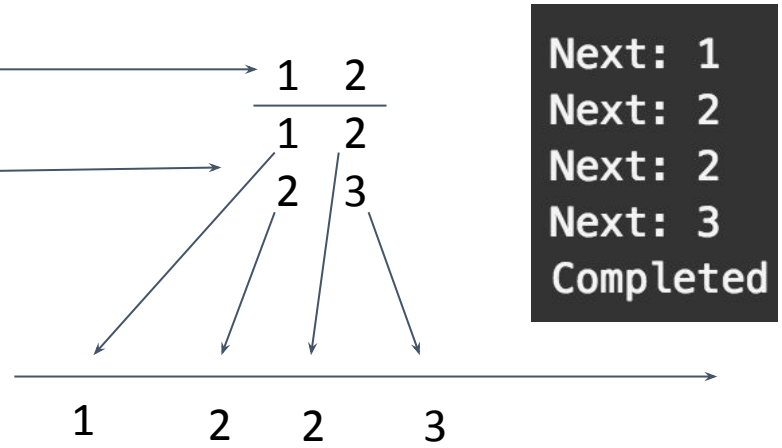

```
var source = Observable.range(0, 5).take(3);

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
```

FLATMAP OPERATOR

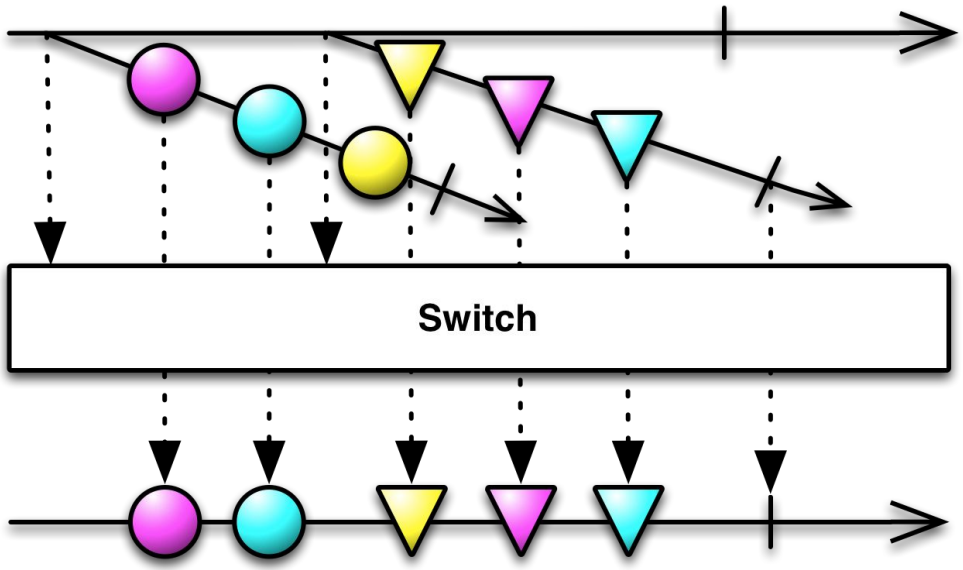



```
var source = Observable
    .range(1, 2)
    .flatMap(function (x) {
        return Observable.range(x, 2);
    });

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



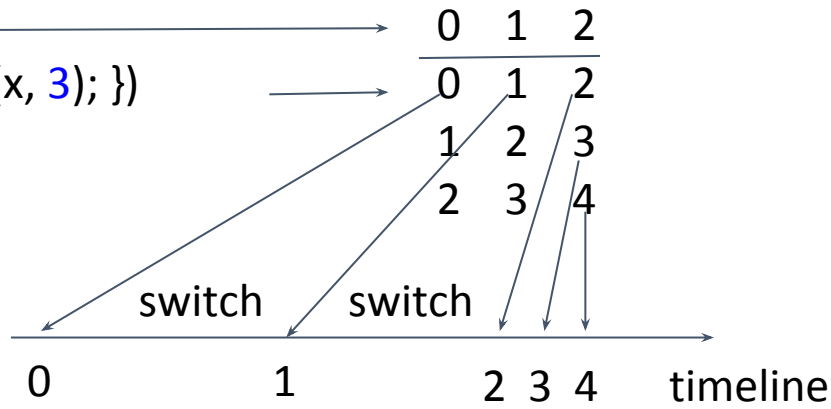

```
Next: 1
Next: 2
Next: 2
Next: 3
Completed
```

SWITCH OPERATOR

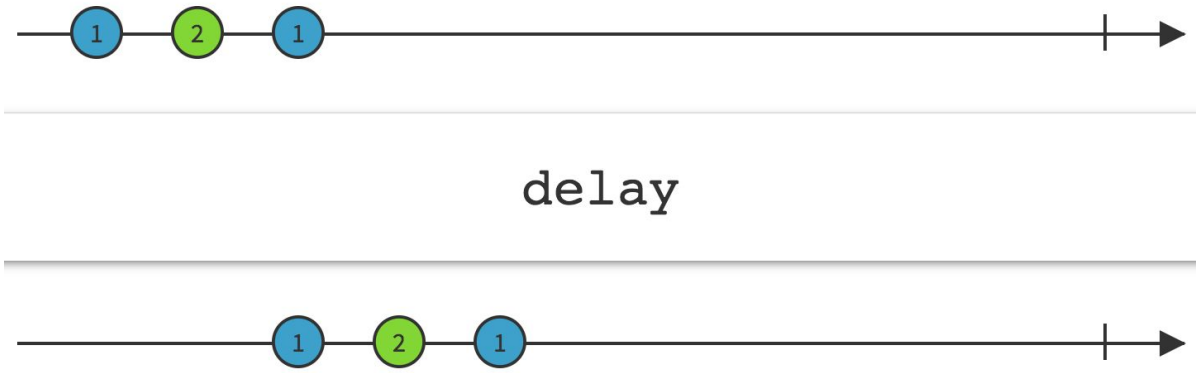



```
Next: 0
Next: 1
Next: 2
Next: 3
Next: 4
Completed
```

```
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'); });
```

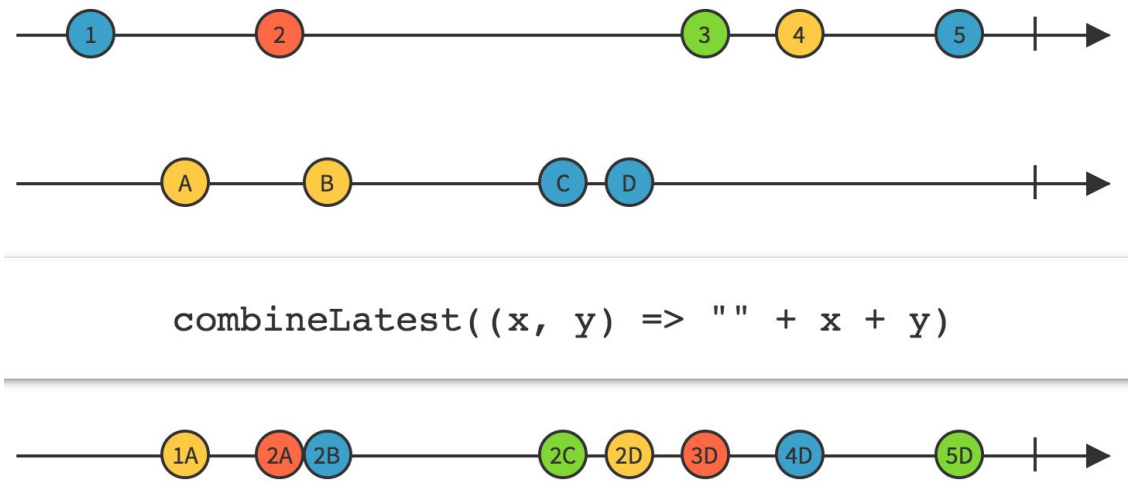


DELAY OPERATOR




```
var source = Observable.range(0, 3)
    .delay(1000);

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

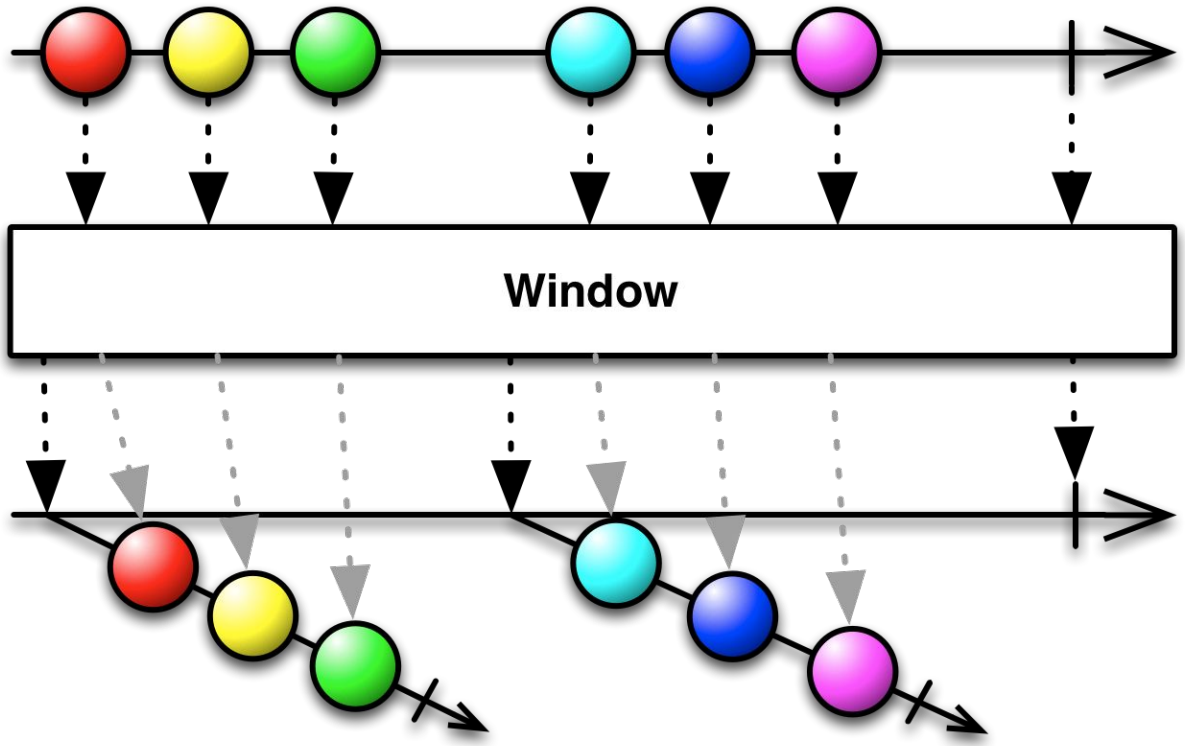
COMBINELATEST OPERATOR




```
/* Have staggering intervals */
var source1 = Observable.interval(100)
    .map(function (i) { return 'First: ' + i; });
var source2 = Observable.interval(150)
    .map(function (i) { return 'Second: ' + i; });
// Combine latest of source1 and source2 whenever either gives a value
var source = source1.combineLatest(source2,
    function (s1, s2) { return s1 + ', ' + s2; }).take(4);
var subscription = source.subscribe(x=>console.log('Next: ' + x.toString()),
    err=>console.log('Error: ' + err), ()=>console.log('Completed'));
```

```
Next: First: 0, Second: 0
Next: First: 1, Second: 0
Next: First: 1, Second: 1
Next: First: 2, Second: 1
Completed
```

WINDOW OPERATOR

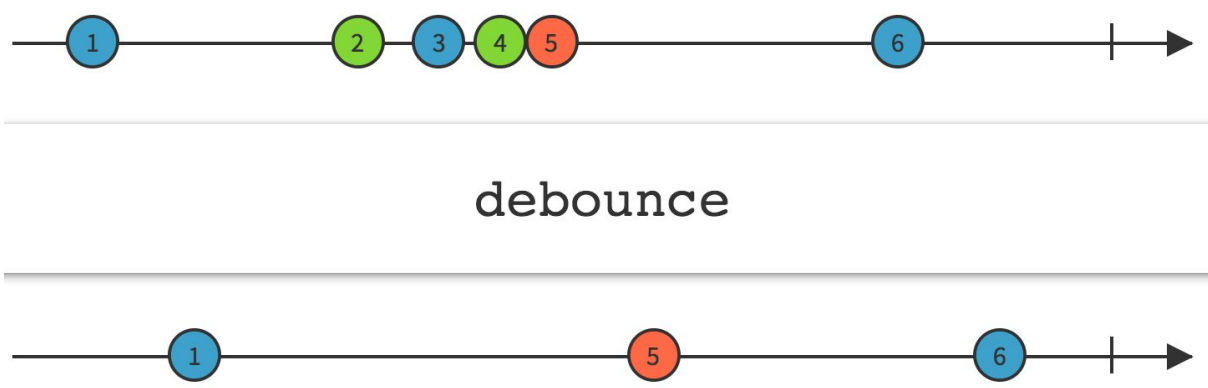


window(windowClosingSelector)
windowWithCount(count)
windowWithTime(timeSpan)
windowWithTimeOrCount(timeSpan,count)
…

DEBOUNCE OPERATOR

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

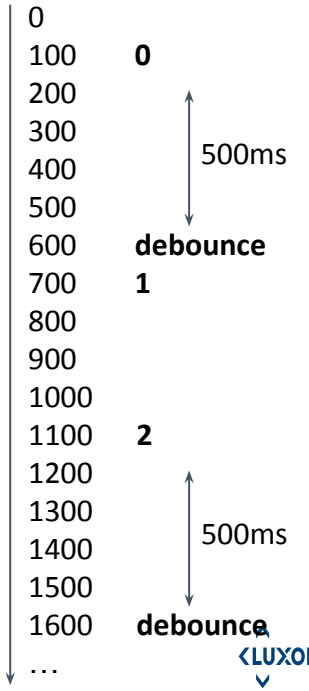
Debouncing enforces that a function not be called again until a certain amount of time has passed without it being called. As in "***execute this function only if 100 milliseconds have passed without it being called***."




```
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));
```



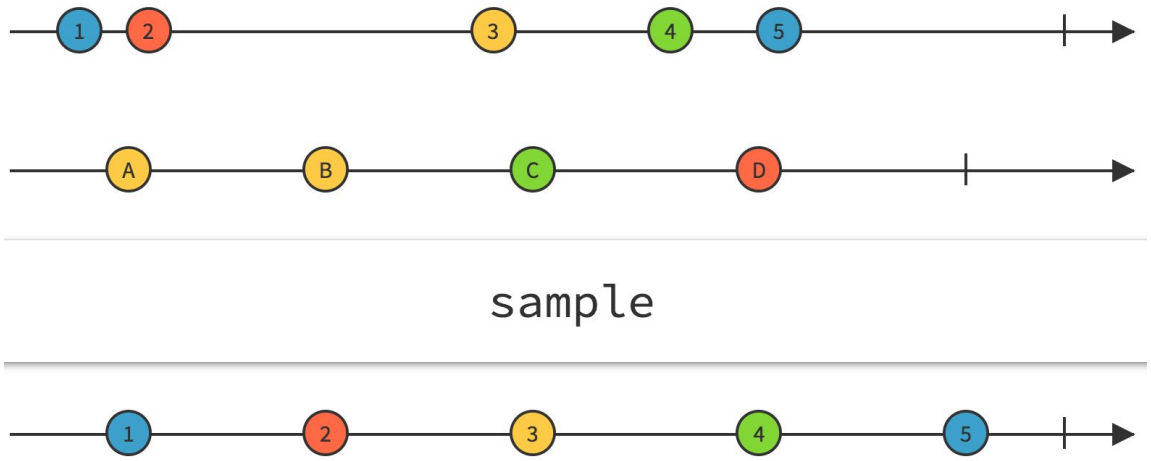
DEBOUNCEWITHSELECTOR OPERATOR


```
debounceWithSelector(x => Rx.Observable.timer(10 * x))
```

SAMPLE OPERATOR (THROTTLELAST)

The main difference between throttle and debounce is that throttle guarantees the execution of the function regularly, at least every X milliseconds.

Throttling enforces a maximum number of times a function can be called over time. As in "***execute this function at most once every 100 milliseconds***."

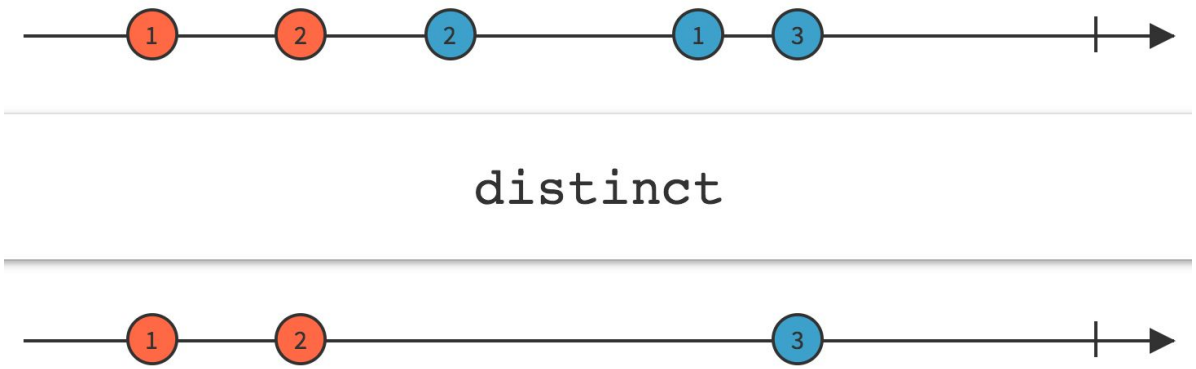


time	value
1s	0
2s	1
3s	2
4s	3
5s	sample: 3
5s	4
6s	5
7s	6
8s	7
9s	8
10s	sample:8

```
var source = Observable.interval(1000)
    .sample(5000)
    .take(2);

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

```
Next: 3
Next: 8
Completed
```

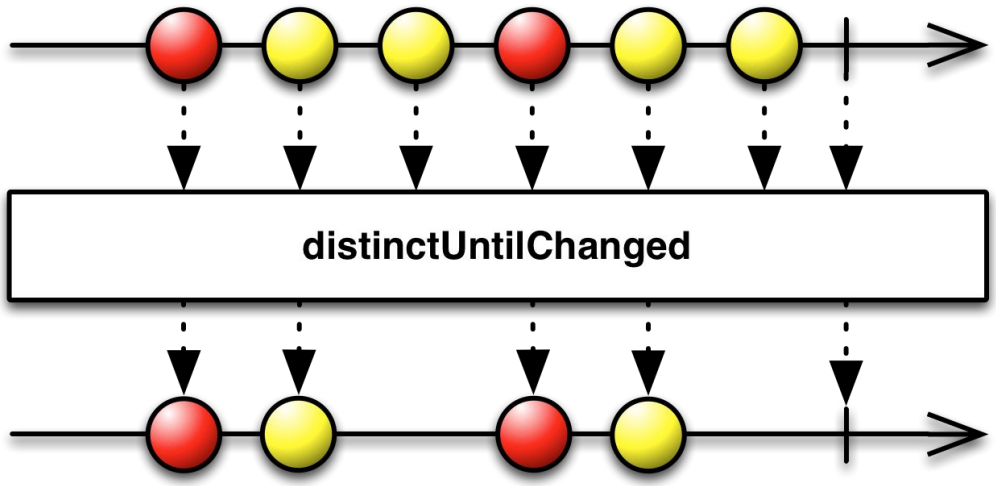
DISTINCT OPERATOR




```
/* Without key selector */
var source = Observable.fromArray([
   42, 24, 42, 24
])
.distinct();

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

```
Next: 42
Next: 24
Completed
```

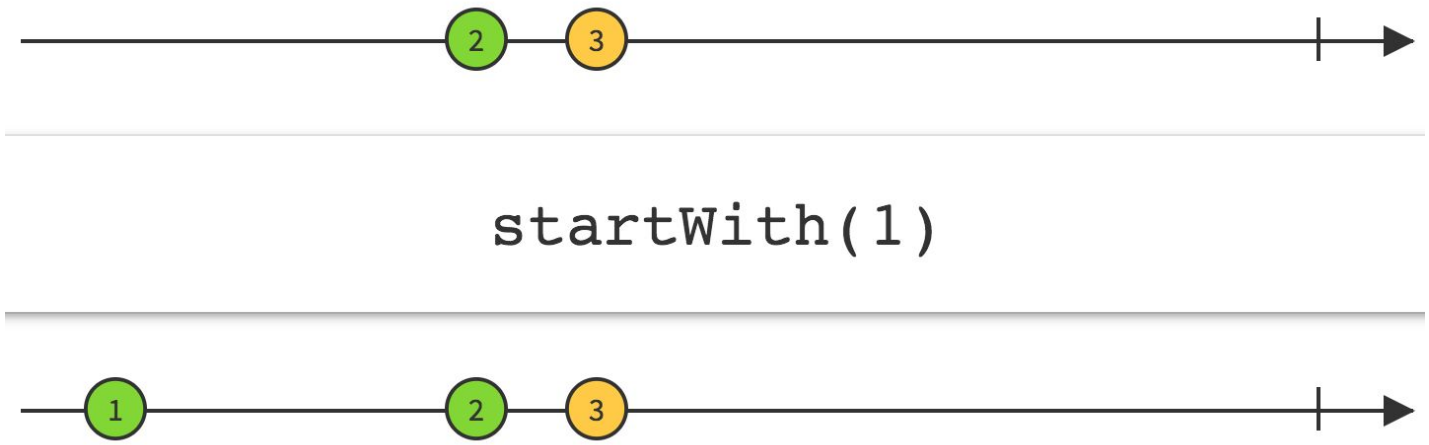
DISTINCTUNTILCHANGED OPERATOR



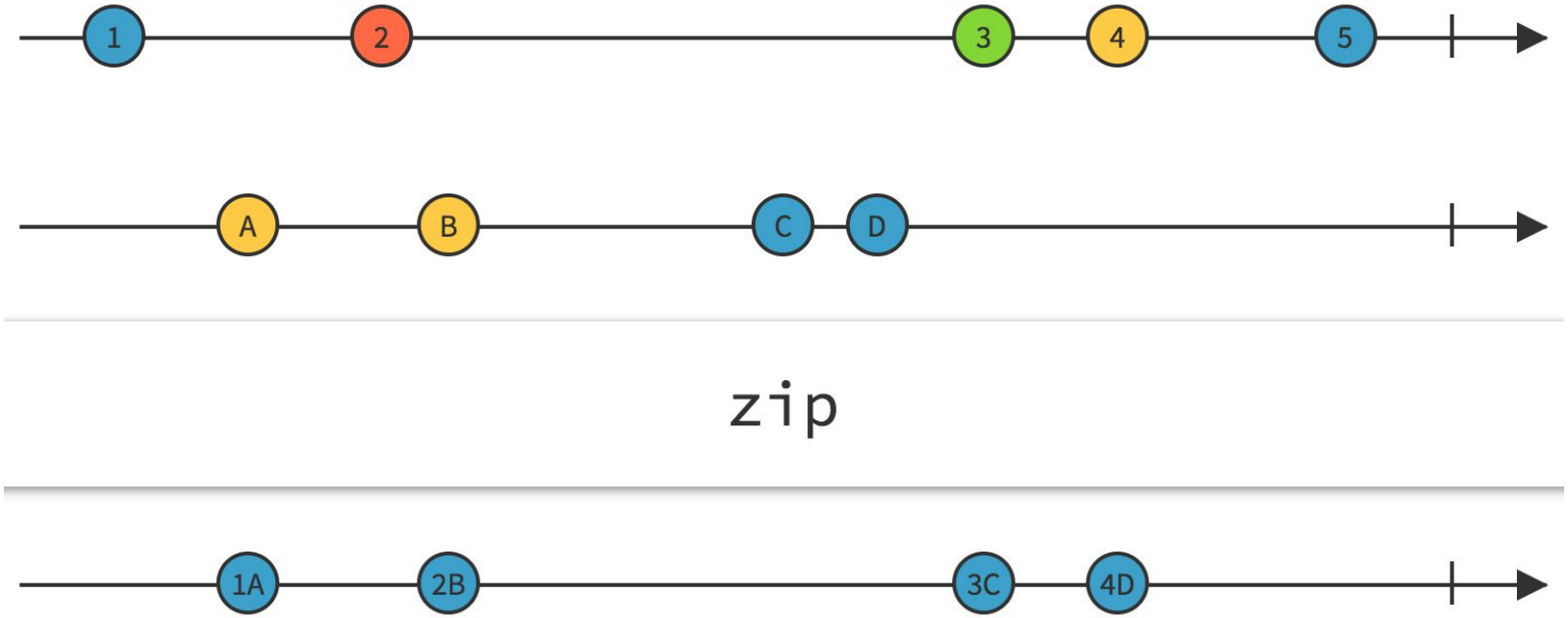

```
var source = Observable.fromArray([
   24, 42, 24, 24
])
.distinctUntilChanged();

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

```
Next: 24
Next: 42
Next: 24
Completed
```

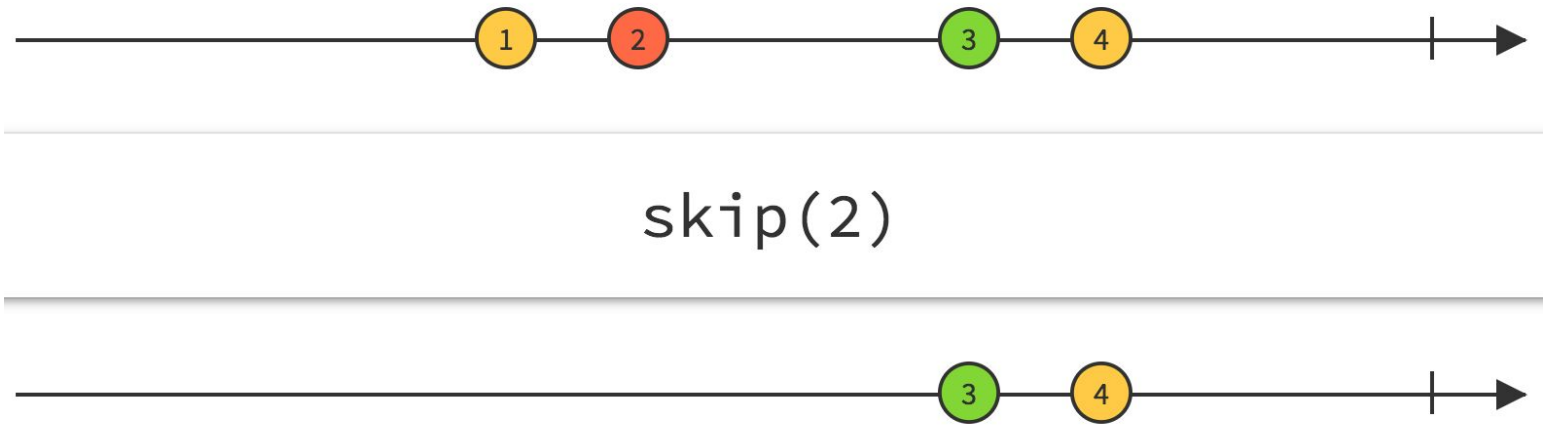
STARTWITH OPERATOR



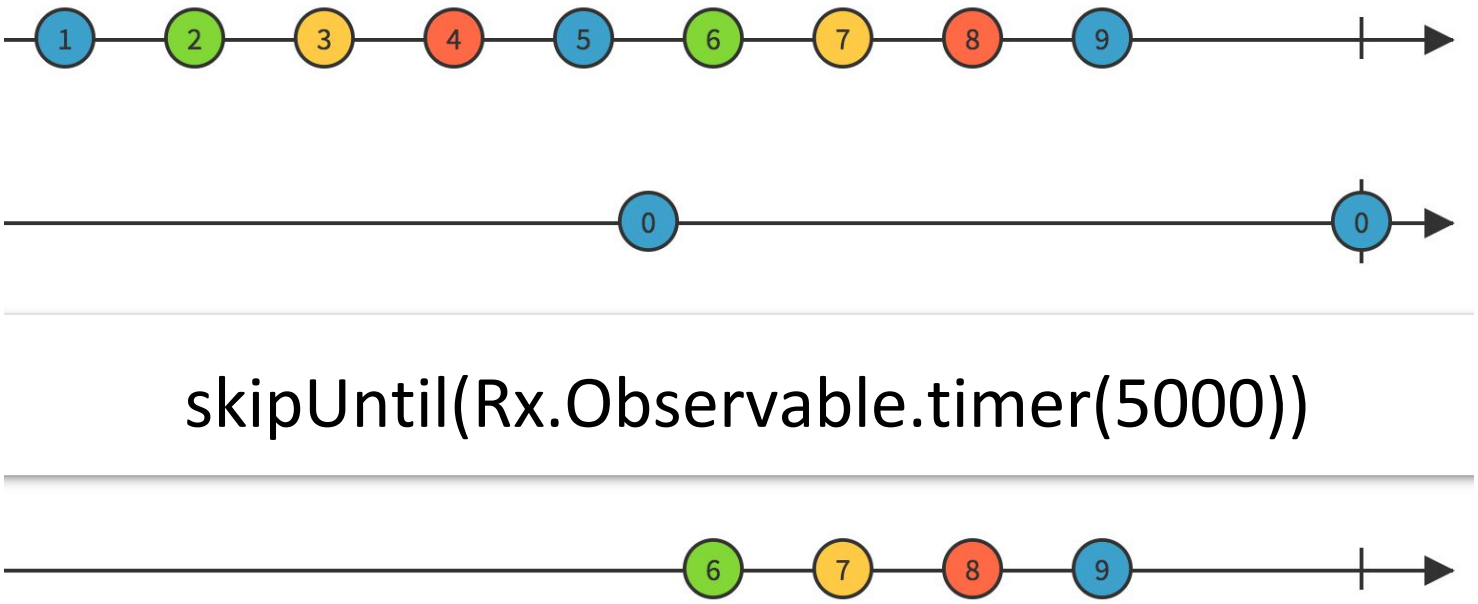
ZIP OPERATOR



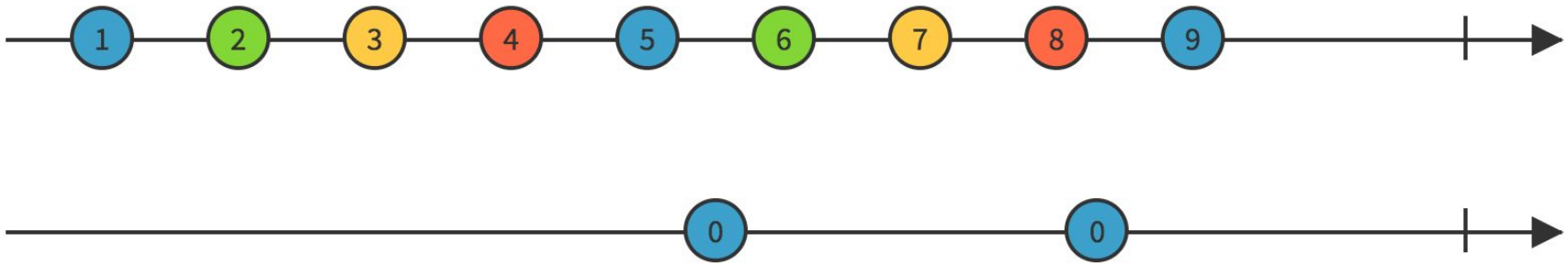
SKIP



SKIPUNTIL

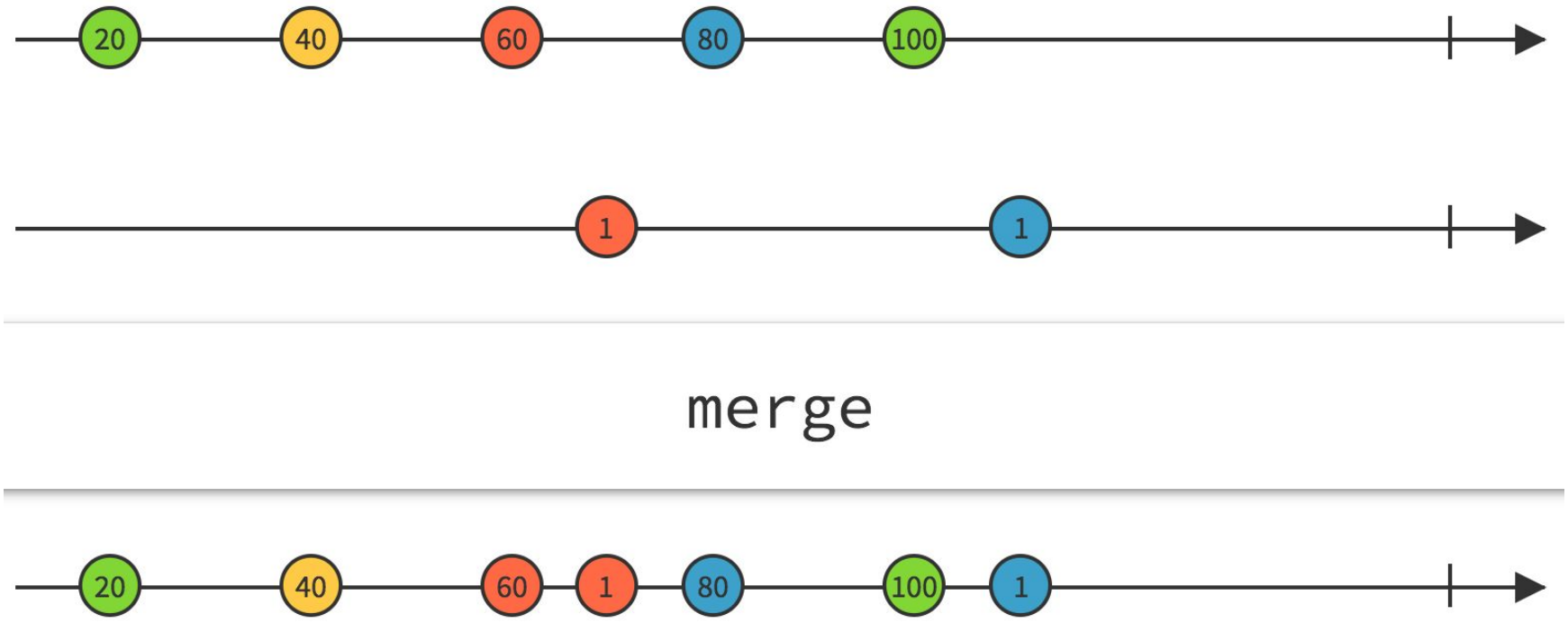


TAKEUNTIL



takeUntil(Rx.Observable.timer(5000))

MERGE OPERATOR



OPERATOR SUMMARY

Simple operators

• map() • filter() • reduce() • first() • last() • single() • elementAt()
• toArray() • isEmpty() • take() • skip() • startWith()
• many more…

Merging and joining operators

• merge • mergeMap (flatMap) • concat • concatMap • switch
• switchMap • combineLatest • withLatestFrom • zip • forkJoin • expand

Splitting and grouping operators

• groupBy • window • partition

Buffering strategy operators

• buffer • throttle • debounce • sample

HOW ANGULAR 2 USE OBSERVABLES?

Angular 2 currently uses RxJs Observables in two different ways:

- as an internal implementation mechanism, to implement some of its core logic like EventEmitter
- as part of its public API, namely in Forms and the HTTP module

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));
   }
}
```

PROCESSING ASYNC DATA WITH RXJS

```
/* Get stock data somehow */
const source = getAsyncStockData();

const subscription = source
    .filter(quote => quote.price > 30)
    .map(quote => quote.price)
    .forEach(price => console.log(`Prices higher than $30: ${price}`);
```

PROCESSING ASYNC DATA WITH RXJS

```
// Listen to keypresses on input
Observable.fromEvent(searchInput, 'keyup')
// Get the value of the input
   .map(event => event.target.value)
   // Only pass through values with 3 or more characters
   .filter(value => value.length > 2)
   // As keypresses does not necessarily change the value of the input
   // make sure we only move on when the value has changed
   .distinctUntilChanged()
   // Only move on with latest keypress with 500 ms interval
   .debounceTime(500)
   // Send search request, retry 3 times on fail and return request. Since we
   // return a request observable we need to use flatMap to extract the actual value
   .flatMap(value => this.http.get(`/api/items?query=${value}`).retry(3))
   // Extract the data of the response
   .map(response => response.data)
   // Render the items
   .forEach(items => renderItems(items))
   // If something fails show error
   .catch(err => renderError(err.message))
```

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