Reactive Programming in JavaScript

University of Parma

by Giuseppe La Gualano - 2019/2020

Reactive Manifesto

- 📱 **Responsive**: The system responds in a timely manner if at all possible.
- 💥 **Resilient**: The system stays responsive in the face of failure.
- 😎 **Elastic**: The system stays responsive under varying workload.
- 📨 **Message Driven**: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

...and more.

What is Asynchronicity in JavaScript?

JavaScript is **synchronous** and **single-threaded**, but we can make it asynchronous.

```
setTimeout(function(){
   console.log('1');
}, 2000);

console.log('2');
```

-> 2 will be written to the console before 1. Because setTimeout is async.

Sequential description of asynchronous operations

An asynchronous operation will terminate sometime in the future.

```
doA(function(aResult) {
    // do some stuff inside b then fire callback
    doB(aResult, function(bResult) {
        // ok b is done...
    });
});
```

Who promises that they will be finished?

by Giuseppe La Gualano

Promise and Future

Similar concepts but not the same thing (except for JavaScript).

- `asynchronous operation` : will terminate sometime in the future.
- `return` : either a value or an error
- `meantime` : we want to “declare” what to do next with the value (or the error)...
- `chaining` : we would like to repeat this over and over again

The Dark Side of Encapsulation

> **Promise** and **Future** encapsulate a value that will eventually be available in the future.

- `Promise` : is the way to generate the value in an asynchronous manner.
- `Future` : The Future is a Promise seen from the "consumer side".

It's what we use to chain operations and react to value when it's available.

```
Promise.resolve("1").then(console.log); // then callbacks are always asynchronous
console.log("2");  // Still 2 before 1
```

`then` does always **schedule** the callback function that you pass in for later.

Problem: Callback hell

```
getData = function(param, callback){
  $.get('http://example.com/get/'+param,
    function(responseText){
      callback(responseText);
    });
} // '$' refers to JQuery library, it's just for example...

getData(0, function(a){
  getData(a, function(b){
    getData(b, function(c){
      getData(c, function(d){
        getData(d, function(e){
          // ... force us into a continuation passing style of execution
        });
      });
    });
  });
});
```

Solution: Invert the chain of responsibility

```
getData = function(param, callback){
  return new Promise(function(resolve, reject) {
    $.get('http://example.com/get/'+param,
    function(responseText){
      resolve(responseText);
    });
  });
}

getData(0).then(getData)
  .then(getData)
  .then(getData)
  .then(getData);
```

Now the caller is responsible for handling the result of the promise when it is resolved.

Async-Await

There’s a special syntax to work with promises in a more comfortable fashion, called “async/await”.

- Simplifies sequential description of asynchronous operations
- `await` : used to wait for value to be ready (just like .then())
- `async` : must be declared for any function that need to use **await**
- Can be used in conjunction with **try-catch** blocks

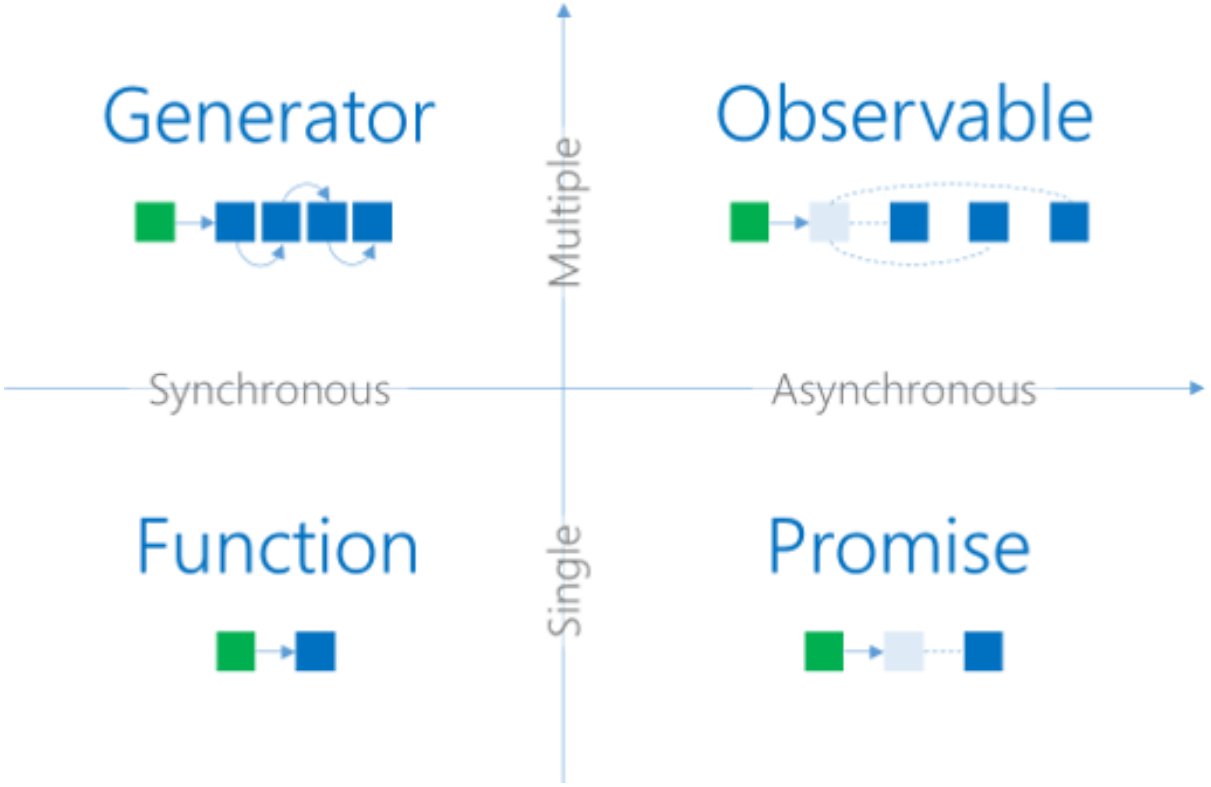
```
async function sequence() {
  await promise1(50);
  await promise2(50); // executes only after promise1 is resolved, like ".then"
  return "done!";
} // Not very good... We have to make all executions parallel!
```

Paradigm comparison

1 Promise -> 1 Event.

How to deal with more events?

	Single	Multiple
Pull	Function	List, Array, Iterator, etc.
Push	Promise	Observable



But first... Classes Constructor

> The **constructor method** is a special method for creating and initializing an object created **within a class**.

Think of it as a **collection of (asynchronous) events**.

```
class Person {
  constructor(name) {
    this.name = name;
  }

  showPerson() {
    console.log('Hello, my name is ${this.name}');
  }
}

const Giuseppe = new Person('Giuseppe');
Giuseppe.showPerson();
```

Observable

> A design pattern used to inform some observers about “changes” occurred to a subject.

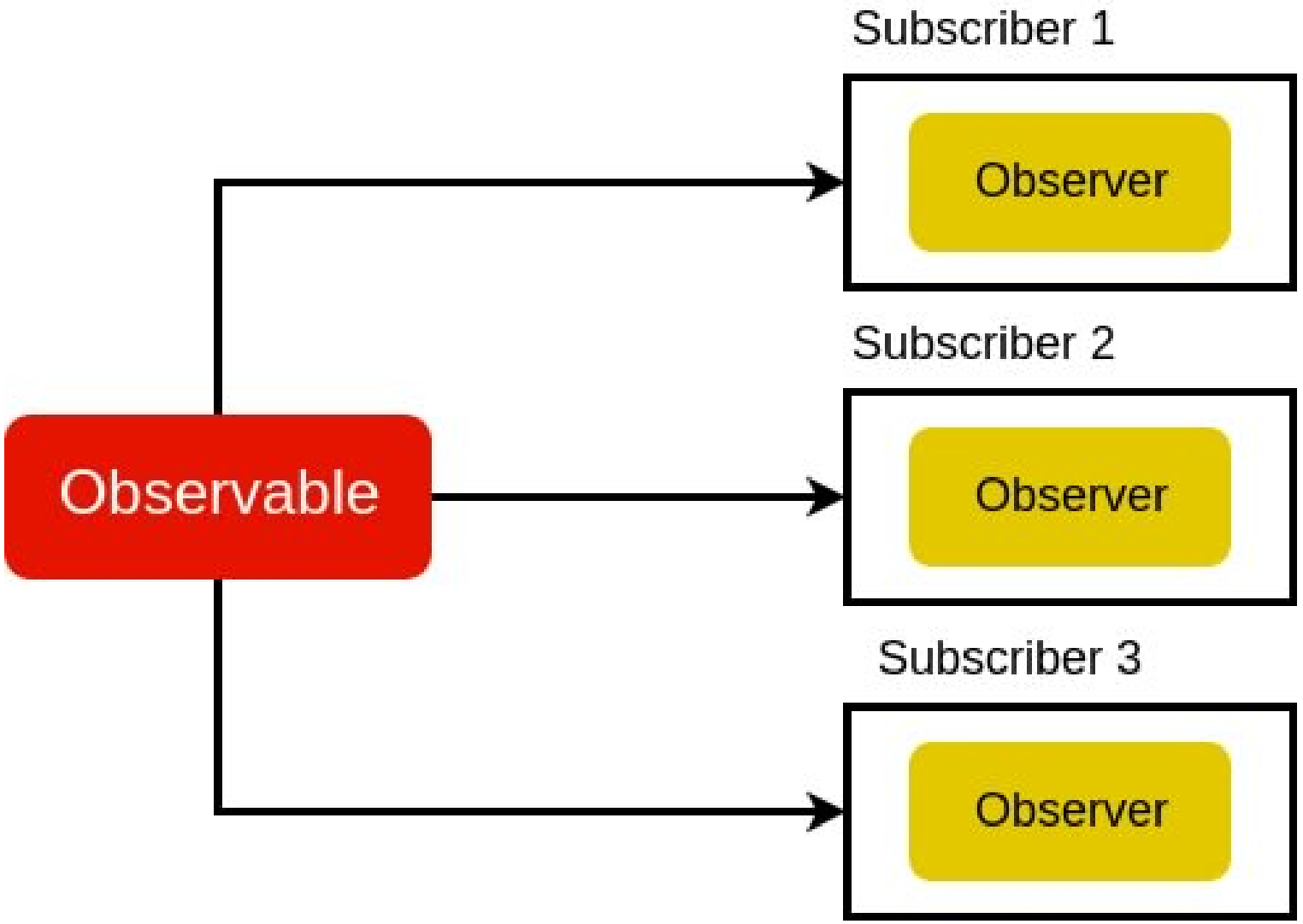
Think of it as a **collection of (asynchronous) events**.

Observables are `lazy` (or cold), no execution until someone subscribes, either synchronously or asynchronously.

```
class Observable {
    constructor(functionThatTakesObserver){
      this._functionThatTakesObserver = functionThatTakesObserver;
    }
}

subscribe(observer) {
  return this._functionThatTakesObserver(observer);
}
```

How it works


Subscriber 1
Observer
Subscriber 2
Observable
Observer
Subscriber 3
Observer

JavaScript's Lambda and Arrow Functions

In JavaScript **pre-ES6** we have function expressions which give us an anonymous function (a function without a name).

```
var anon = function (a, b) { return a + b };
```

In **ES6** we have **arrow functions** with a more flexible syntax that has some bonus features and gotchas.

```
var anon = (a, b) => a + b;
// or
var anon = (a, b) => { return a + b };
```

One of the major advantages of arrow functions is that it does **not have its own value**. It's this is lexically bound to the enclosing scope.

by Giuseppe La Gualano

Subject

Observable is for the consumer, it can be transformed and subscribed:

```
observable.map(x => ...).filter(x => ...).subscribe(x => ...)
```

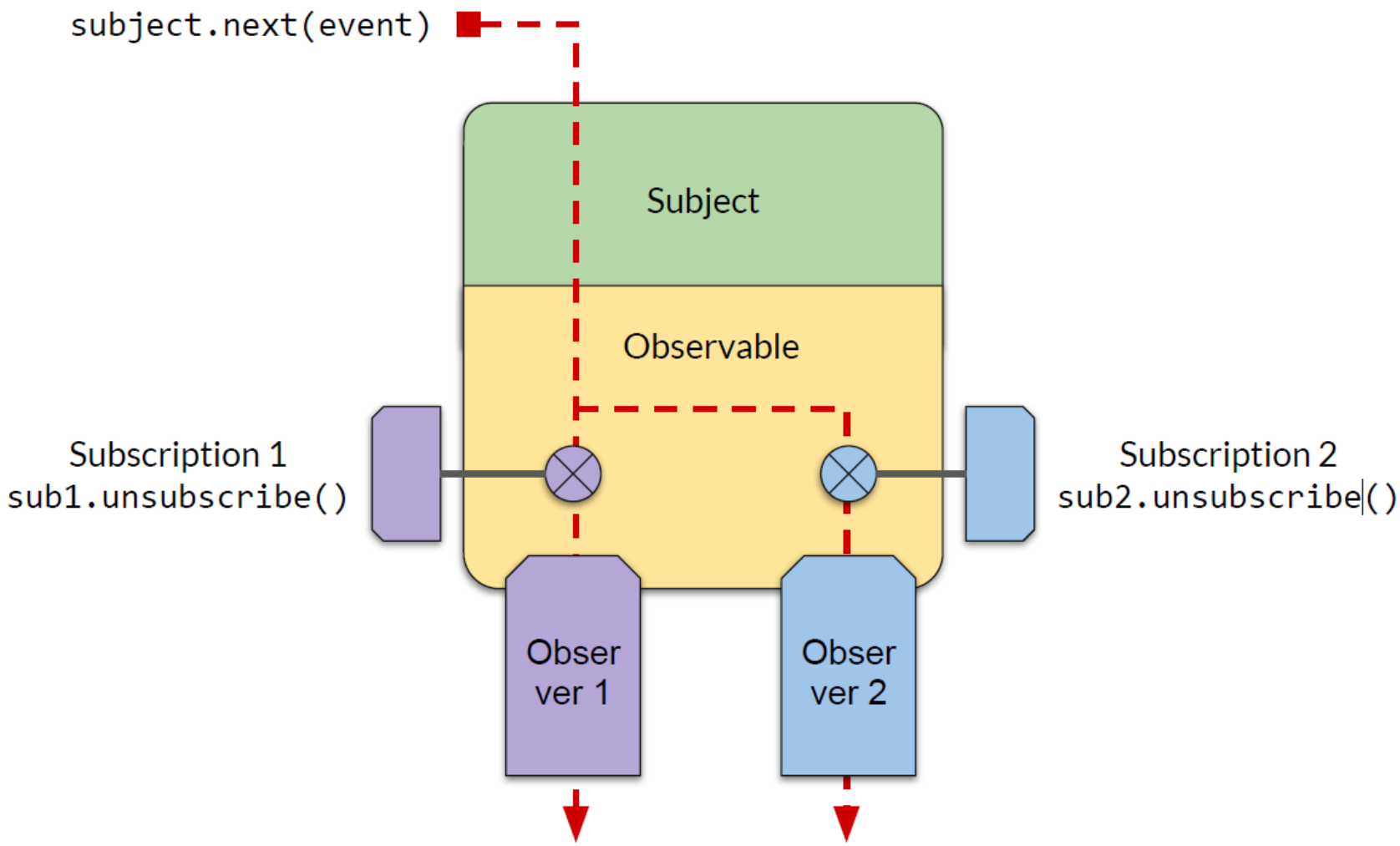
Observer is the interface which is used to feed an *observable source*:

```
observer.next(newItem)
```

We can use a **Subject** which implements both the Observable and the Observer interfaces:

```
var source = new Subject();
source.map(x => ...).filter(x => ...).subscribe(x => ...)
source.next('first')
source.next('second')
```

Subscribe/Unsubscribe



Img courtesy Fabio Strozzi

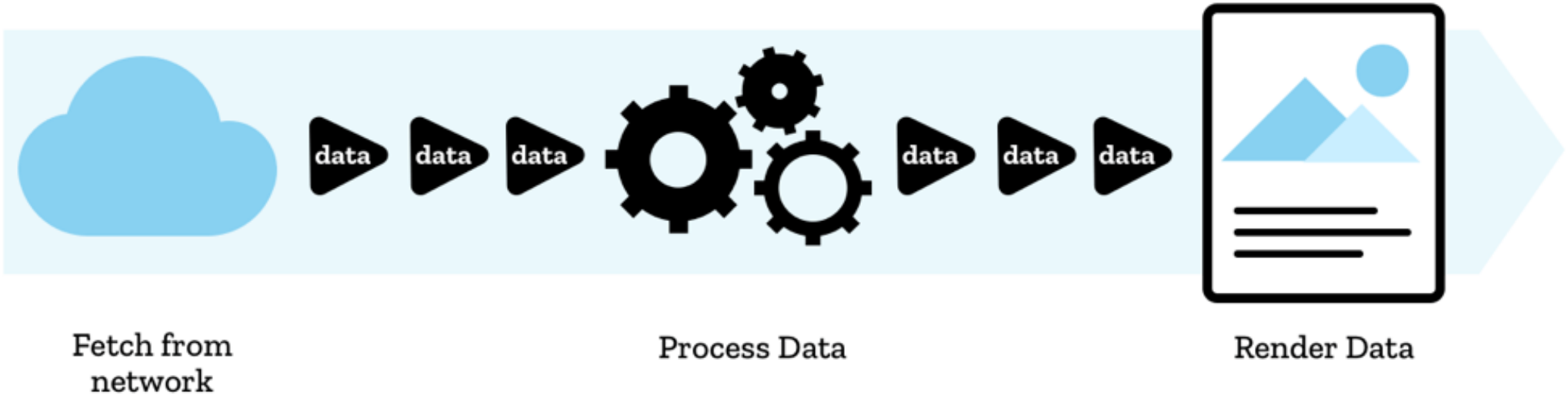
Methods and Notable functional operators

You can extend it with functional programming methods (**map, reduce, etc.**).

- `merge` : merges Observables one another.
- `mergeAll` : like flatten.
- `switchMap` : like flatMap.
- `buffer` : groups consecutive events into arrays.
- `window` : just like buffer but emits Observables instead of arrays.
- `debounce` : delays emission but drops previous pending events.
- `throttle` : like debounce but emits recent events at a fixed rate.
- `timeout` : errors if Observable does not emit a value in given time span.

Streams API

You can create **data streams** of anything: variables, user input, properties, caches, structures. you can then **observe** what is happening and **react** consistently.



Q & A

Reactive Programming in JavaScript

by Giuseppe La Gualano