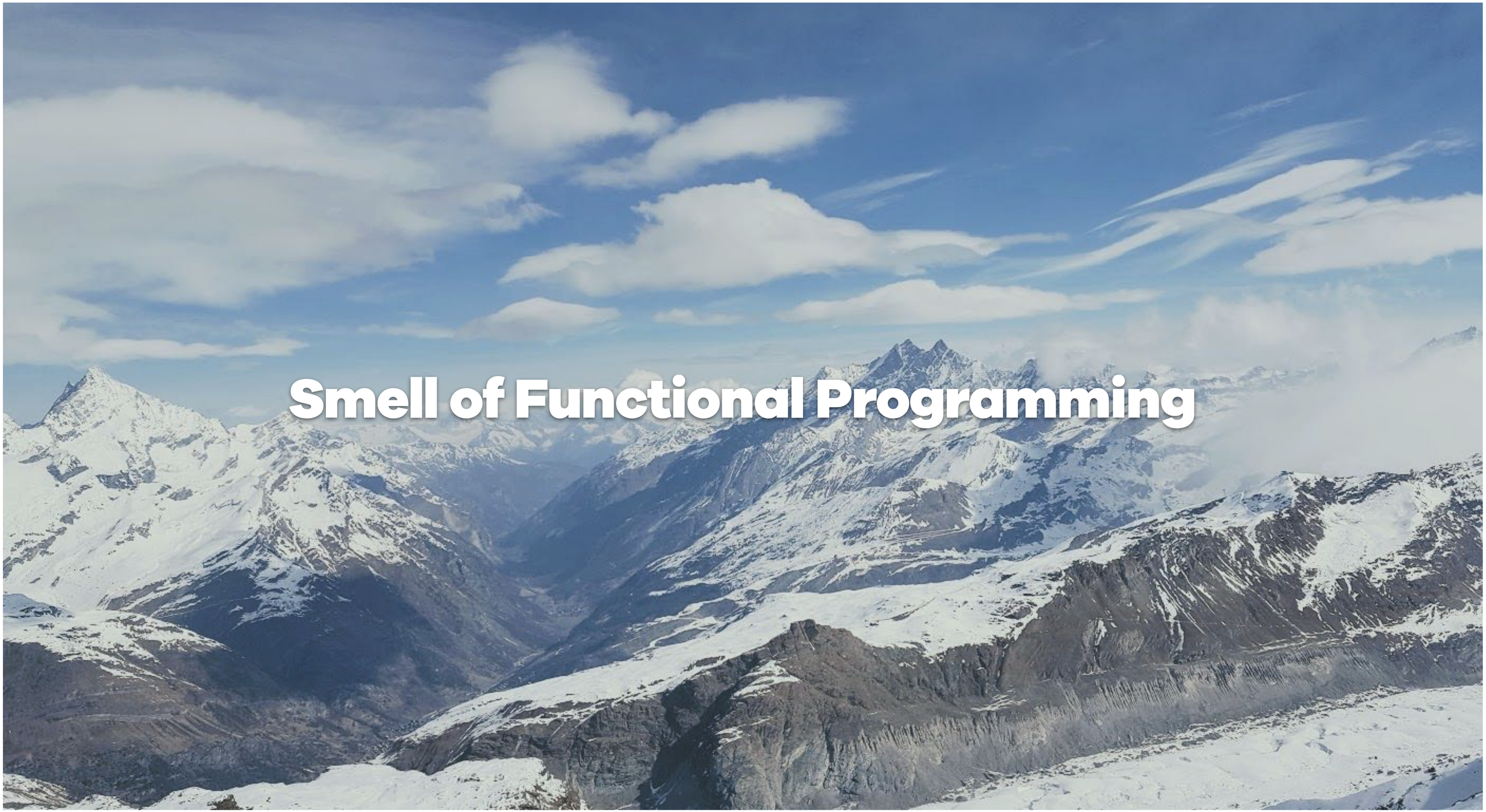
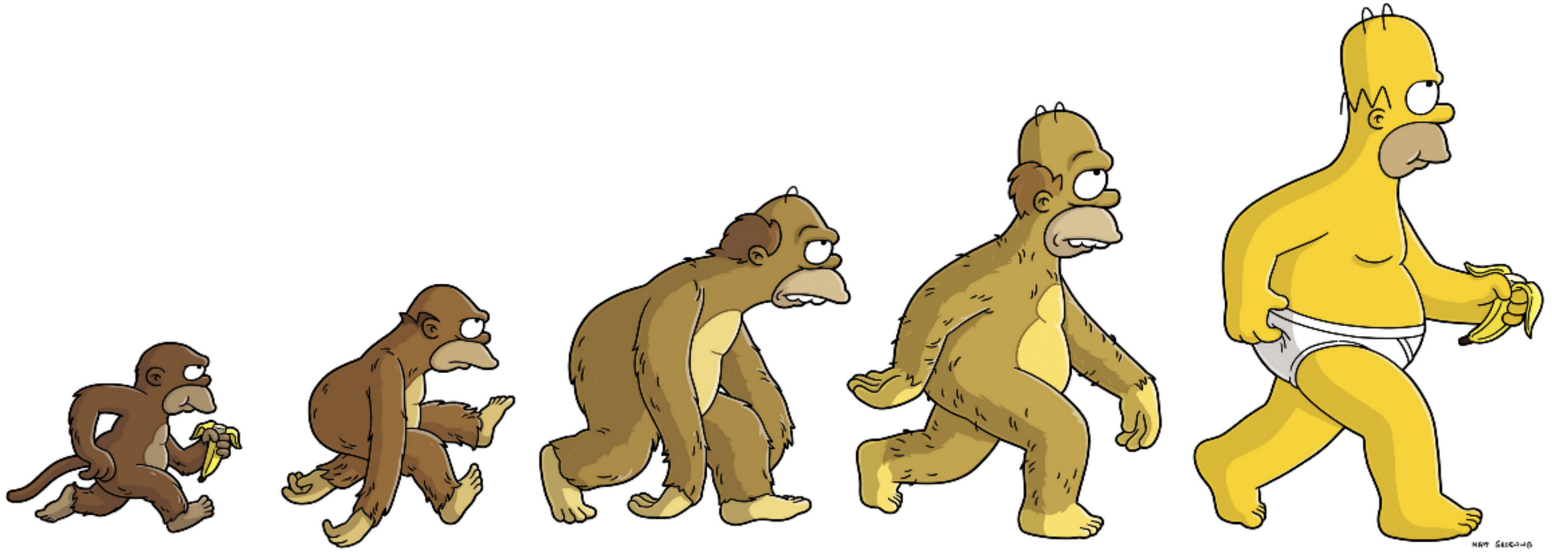

Smell of Functional Programming


MACHINE
ASSEMBLY
PROCEDURAL
OBJECT ORIENTED
FUNCTIONAL


A WORLD FULL OF OBJECTS


NOV 28 2010

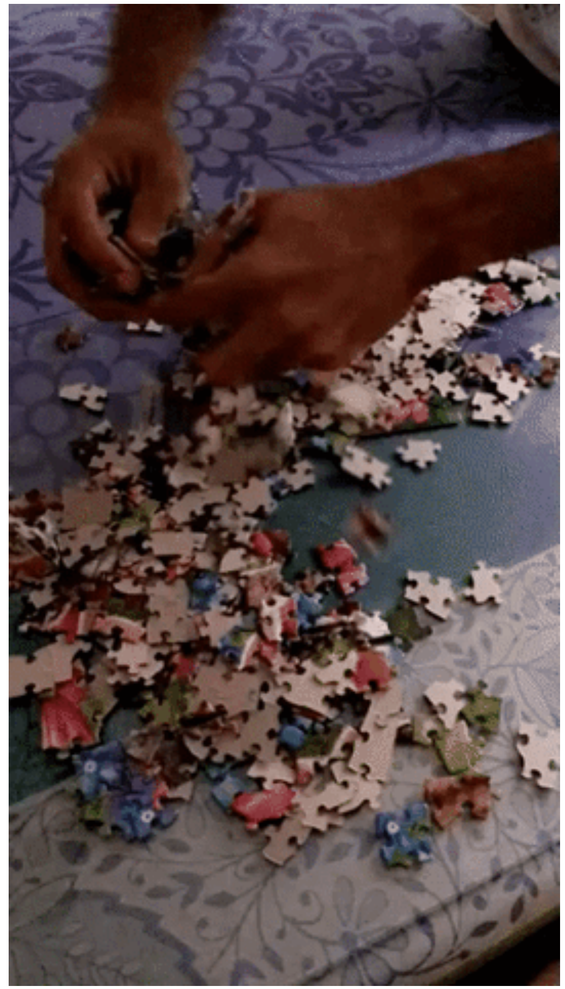
Functional Programming “The Meaning”

Functional Programming is about building **small blocks and combining them together.**

Imperative vs declarative programming

It helps us writing code that is:

- Easier to understand and debug
- Cheaper and easier to maintain
- More legible
- More reusable
- More testable
- Less error-prone



Main Principles

- **Higher order functions**
- **Purity** & **Immutable** Objects
- **Lazy Evaluation & Recursion**
- **Reactive Streams RX**


Higher Order Functions

Higher order functions

A higher order function does at least one of the following:

- **takes one or more functions as arguments**
- **returns a function as its result**

Higher order functions

```
const array = [1, 2, 3]

array.map(number => number + 1) // [2, 3, 4]
```

Higher order functions

Partial application (curry)

Iterators (values, entries, reduce, map, filter)

Function Composition (compose)

Higher order functions

Let’s check an example

Higher order functions - bad example

```
import developers from '../data/devs.json'

function getDevsFromZurich() {
  var developersList = []

  for (var i = 0; i < developers.length; i++) {
    var developer = developers[i]

    if (developer.city === 'Zuerich') {
      developersList.push(developer)
    }
  }

  return developersList
}
```

Higher order functions - going functional

```
import developers from '../data/devs.json'

const isFromZuerich = ({city}) => city === 'Zuerich'
const zuerichDevs = developers.filter(isFromZuerich)
```


Purity & Immutable Objects

Purity & Immutable Objects

A function given the same argument value(s) **always evaluates the same result value**

Evaluation of the result **does not cause any semantically observable side effect**

Purity & Immutable Objects

```
1 const add = (a, b) => a + b
2
3 add(1, 2) // 3
```

Is this function pure?

Purity & Immutable Objects

```
1 const add = (a, b) => alert('Grüezi') || a + b
2
3 add(1, 2) // window.alert and maybe 3
```

Purity & Immutable Objects - Void Methods

```
element.addEventListener('mousedown', () => {
  isMouseDown = true
}, false)

element.addEventListener('mouseup', () => {
  isMouseDown = false
}, false)
```

Purity & Immutable Objects - Void Methods

```
// vue.js
export default {
  methods: {
    onClick() {
       this.wasClicked = true
    }
  }
}
```

Purity & Immutable Objects - Void Methods

```
// React.js
class MyComponent extends React.Component {
    onClicked = () => {
        this.setState({
          wasClicked: true
        })
    }
}
```

INTO THE VOID

Purity & Immutable Objects

Let’s check an example


Lazy Evaluation & Recursion

Lazy Evaluation

evaluate something **only when you actually need it**

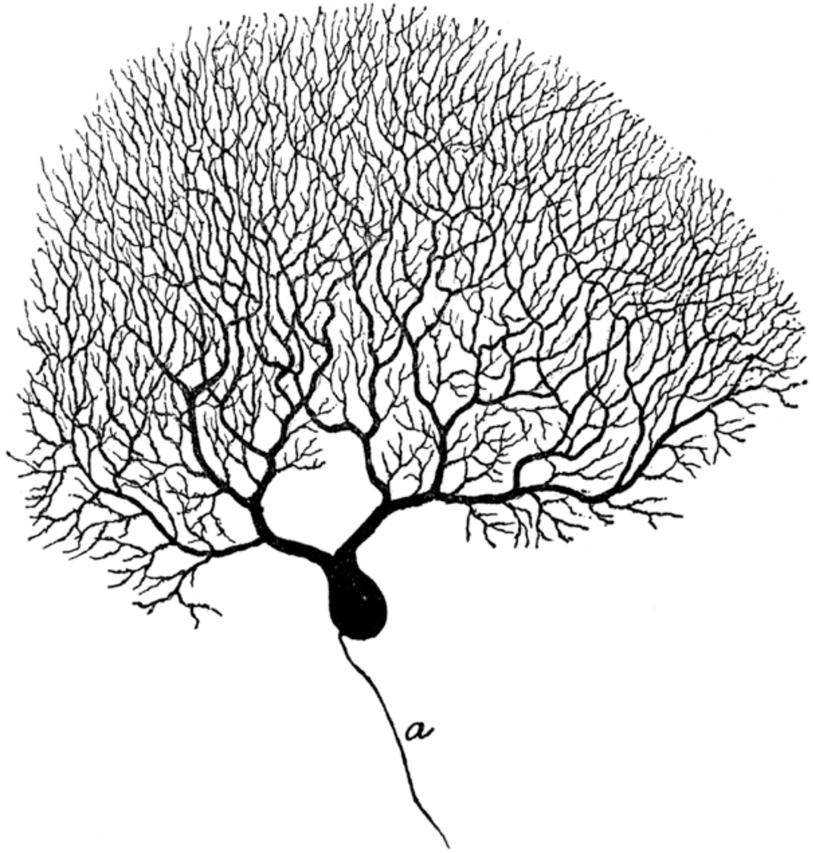
Recursion

A function that calls a function

Lazy Evaluation - Infinite Data Structures

```haskell
ones = 1 : ones
take 5 ones
-- [1, 1, 1, 1, 1]
```

Lazy Evaluation - Infinite Data Structures



Lazy Evaluation

Let’s check an example

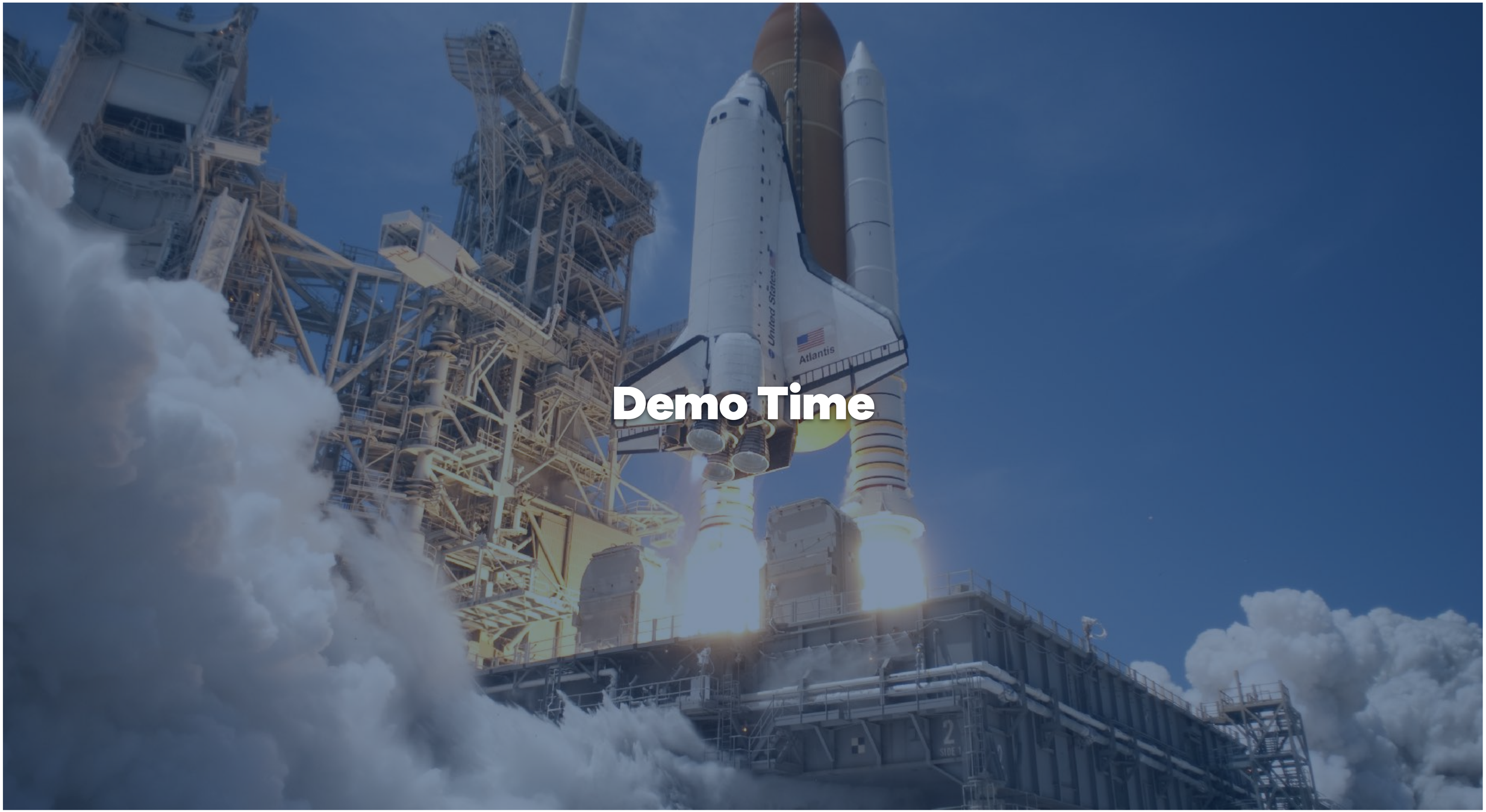

Reactive Streams RX

Reactive Streams RX

composable asynchronous **callbacks**

Lazy Evaluation - UI Streams

Time



Demo Time

Where can I start?

- Never use `*let*` or `*var*`, prefer **always `*const*`** instead
- Use always **functions with`*return*` statements**
- **Avoid** `*for in*` `*while*` loops in favour of `*map, filter, reduce...*`
- **Avoid** "the magic" `*get*` ,`*set*` *or* `*Proxy*`
- **Reduce the use of Classes** in favour of static methods
- Try solving some problems with **recursion** and **generators**
- Write **semantic code** and split it in **smaller functions in separate files**

Vielen Dank

Code

github.com/GianlucaGuarini/smell-of-funtional-programming

Final Demo

bit.ly/2NKwbEE

Gianluca Guarini

@gianlucaguarini