3 things you must know to think reactive

JUG.ch - August 2015

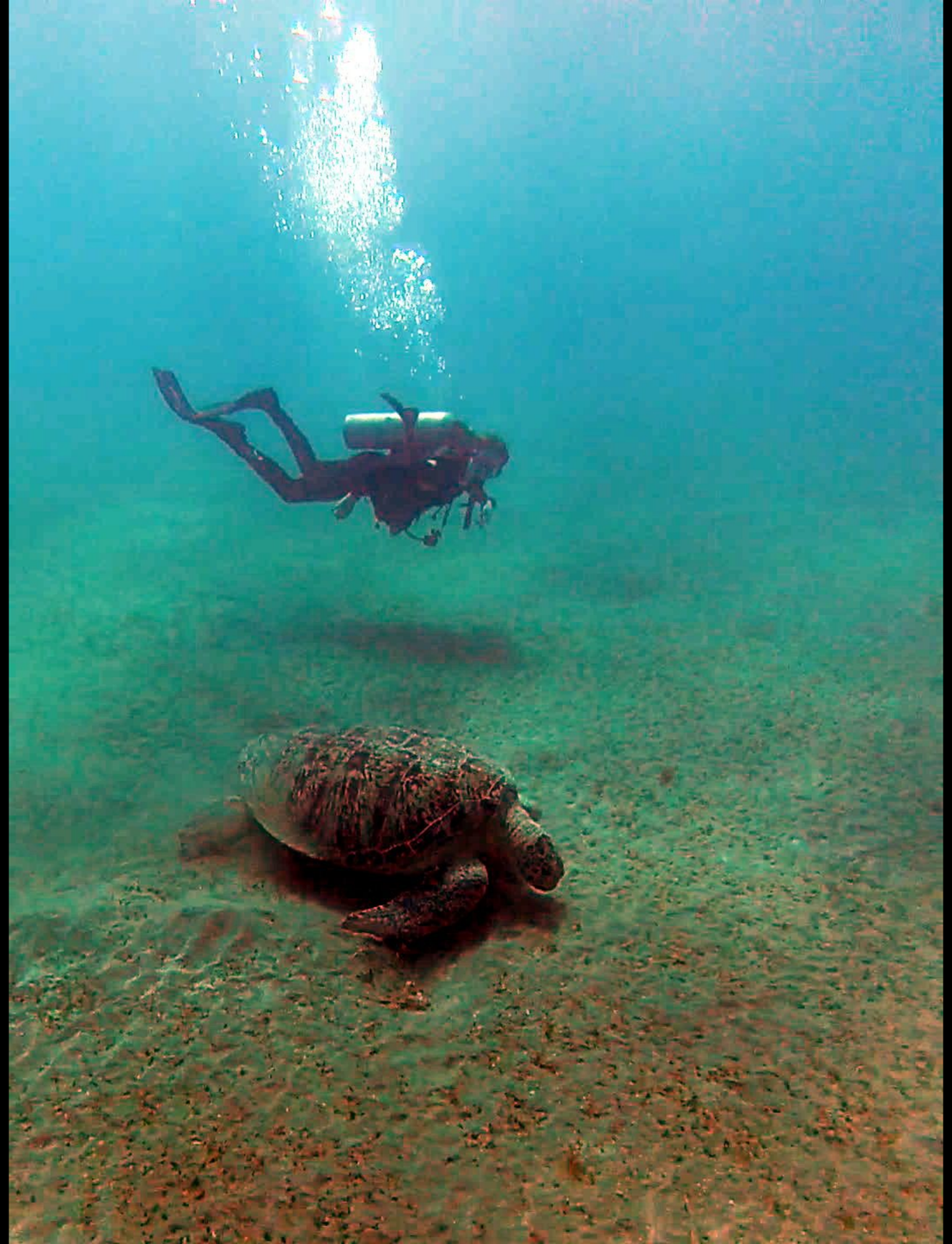
Manuel Bernhardt - @elmanu

Agenda

1. Reactive?
2. Mutability & Immutability
3. Functions & Higher-order functions
4. Why functions?
5. Functional for Reactive

Who is speaking?

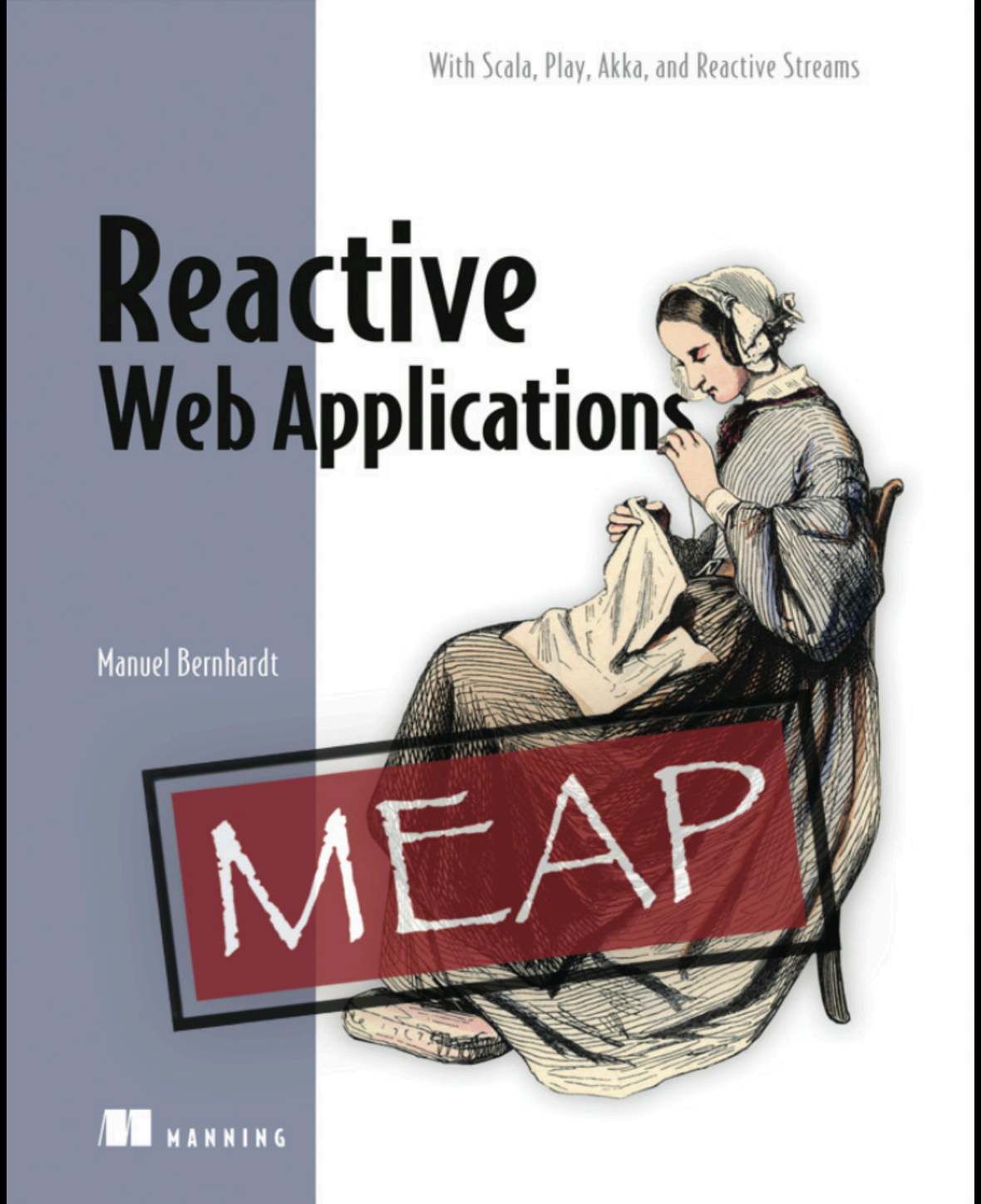
- freelance software consultant based in Vienna
- Vienna Scala User Group
- web, web, web

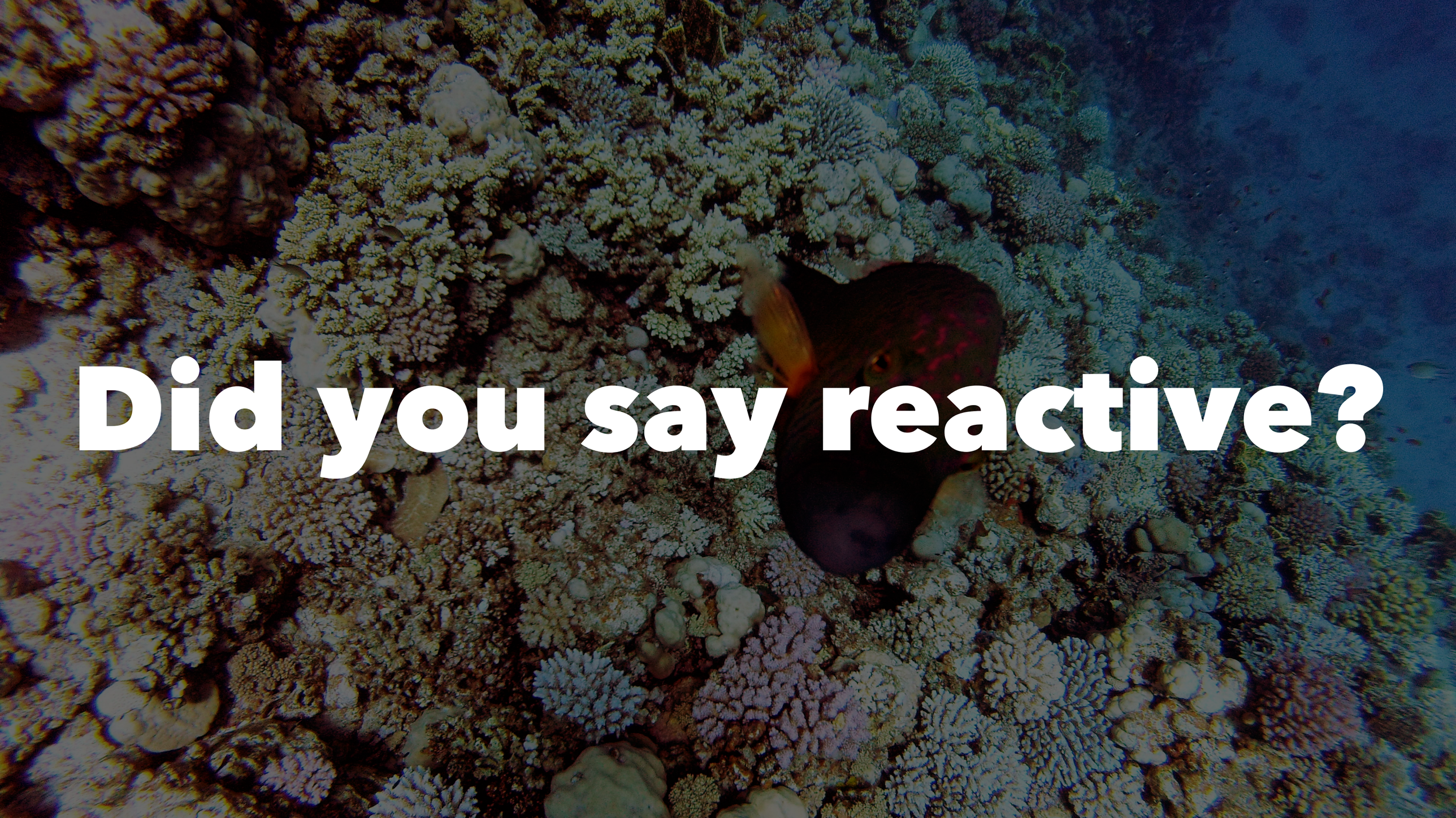


Who is speaking?

- freelance software consultant based in Vienna
- Vienna Scala User Group
- web, web, web
- writing a book on **reactive** web-applications

http://www.manning.com/bernhardt



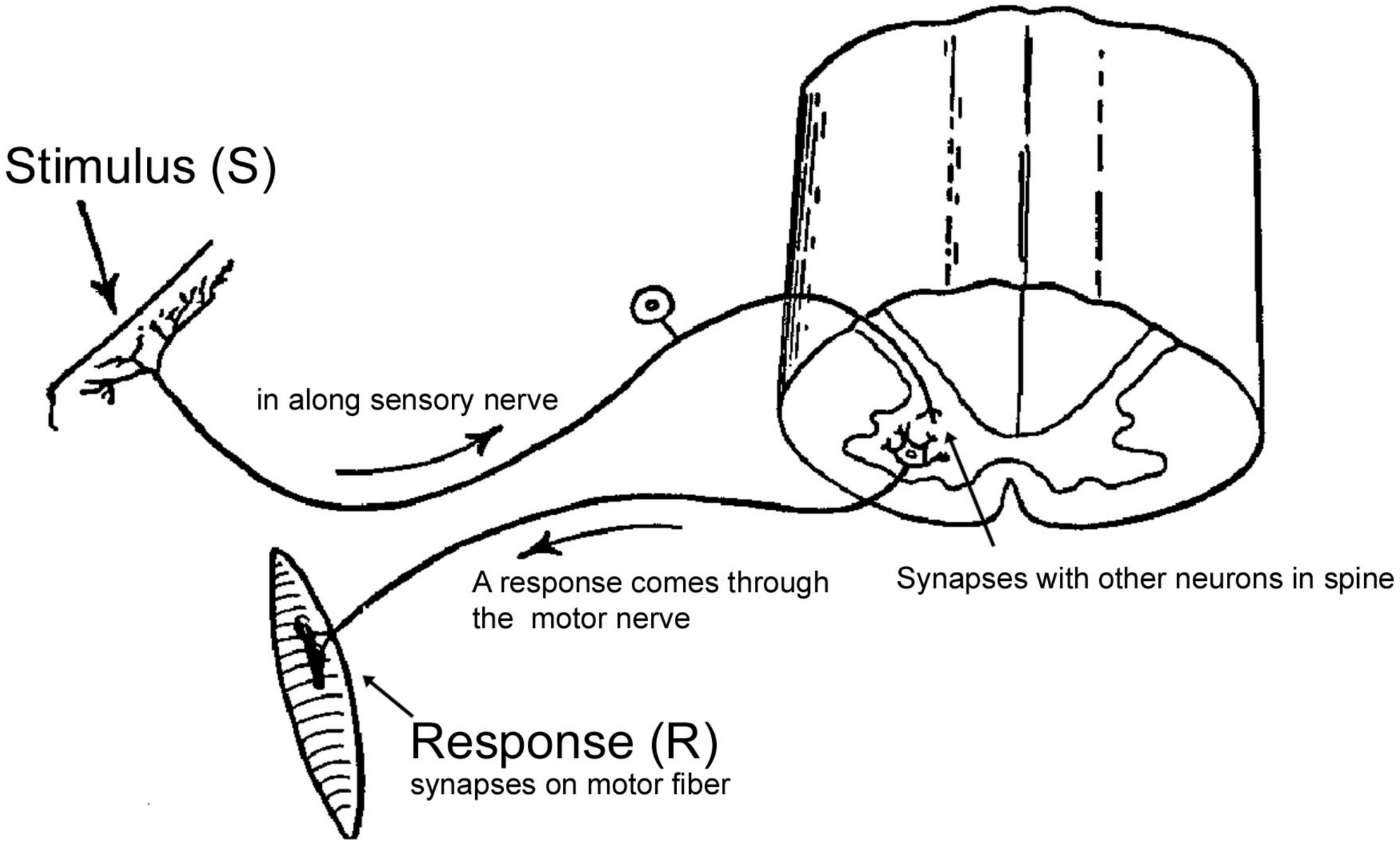

Did you say reactive?

Disambiguation

- Reactive Programming
- Functional Reactive Programming
- Reactive Application
- Responsive Web-Application

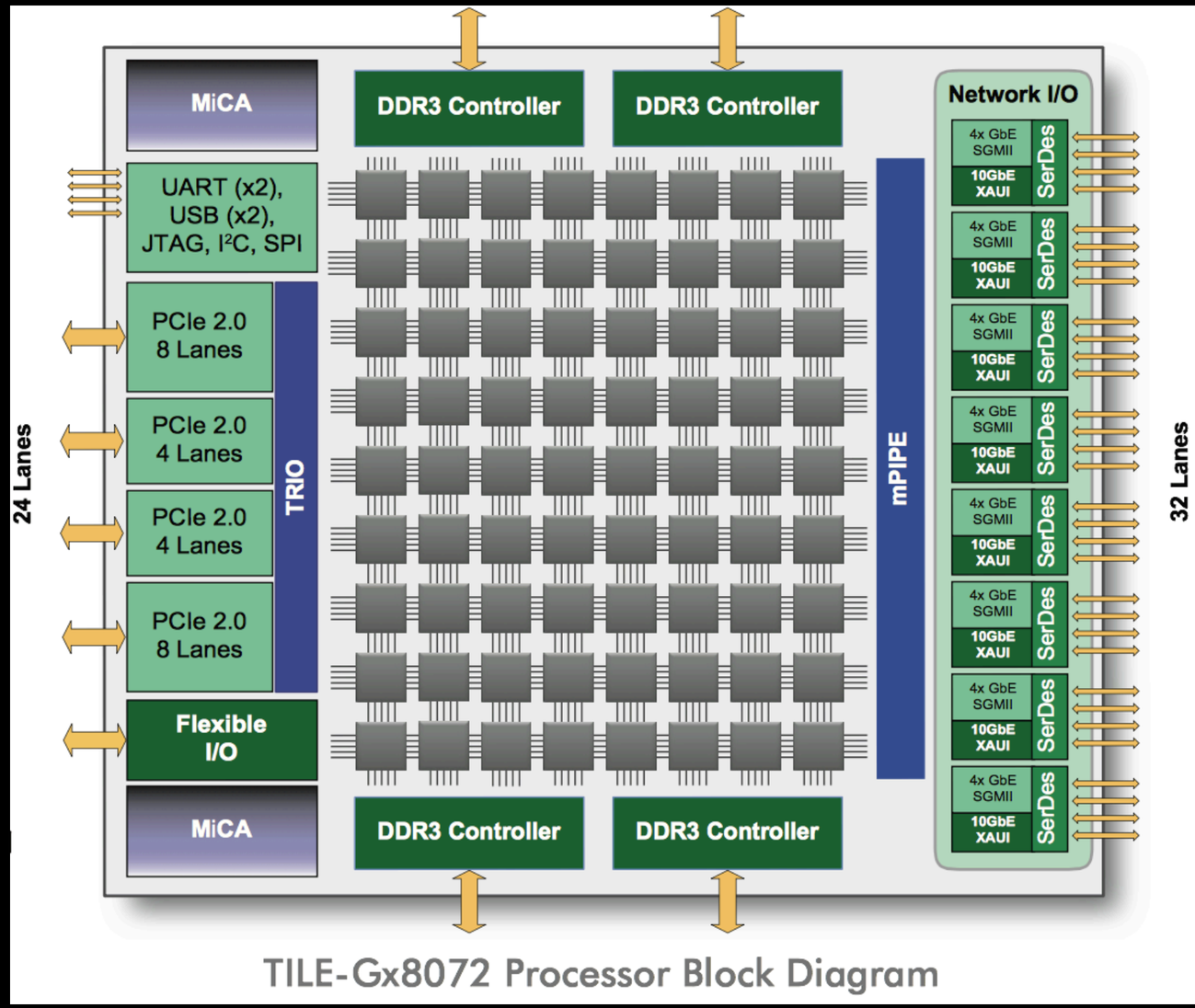
Disambiguation

- Reactive Programming ⇛ async data flows
- Functional Reactive Programming ⇛ async data flows + FP
- Reactive Application ⇛ architectural pattern
- Responsive Web-Application ⇛ Twitter Bootstrap


Stimulus (S)
in along sensory nerve
A response comes through the motor nerve
Synapses with other neurons in spine
Response (R)
synapses on motor fiber

Why Reactive: many cores

- End of the ~~single core~~ multi-core era
- Many players in the space
  - Tilera, Cavium
  - Adapteva Parallela
  - Xeon PHI



MX4
More than better

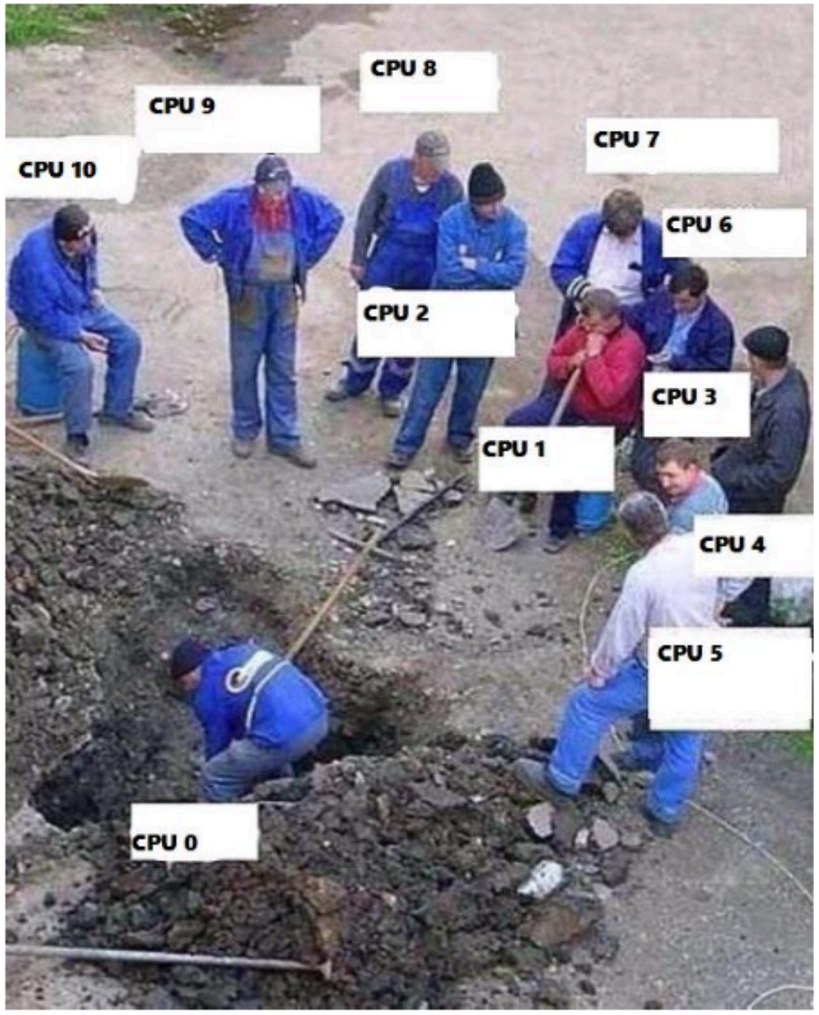
Why Reactive: many cores

- Meizu MX4 Ubuntu Edition
- **Octa-core** MediaTek MT6595 chipset
- 2GB RAM / 20.7 MP rear camera, 2MP front-facing / 16GB built-in flash storage

Özgür Akgün
@ozgurakgun

Follow

this is indeed how most programs work on a multicore computer!



RETWEETS 1,860 FAVORITES 964

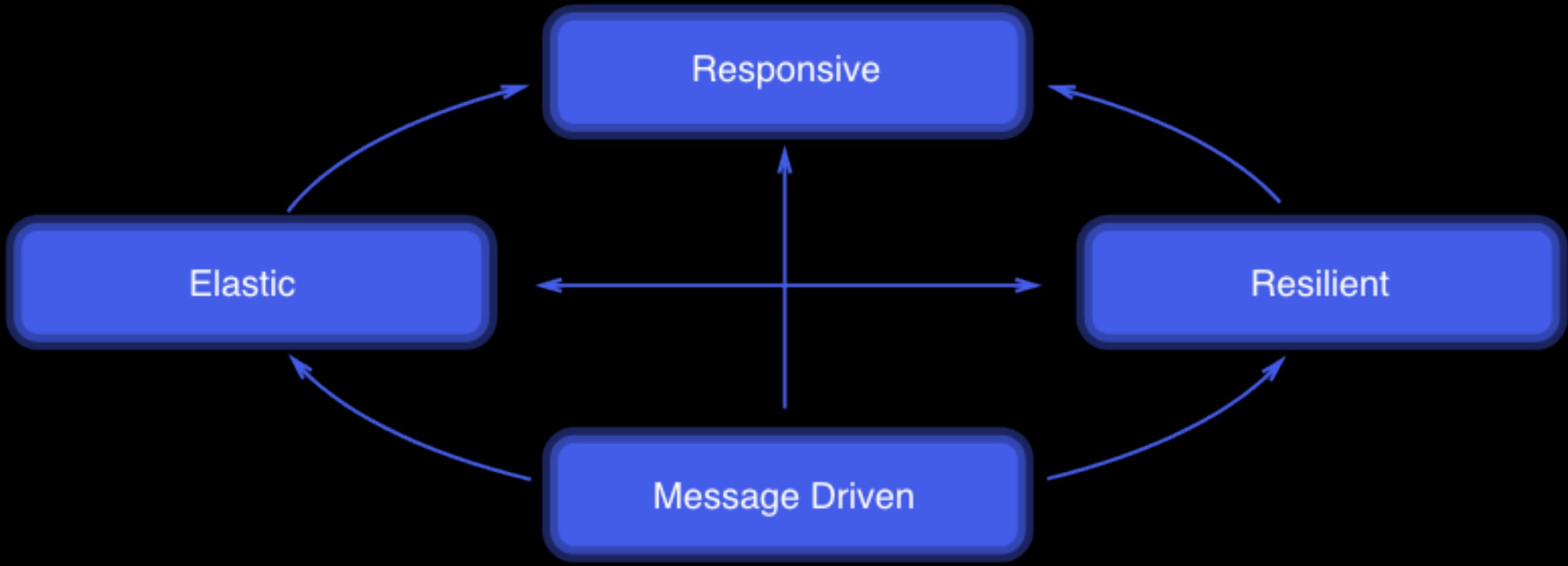
Why reactive: distribution (theory)

- scaling out to handle large loads
- scaling out / replication to handle node failure

Why reactive: distribution (reality)

- networks, networks, networks
- they fail all the time
- Jepsen series[1]

[1] http://aphyr.com


Responsive
Elastic
Resilient
Message Driven

Reactive: how?

```
public class PaymentController {


    public PaymentConfirmation makePayment(CreditCard card) { ... }

    public PaymentHistory getPastPayments() { ... }

}
```

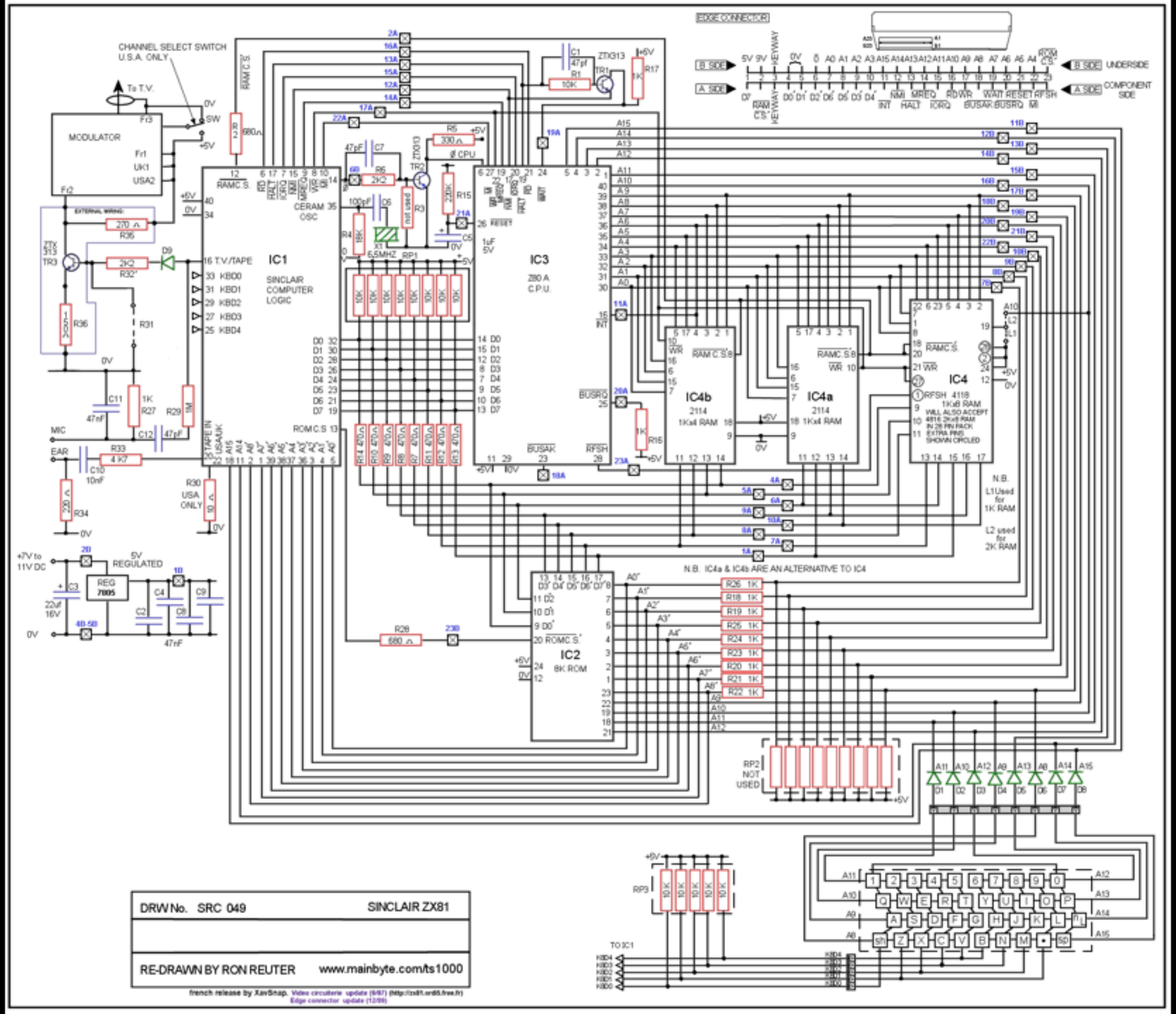
Reactive: how?

```
@Elastic(minNodes = 5, maxNodes = 15)
@Resilient(gracefullyHandleNetworkPartitions = true)
public class PaymentController {

    @Responsive(latency = 500, timeUnit = TimeUnit.MILLISECONDS)
    @MessageDriven(messageProvider = Provider.AKKA)
    public PaymentConfirmation makePayment(CreditCard card) { ... }

    @Responsive(latency = 800, timeUnit = TimeUnit.MILLISECONDS)
    public PaymentHistory getPastPayments() { ... }

}
```

Why Reactive: summary

- distribution accross CPU cores
- distribution accross networked machines
- need tooling to work with this type of distribution

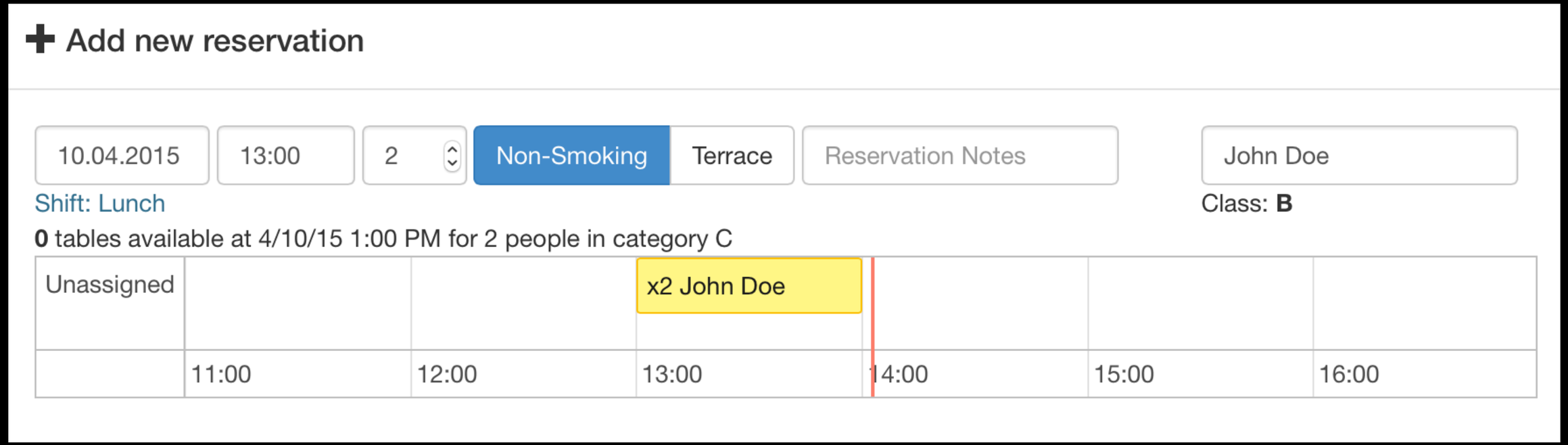

Mutable state


CHANNEL SELECT SWITCH
U.S.A. ONLY
To T.V.
MODULATOR
IC1
SINCLAIR
COMPUTER
LOGIC
IC3
Z80 A
C.P.U.
IC2
8K ROM
IC4
IC4a
IC4b
2114
1Kx8 RAM
CERAM
OSC
6.5MHZ
RESET
EDGE CONNECTOR
B.SIDE
A.SIDE
B.SIDE UNDERSIDE
A.SIDE COMPONENT SIDE
+7V to 11V DC
5V REGULATED
REG 7805
N.B. IC4a & IC4b ARE AN ALTERNATIVE TO IC4
N.B. L1Used for 1K RAM
L2 used for 2K RAM
RP2 NOT USED
TO 3C1
DRW No. SRC 049
SINCLAIR ZX81
RE-DRAWN BY RON REUTER
www.mainbyte.com/ts1000

Why mutable ?

- memory expensive!
- can't afford to keep past state in it
- re-use, overwrite, optimize

Mutable issues - example 1



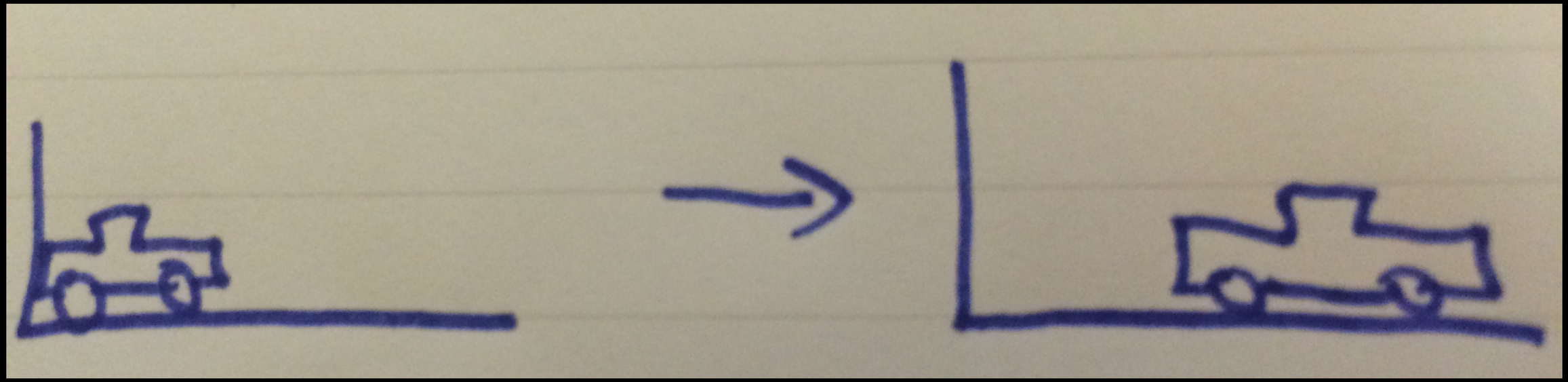
Mutable issues - example 1

```
$scope.reservation = {
  id: 42,
  start: moment({ hour: 13, minute: 15 }),
  end: moment({ hour: 14, minute: 30 })
};

timeline.setOptions({
  min: $scope.reservation.start.startOf('hour').toDate(),
  max: $scope.reservation.start.add(3, 'hour').toDate()
});
```

Mutable issues - example 1

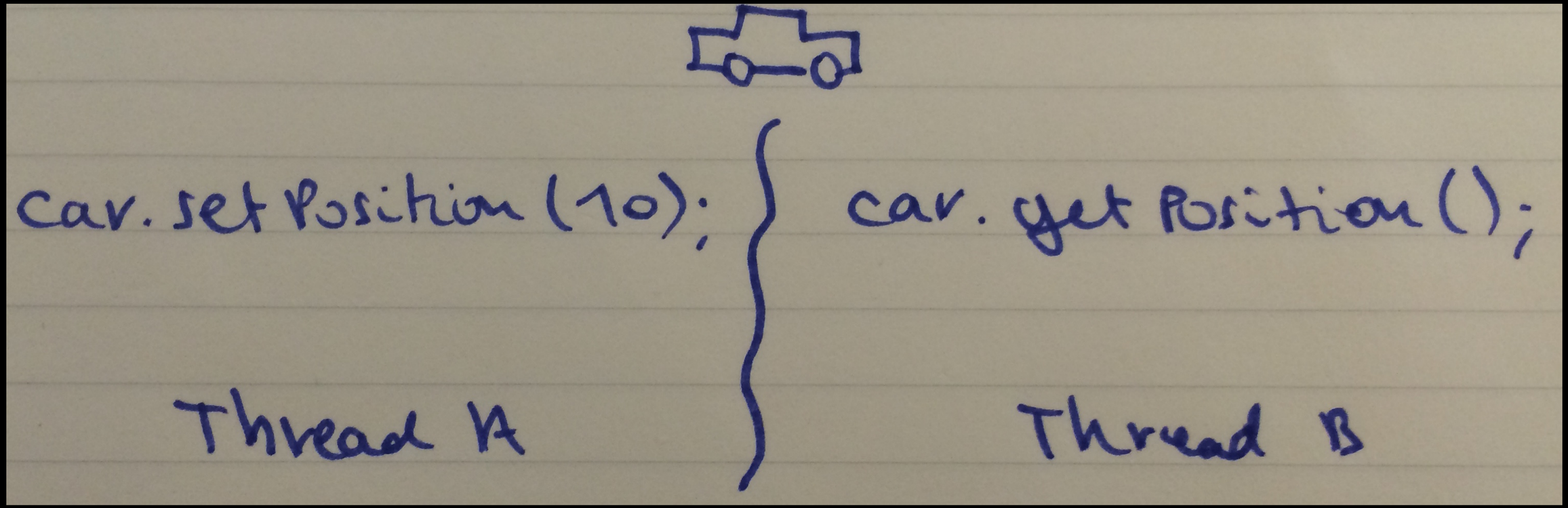
```
$scope.reservation = {
  id: 42,
  start: moment({ hour: 13, minute: 15 }),
  end: moment({ hour: 14, minute: 30 })
};

timeline.setOptions({
  min: $scope.reservation.start.clone().startOf('hour').toDate(),
  max: $scope.reservation.start.clone().add(3, 'hour').toDate()
});
```

Mutable issues - example 2



```
car.setPosition(0);
car.setPosition(10);
```

Mutable issues - example 2



The problem with locks / latches

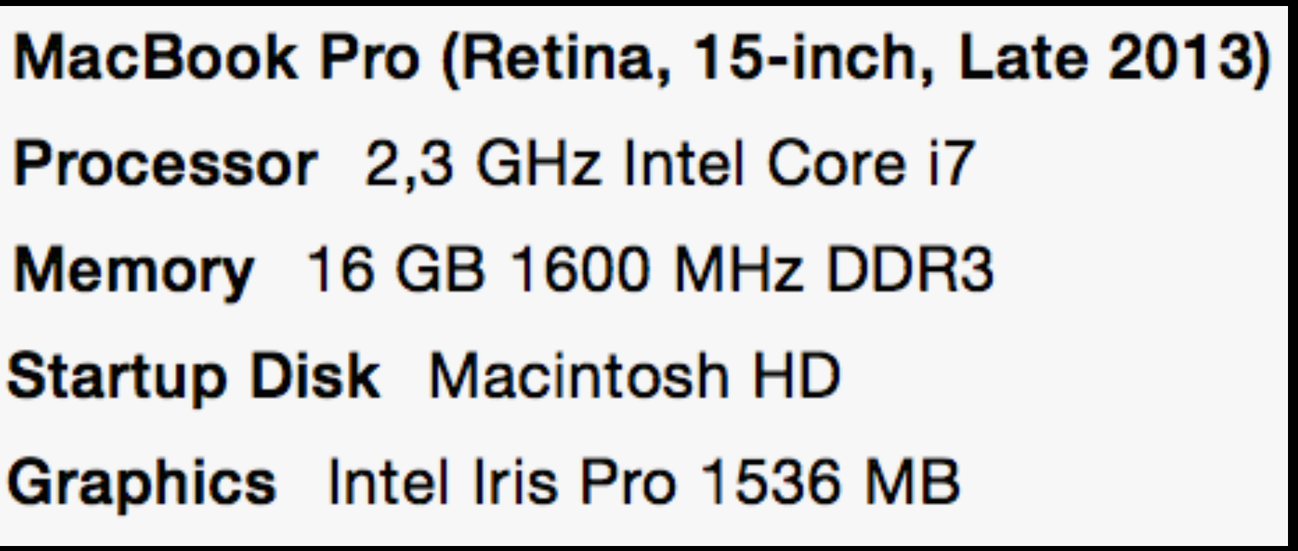
- ~~solution~~ workaround for a broken conceptual model
- huge coordination overhead! Even more so when distributed
- hard to reason about
- performance hit

Mutability: summary

- increased difficulty for the programmer (moving parts)
- makes life hard when working concurrently


Immutable
state

Immutable state - why now?



- main memory is cheap!
- disk memory is cheap!

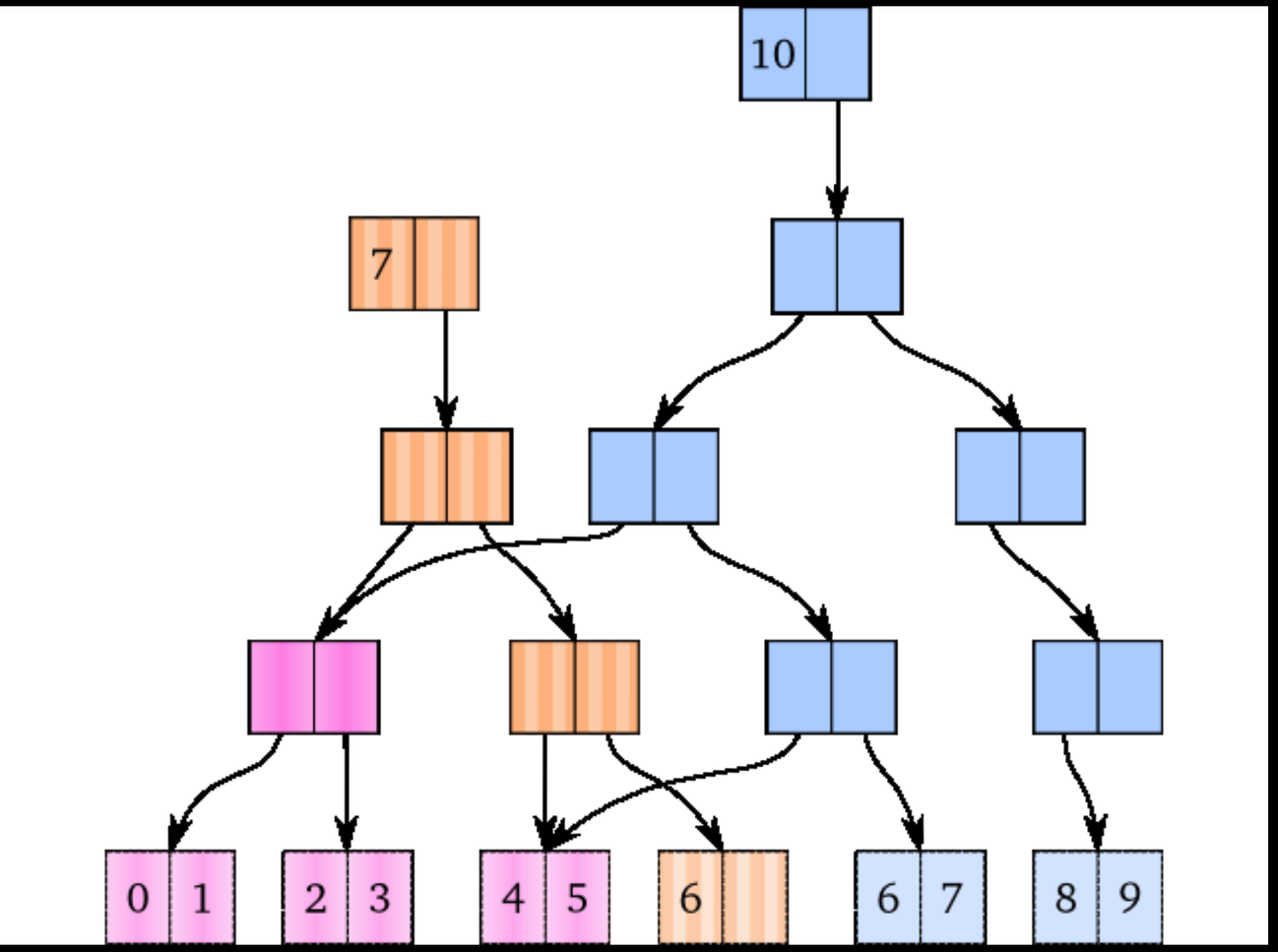
We can afford **copies of past state** around in order to **reduce coordination efforts**

Immutable state - how?

```
case class Car(brand: String, position: Int)

val car = Car(brand = "DeLorean", position = 0)
val movedCar = car.copy(position = 10)
val movedCarLaterOn = car.copy(position = 30)
```

Working with different version

"Snapshots" of reality



Immutable state - how?

- clever immutable data structures, e.g. Bitmapped Vector Trie [2]
- do not copy data around - point to unchanged data instead
- constant time for all operations

[2] http://lampwww.epfl.ch/papers/idealhashtrees.pdf

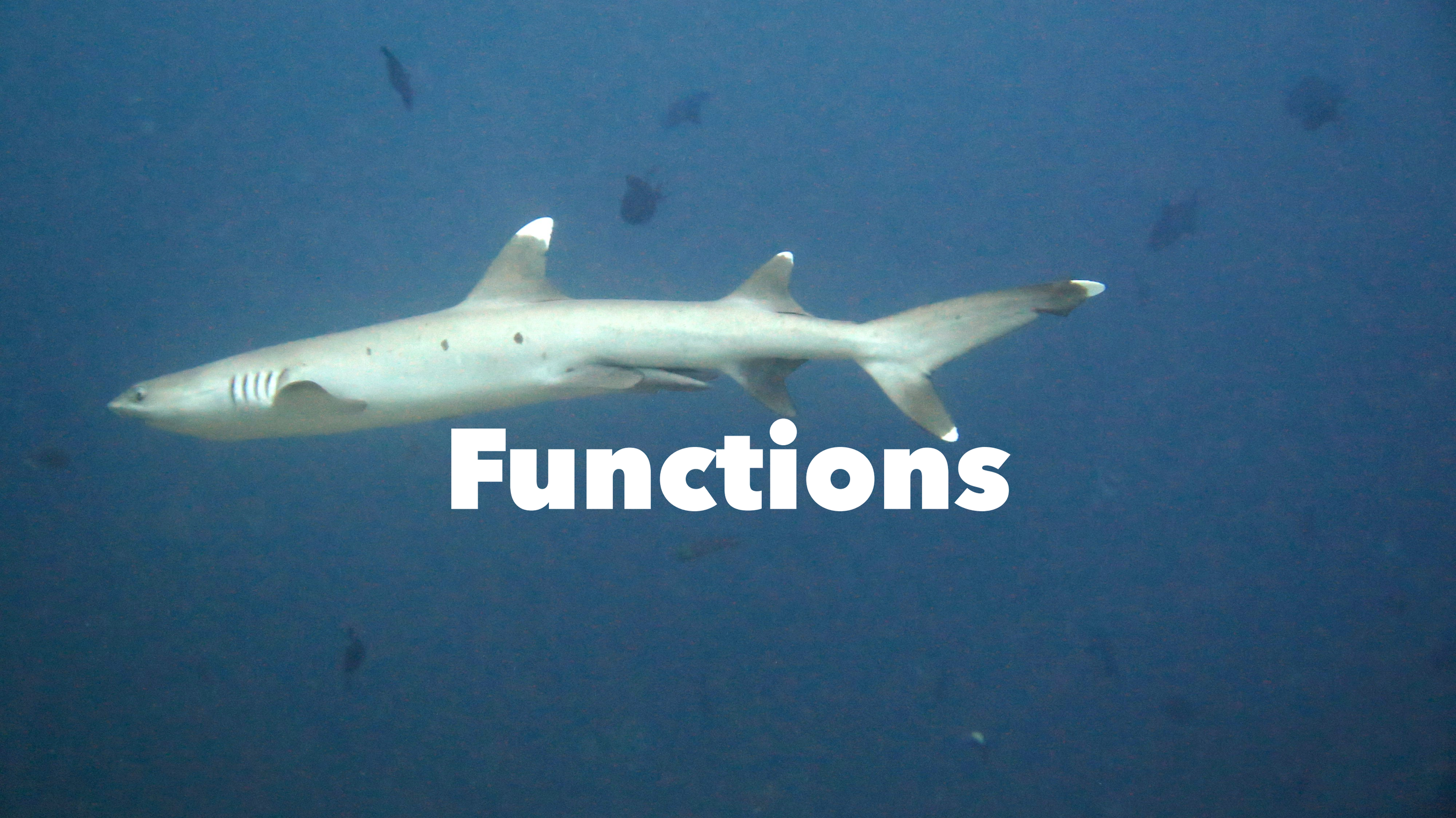
Immutable all the way down

- immutability changes everything [3]
  - programming languages
  - databases: insert-only, event stores
  - SSD drives

[3] http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf

Immutability: summary

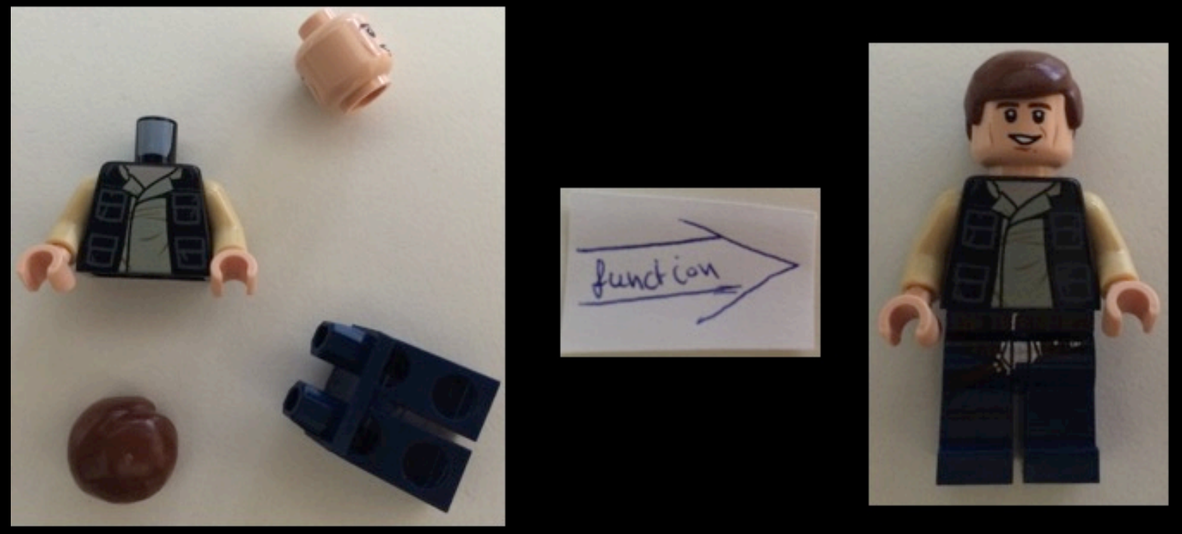
- we can afford to keep everything, with good performance
- reduces the headeache of coordination accross CPU cores and networked nodes
- audit trail of changes for free



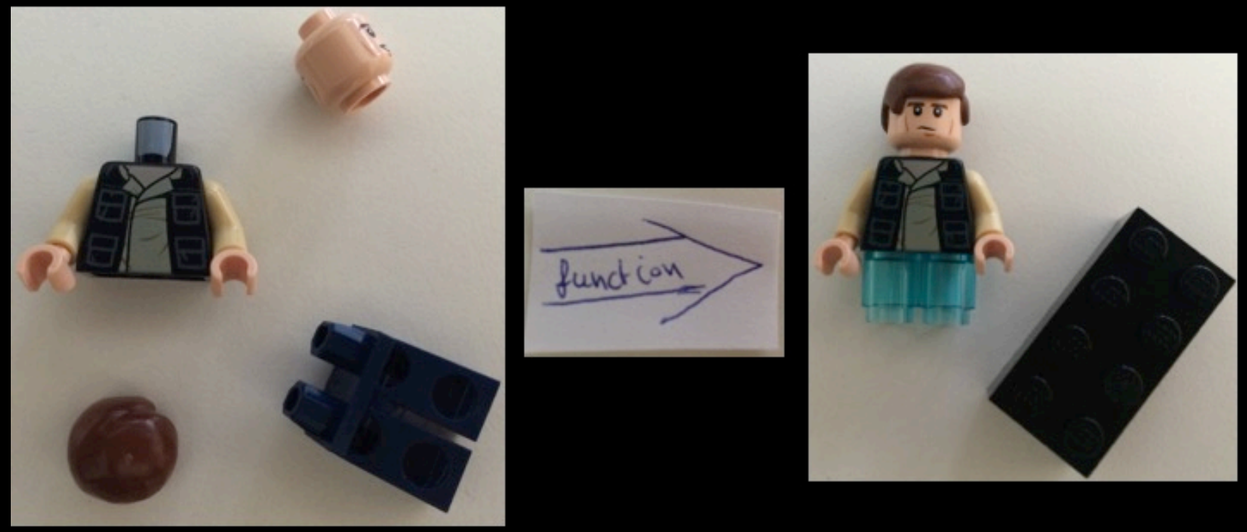
Functions

Functions, the Starwars Lego way
(three kinds of awesome united)

Pure function



Side-effecting function

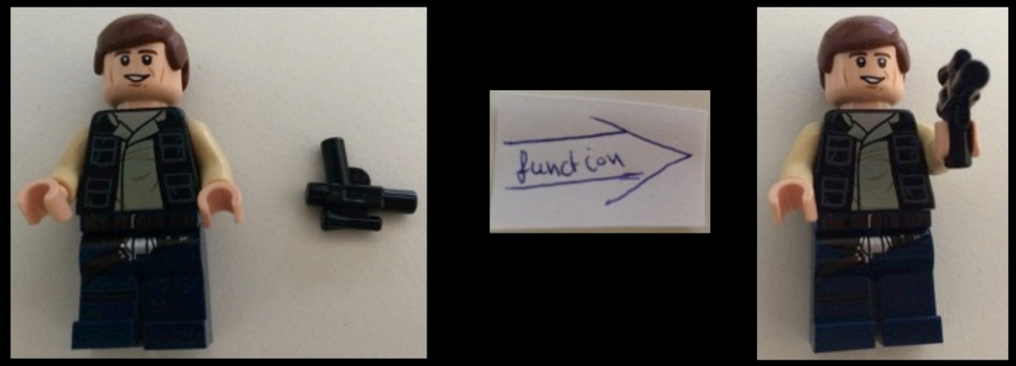


Side-effecting function

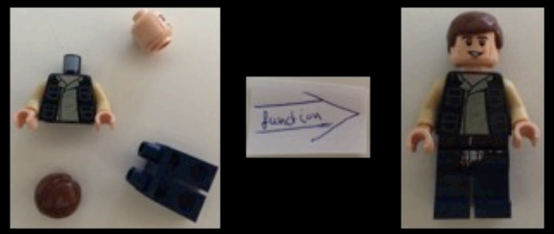
Side-effecting function

The dark side clouds everything. Impossible to see the future is.
-- Master Yoda

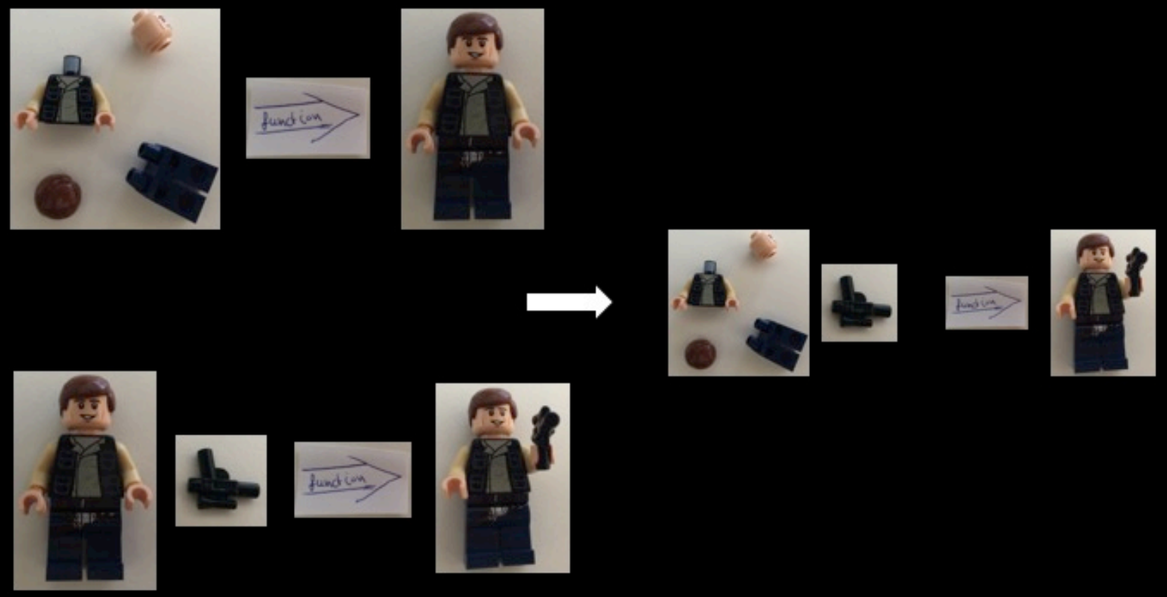
Again a pure function
(this time with a laser gun)



Hmm...



Function composition



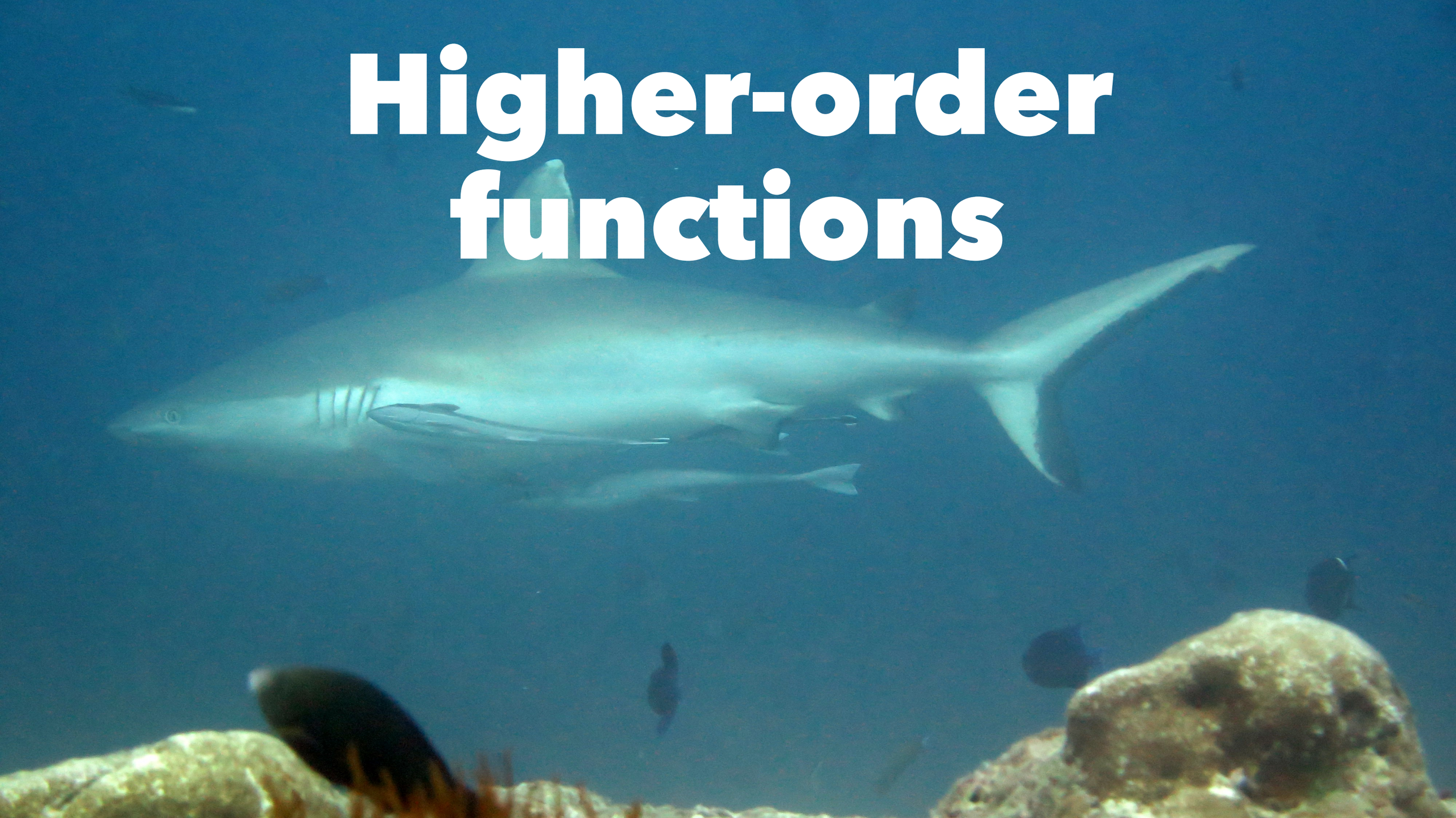
Function composition

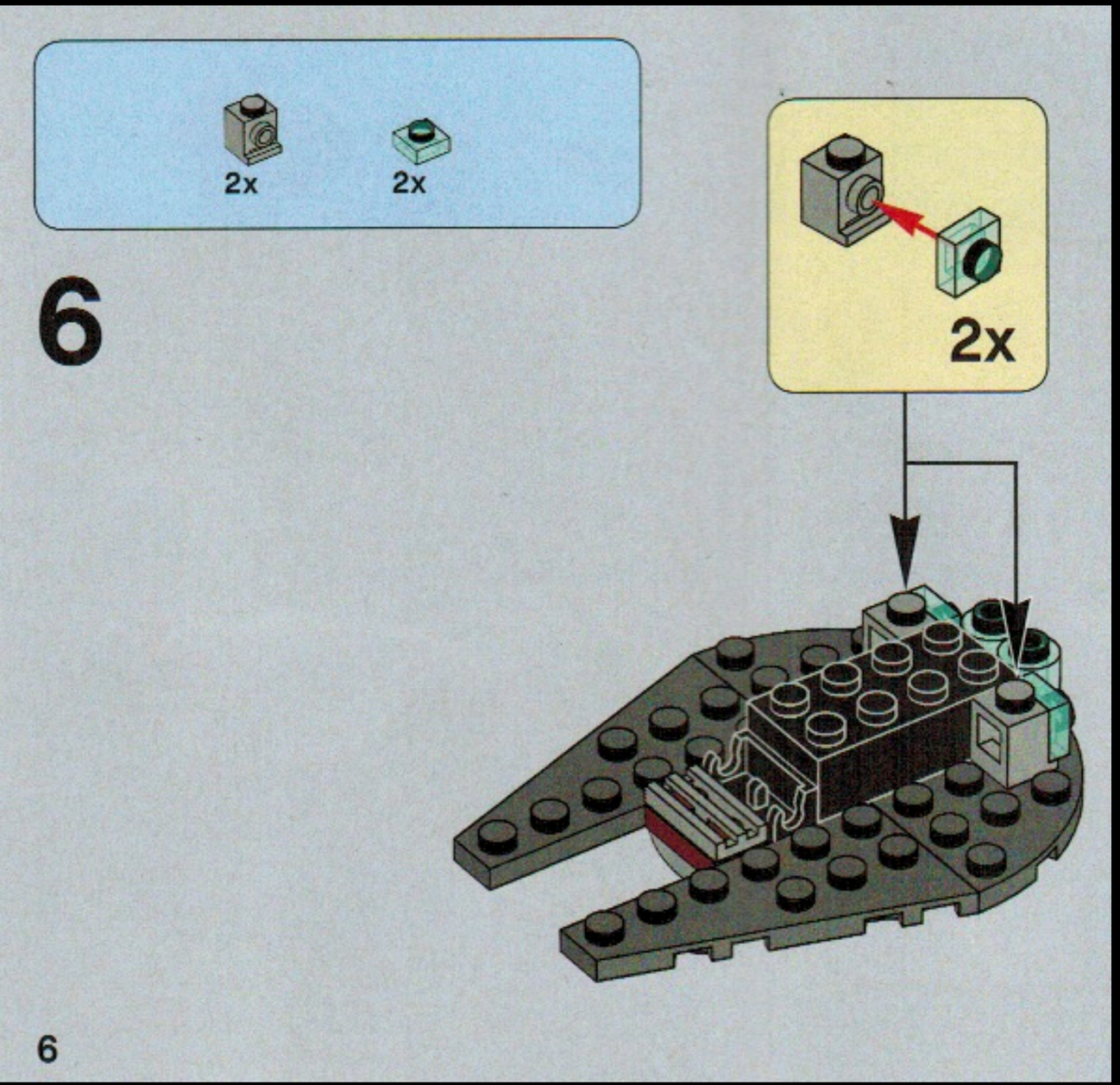
```
def assemble(parts: (Head, Body, Legs, Hair)): HanSolo = ...

def arm(h: HanSolo, lg: LaserGun): ArmedHanSolo = ...
```

Function composition

```
def assemble(parts: (Head, Body, Legs, Hair)): HanSolo = ...

def arm(h: HanSolo, lg: LaserGun): ArmedHanSolo = ...

def build(parts: (Head, Body, Legs, Hair), lg: LaserGun):
  ArmedHanSolo =
    arm(assemble(parts), lg)
```

Higher-order functions



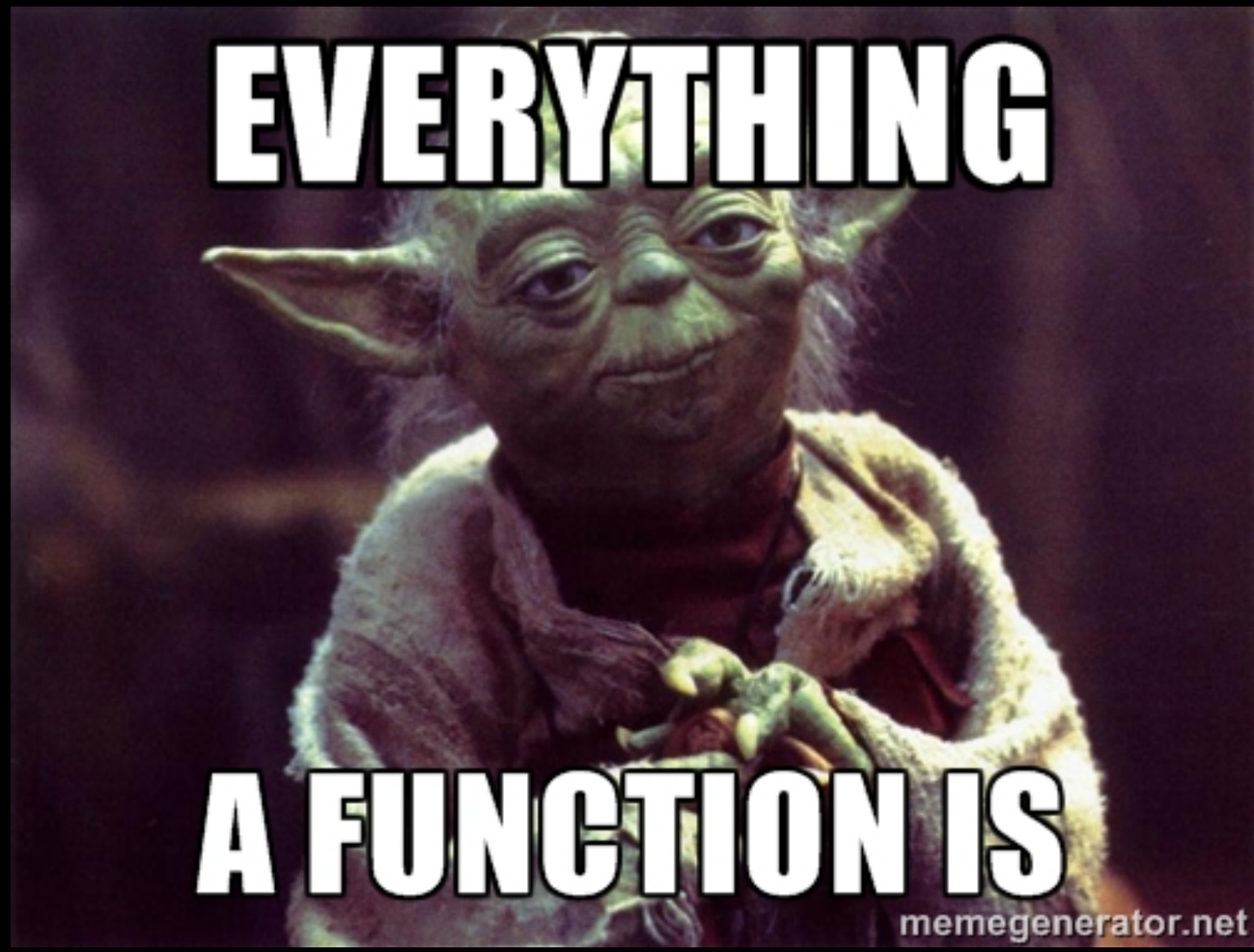


Definition

A function that takes *another function* as parameter (or produces a function as result).

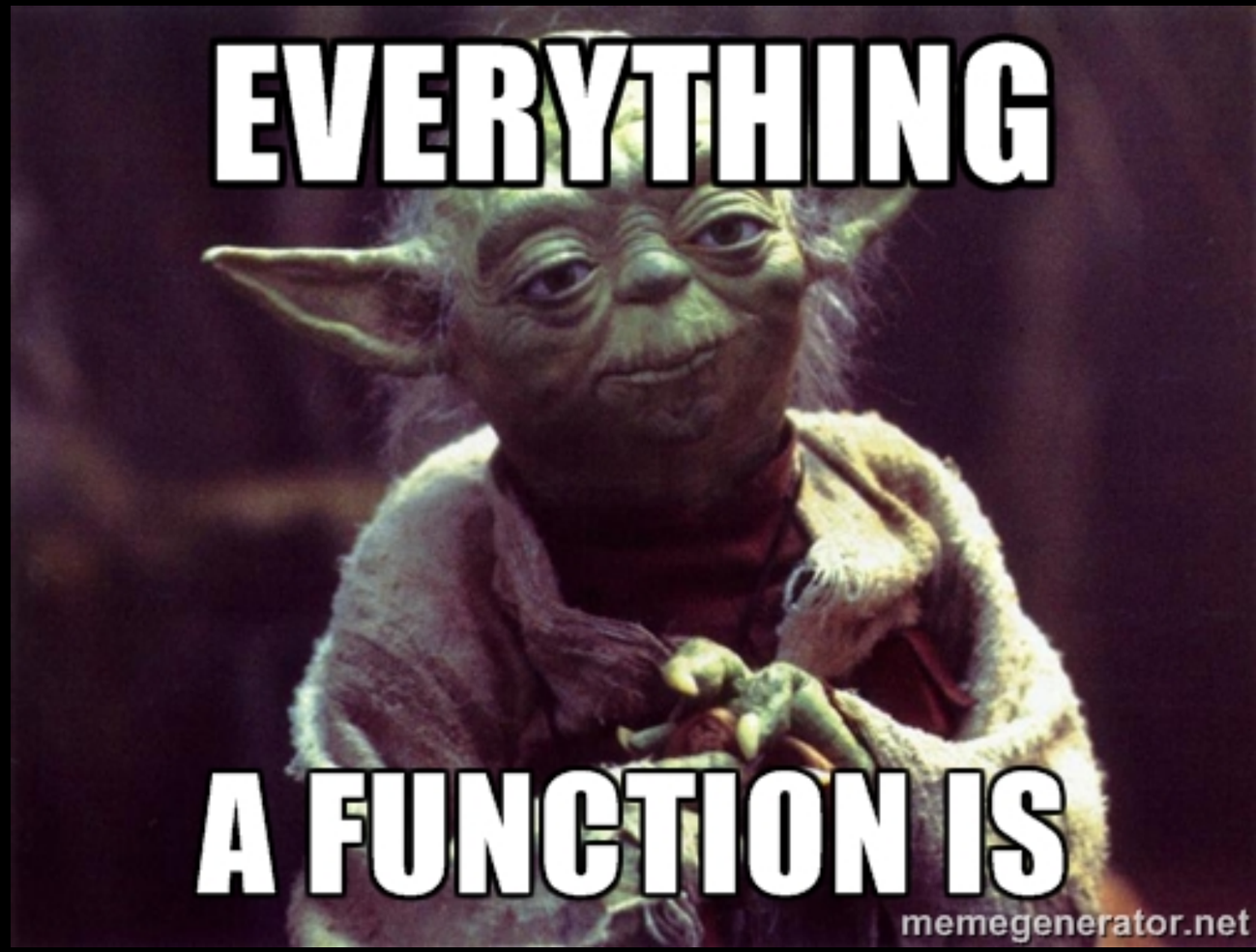
Higher-order functions

```
val users: List[User] = ...

val (minors, majors) =
  users.partition(_.age < 18)
```



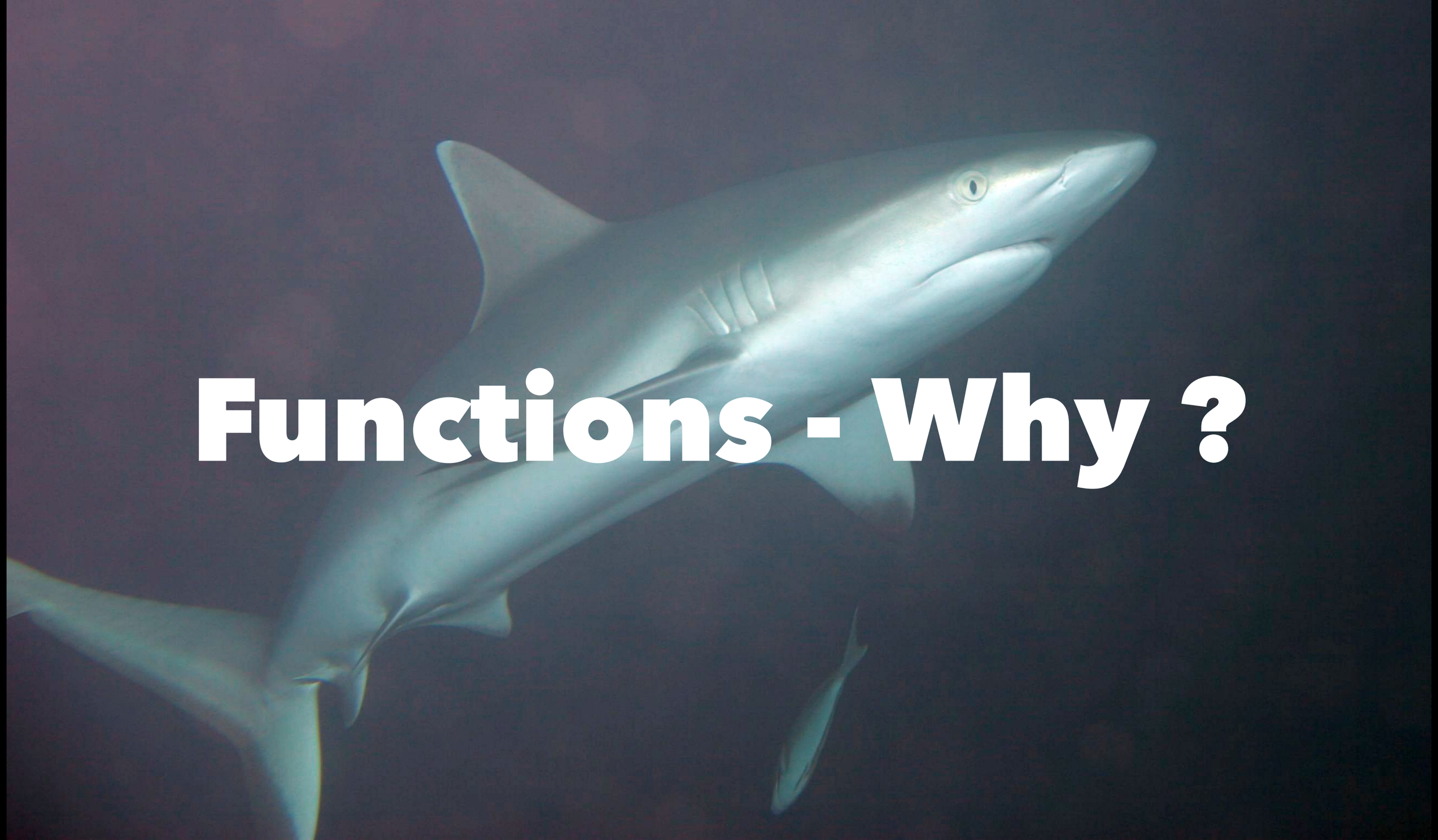
Higher-order functions

```
val users: List[User] = ...

val isMinor =
  (user: User) => user.age < 18

val (minors, majors) =
  users.partition(isMinor)
```



Higher-order functions

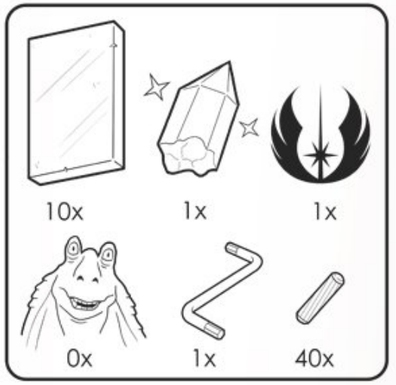
```
def AuthenticatedAction(f: Request => User => Result) = Action { request =>
  findUser(request).map { user =>
    f(request)(user)
  } getOrElse {
    Unauthorized("Get out!")
  }
}

def showSettings = AuthenticatedAction { request =>
  user =>
    userSettingsService.findSettings(user).map { settings =>
      Ok(views.html.settings(user, settings))
    } getOrElse {
      NotFound("We lost all your settings. Sorry.")
    }
}
```

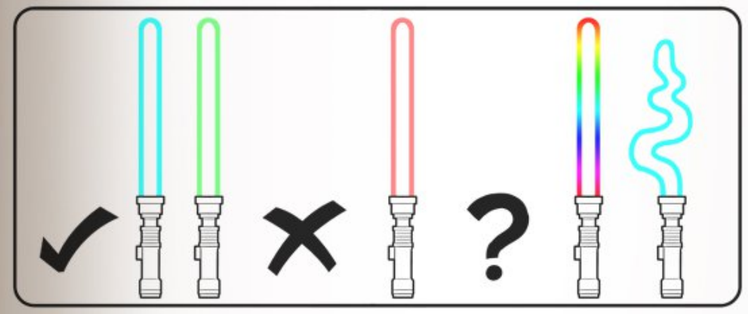

Functions - Why ?


LEGO
STAR WARS
75030
MICROFIGHTERS

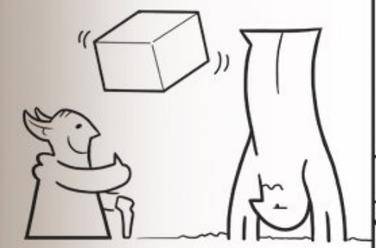
LITSABBUR

IKEA



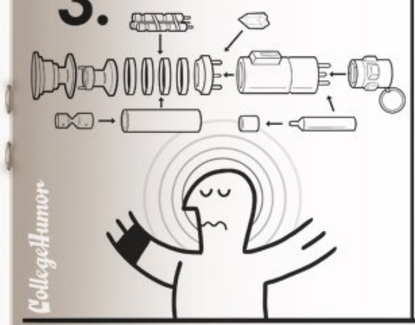


1.

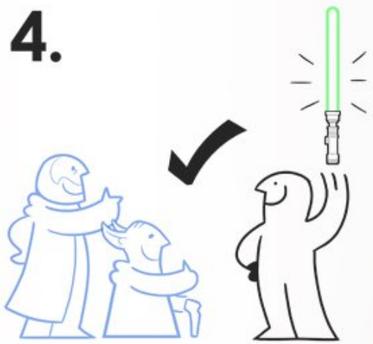


2.

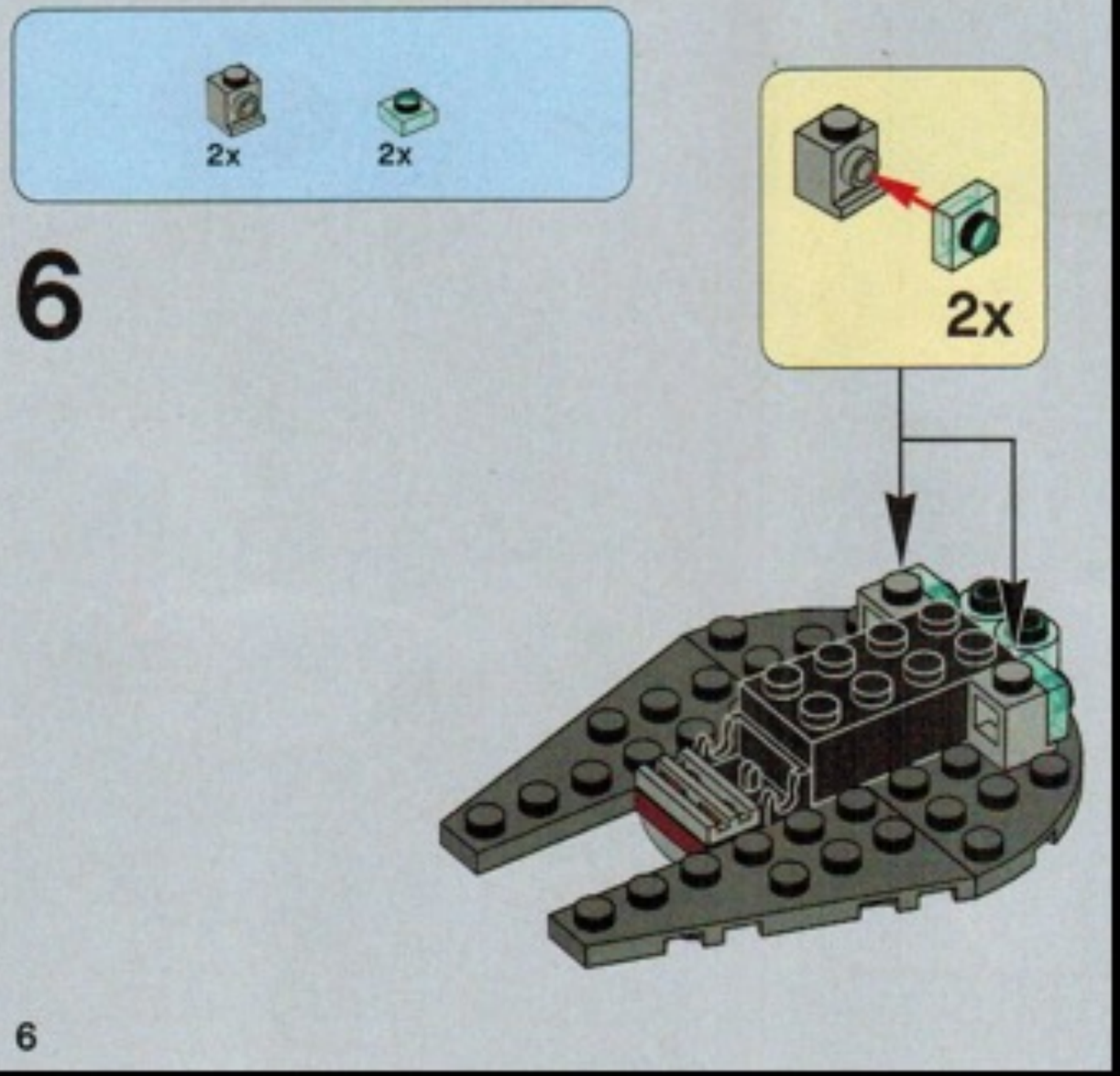
3.

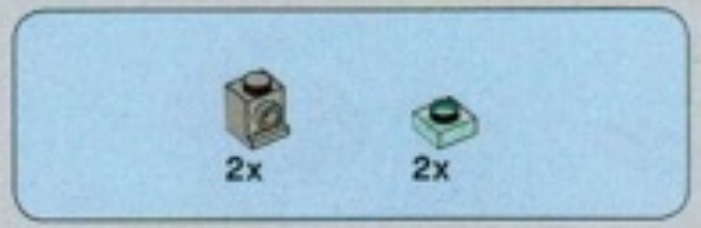


4.

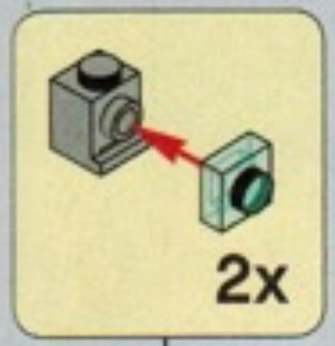


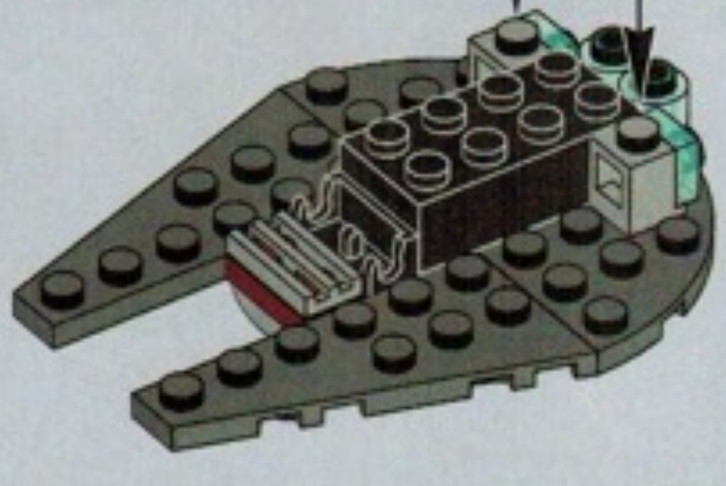
CollegeHumor

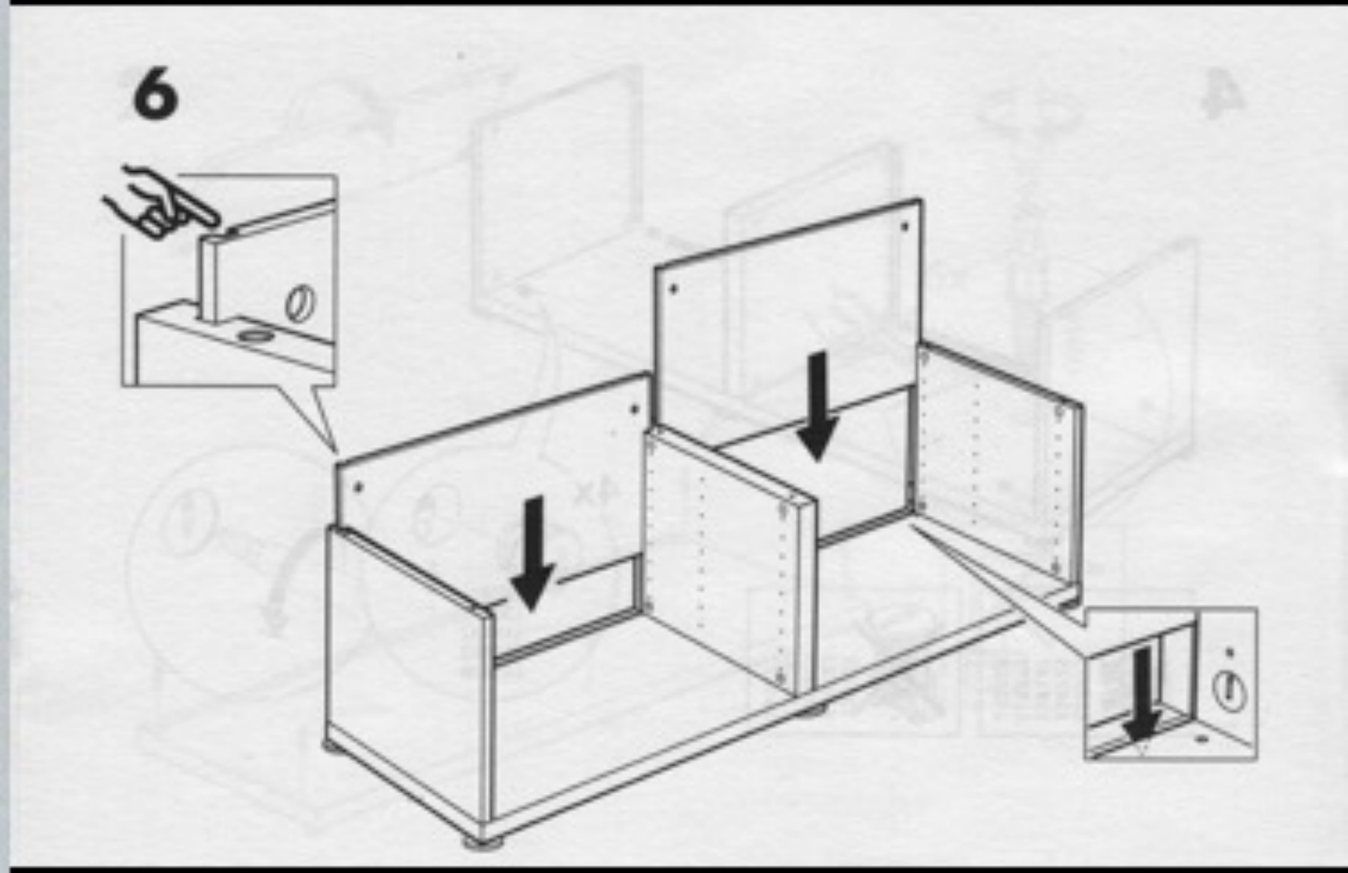

2x
2x
6
2x
6

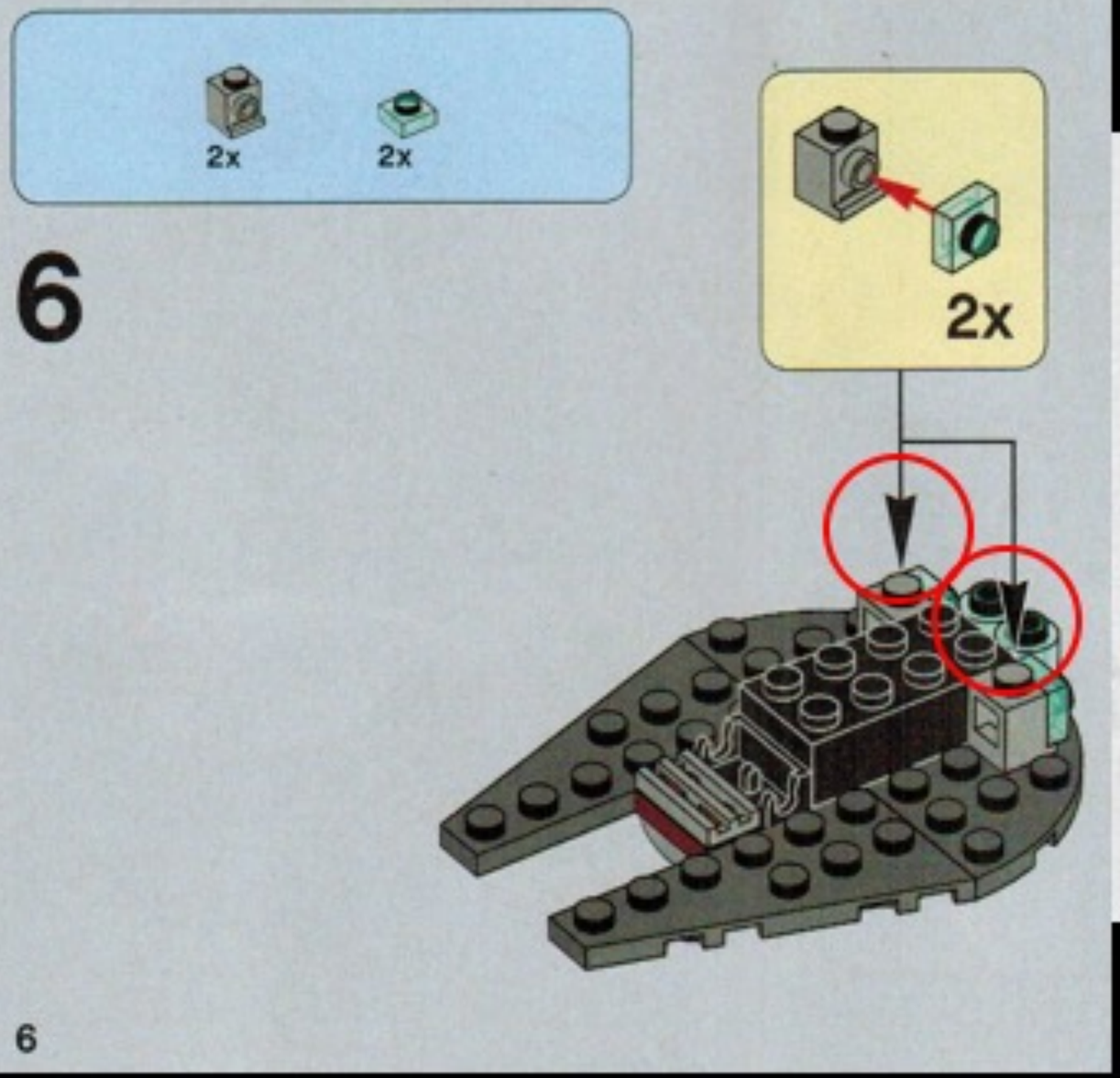


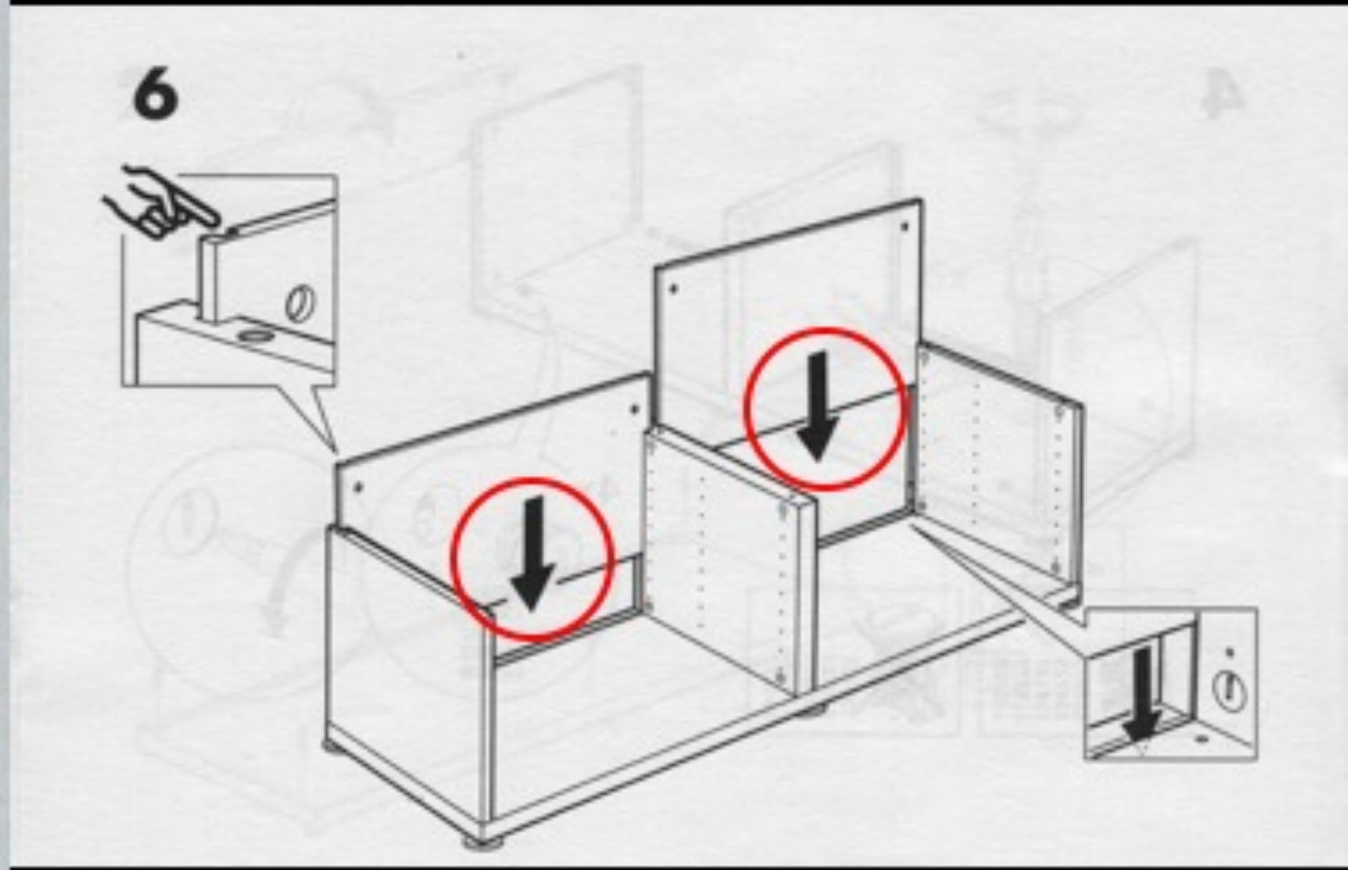
6

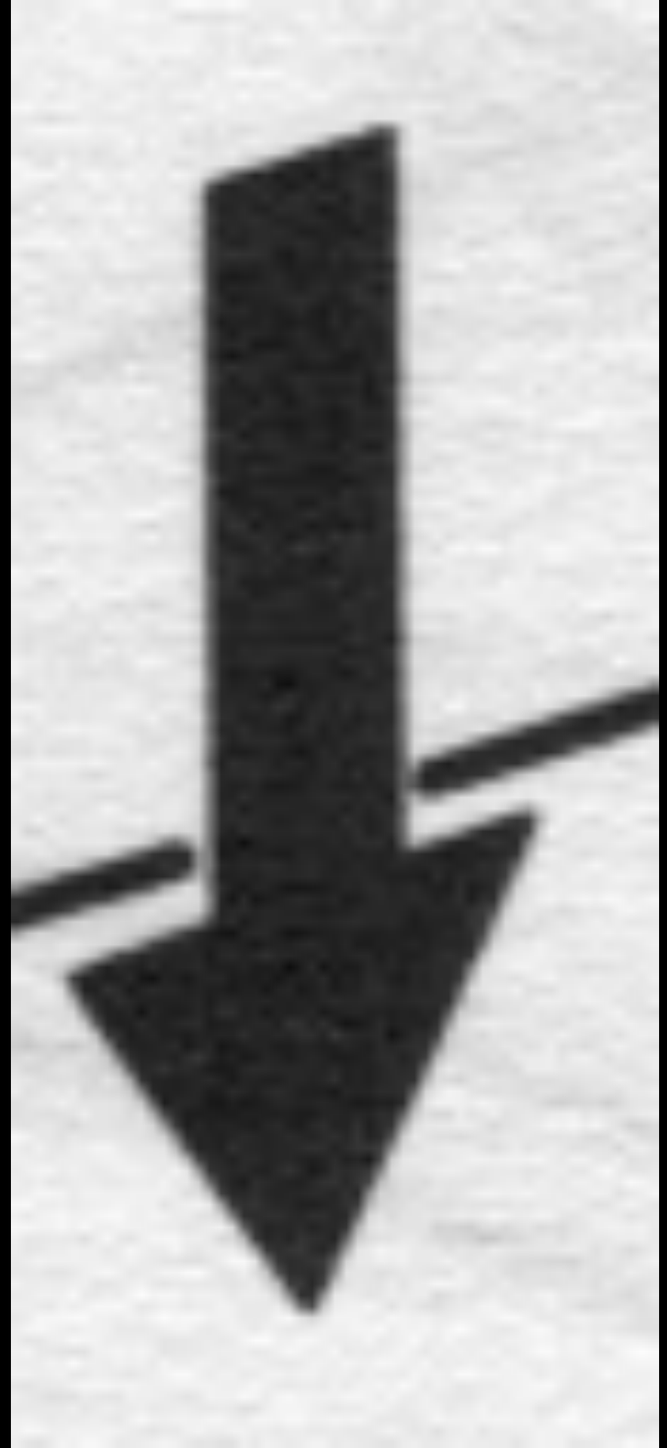







2x
2x
6
2x
6
6


6



Functions

- portable and re-usable behaviour
- data changes, behaviour can be re-used
- functions as data transformation pipelines

Functions = data transformation pipelines

```
val addresses = users.filter(_.age > 18)
                     .map(_.address)
                     .sortBy(_.city)
```

Build increasingly complex behaviour through a series of transformations driven by composing functions

Imperative vs. Functional Separation of Concerns

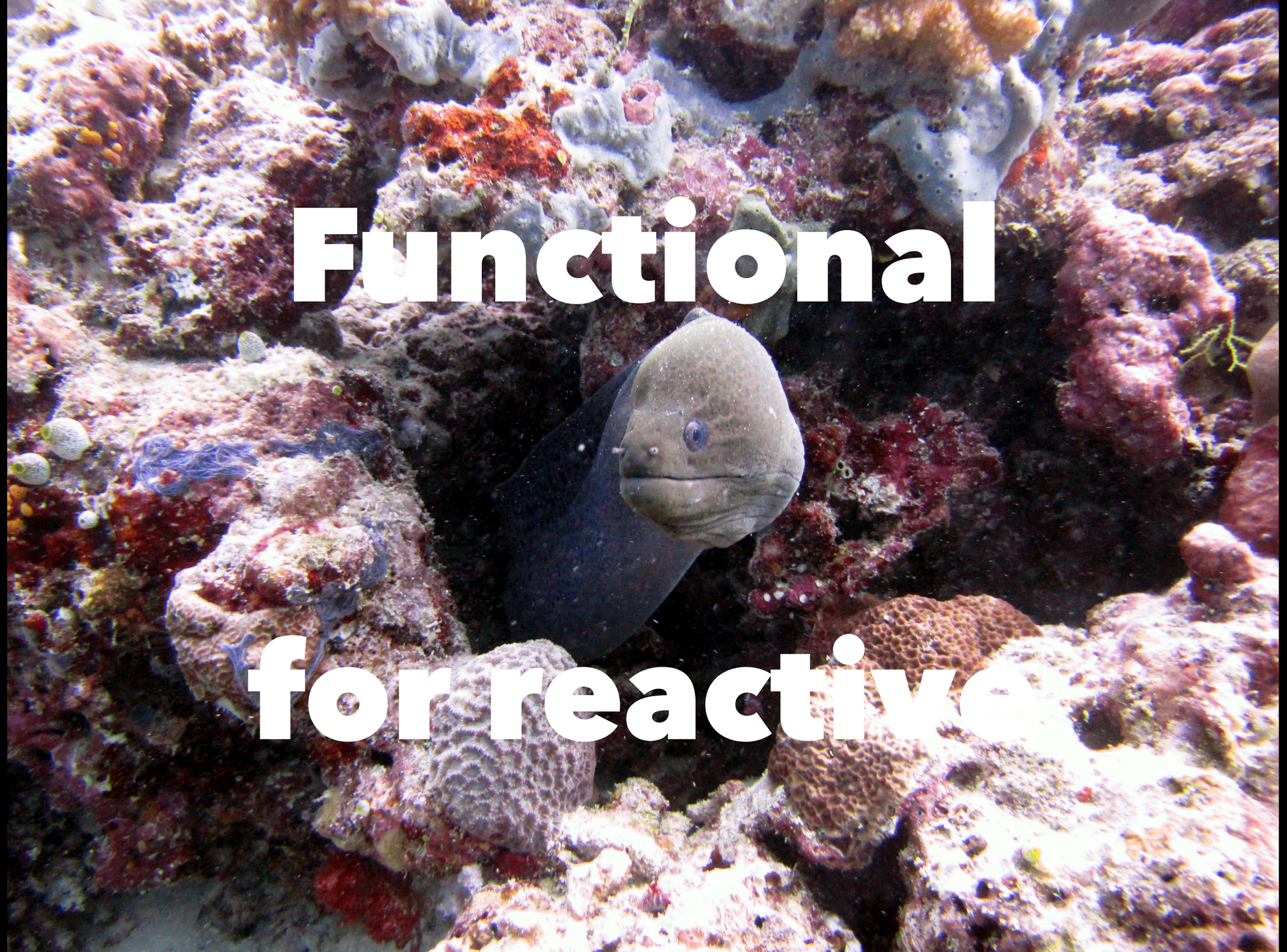
```
public List<String> getErrors(String fileName) throws Exception {
    List<String> errors = new ArrayList<>();
    BufferedReader reader =
                    new BufferedReader(new FileReader(fileName));
    String line = reader.readLine();
    for (int lineCounter = 0; lineCounter < 100 && line != null;
                        lineCounter++, line = reader.readLine()) {
        if (line.startsWith("ERROR")) errors.add(line);
    }
    return errors;
}
```

```
public List<String> getErrors(String fileName)
                                                throws Exception {
    return Files.lines(Paths.get(fileName))
                .limit(100)
                .filter(l -> l.startsWith("ERROR"))
                .collect(toList());
}
```

Mario Fusco @mariofusco · Mar 2

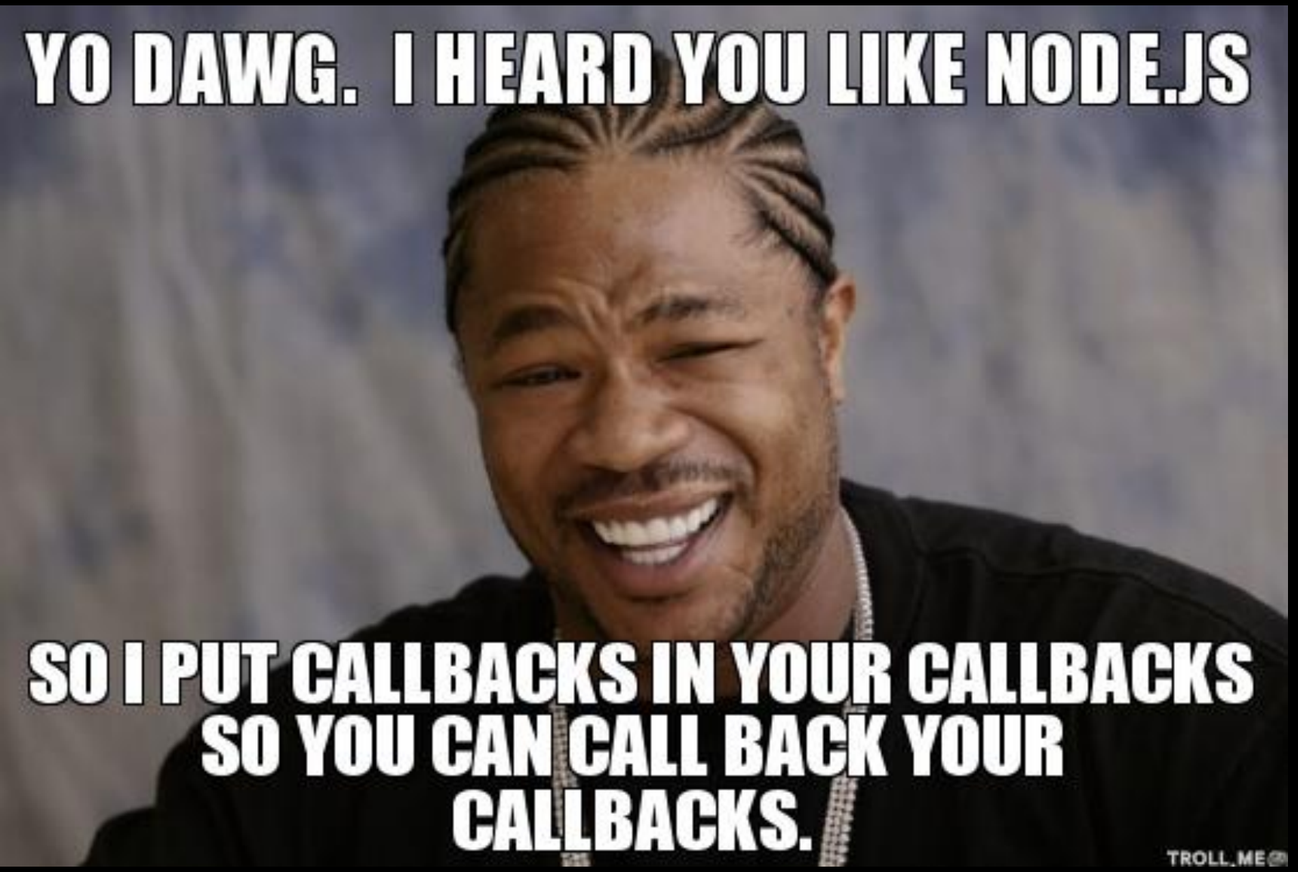
Imp vs. Func example reviewed (thanks for feedback). If you think Imp is unfair and can be improved, show me the code

47 39


Functional
for reactive

Reactive applications

- distributed in nature
- need to be resilient to failure, adapt to changes
- **asynchronous all the way down**



Asynchronous callback hell

```
var fetchPriceList = function() {
    $.get('/items', function(items) {
        var priceList = [];
        items.forEach(function(item, itemIndex) {
            $.get('/prices', { itemId: item.id }, function(price) {
                priceList.push({ item: item, price: price });
                if ( priceList.length == items.length ) {
                    return priceList;
                }
            }).fail(function() {
                priceList.push({ item: item });
                if ( priceList.length == items.length ) {
                    return priceList;
                }
            });
        }
    }).fail(function() {
        alert("Could not retrieve items");
    });
}
```

Asynchronous & functional

```
val fetchItems = WS.get("/items").getJSON[List[Item]]()
val fetchPrices = WS.get("/prices").getJSON[List[Price]]()

val itemPrices: Future[List[(Item, Option[Price])]] = for {
  items <- fetchItems
  prices <- fetchPrices
} yield {
  item -> items.flatMap { item =>
    prices.find(_.itemId == item.id)
  }
}

itemPrices.recover {
  case ce: ConnectionException =>
    log.error("Could not retrieve items")
    List.empty
}
```

Immutable Function Composition

Thank you

http://www.manning.com/bernhardt
@elmanu / manuel@bernhardt.io

Questions?