understanding clojure through data

James Reeves

@weavejester

boolean·knot

disclaimer

I'll be telling a few lies

disclaimer

I'll be telling a few ~~lies~~ simplifications

clojure

a **functional** programming language

a **lisp** (more on that later)

officially targets Java (Clojure)
.NET (ClojureCLR)
javascript (ClojureScript)

first released in September 2007

by Rich Hickey

let's start with edn

extensible data notation

edn ⊆ clojure

json ⊆ javascript

json		edn
`3.14159`	numbers	`3.14159`
`"hello world"`	strings	`"hello world"`
`[1, 2, 3, 4]`	vectors	`[1 2 3 4]`
`{"name": "alice"}`	maps	`{"name" "alice"}`
`true`	booleans	`true`
`null`	nil	`nil`

collections

json

array	`[1 2 3]`	ordered, random access

collections

edn

vector	`[1 2 3]`	ordered, random access
list	`(1 2 3)`	ordered
set	`#{1 2 3}`	unordered, distinct

identifiers

json

string	`"name"`	text data and identifier

identifiers

edn

string	"name"	text data
keyword	:name	references itself
symbol	name	references something else

example

```
{:name "Alice"
 :sex  :female
 :job  cryptographer}
```

universal definition

might reference job description, benefits, etc

namespaces

```
:status/ready
mammal.canine/dog
```

tags

```
#inst "1985-04-12T23:20:50.52Z"

#uuid "f81d4fae-7dec-11d0-a765-00a0c91e6bf6"
```

tags

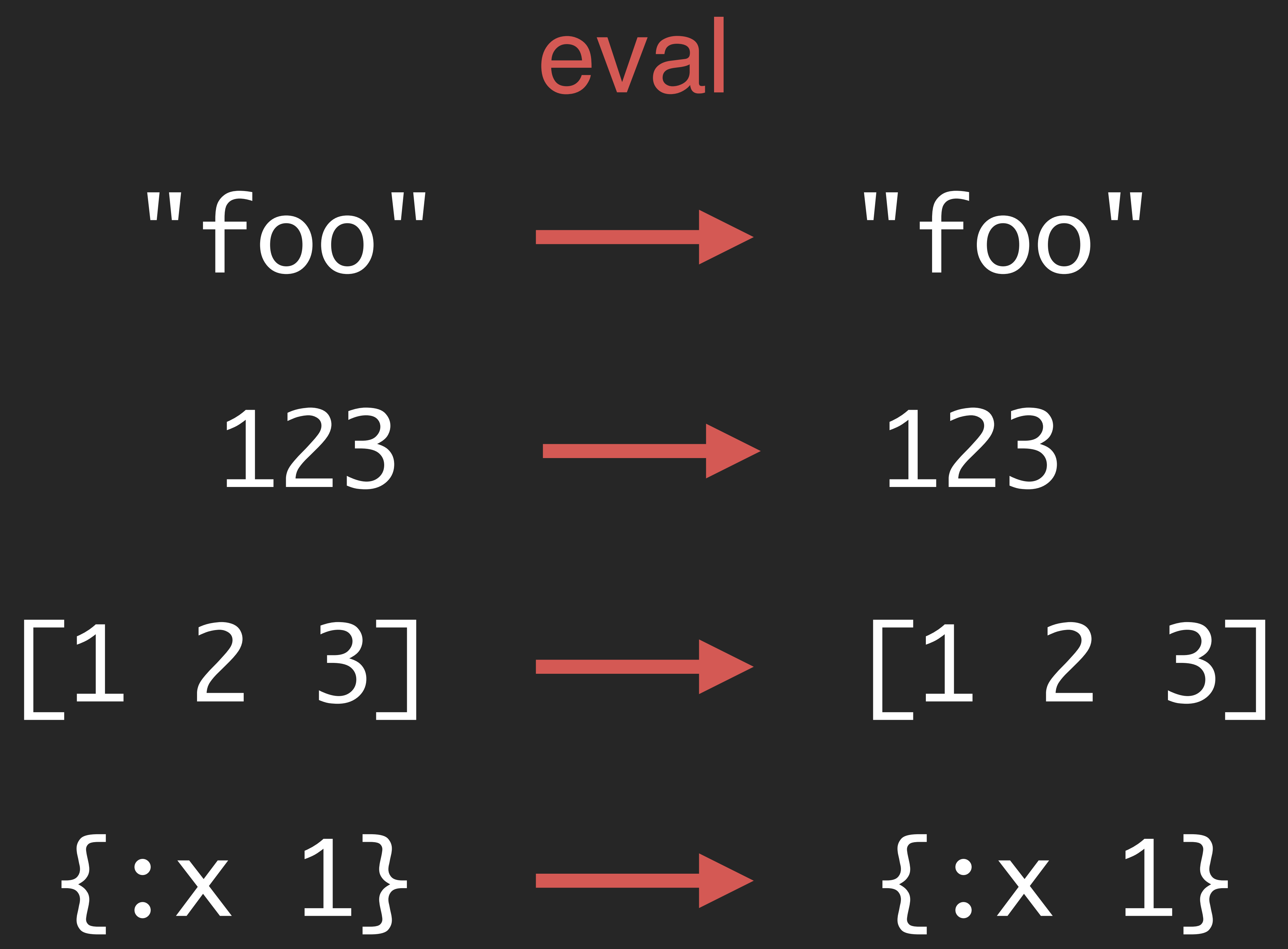
```
#color/rgb "e8a433"

#color/rgb [232 164 51]
```

what's the connection to **clojure**?

clojure = edn + eval

evaluation

most data evaluates to itself

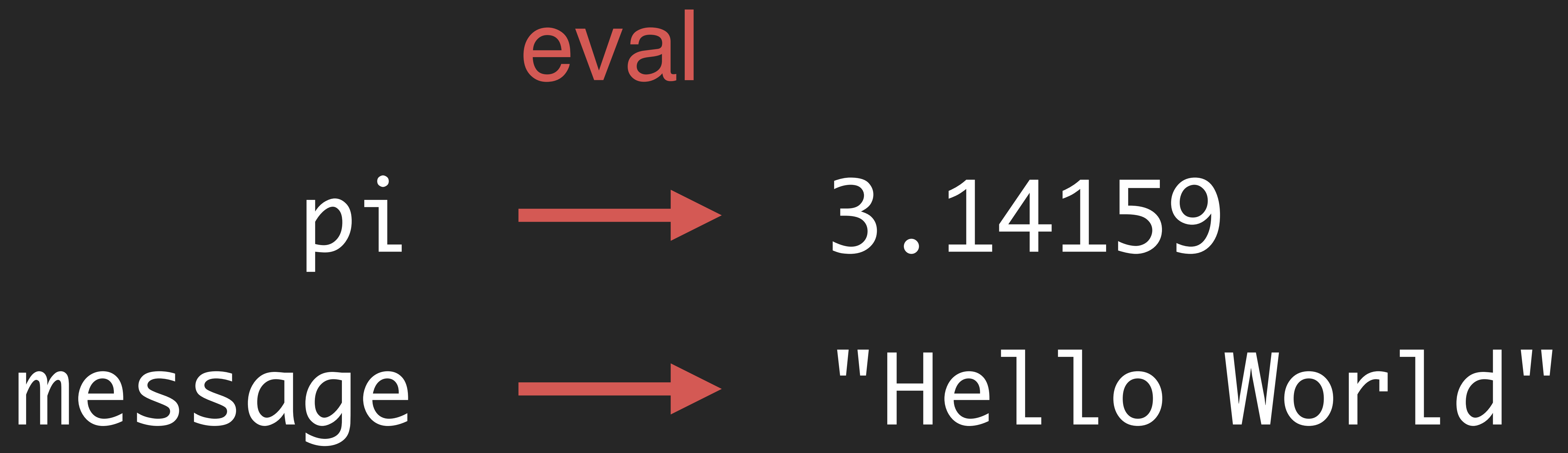


eval

```
"foo"   →   "foo"
 123    →    123
[1 2 3] →  [1 2 3]
{:x 1}  →  {:x 1}
```

evaluation

symbols evaluate to a bound value



eval

pi → 3.14159

message → "Hello World"

evaluation

lists evaluate based on their first element

eval

`(+ 1 1)` → `2`

`(and true false)` → `false`

evaluation

functions evaluate their arguments

```
(+ (* 3 3) (* 4 4))

⇒ (+ 9 16)

⇒ 25
```

evaluation

macros evaluate their return value

```
(postfix (9 16 +))

⇒ (+ 9 16)

⇒ 25
```

homoiconic

homo · iconic

the same representation

homo · iconic

writing code with data

clojure ♥ data

why such a close relationship?

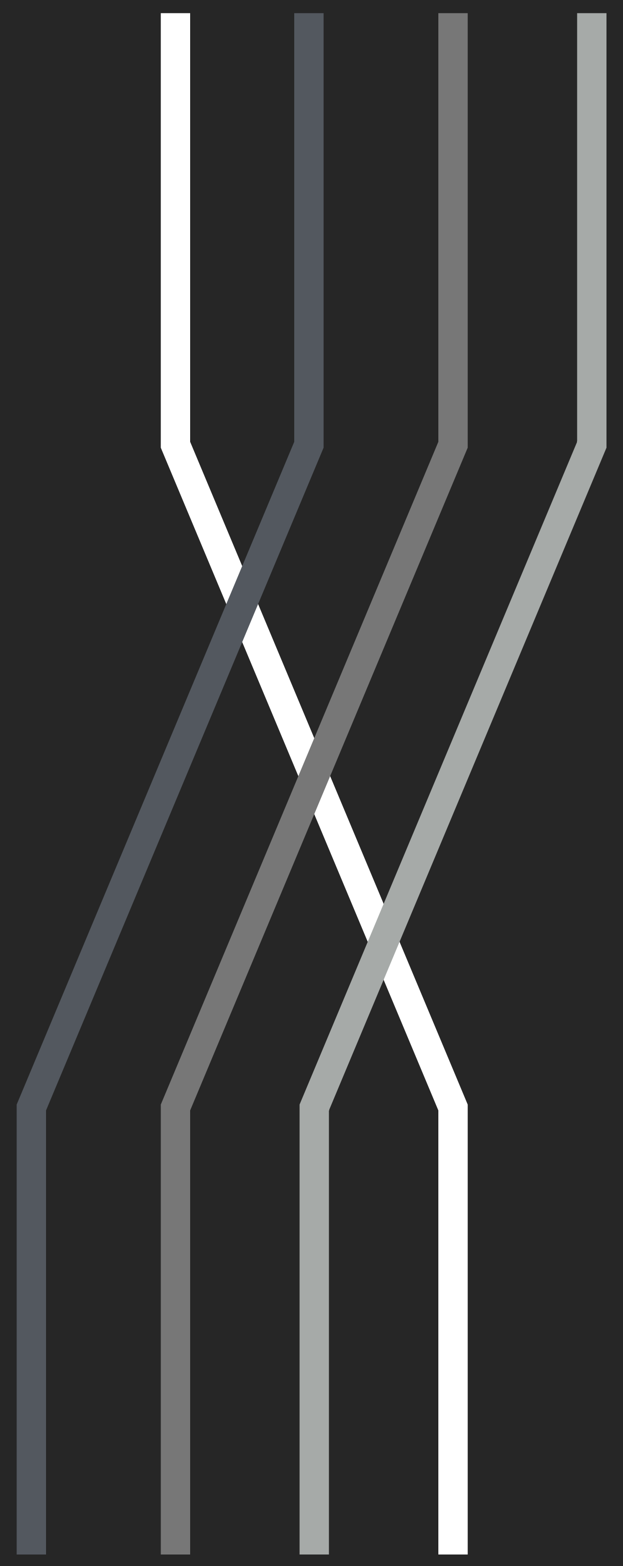
macros allow us to add new syntax through libraries

core.async	async programming
core.logic	logic programming
core.typed	static typing

but is that the **only** reason?

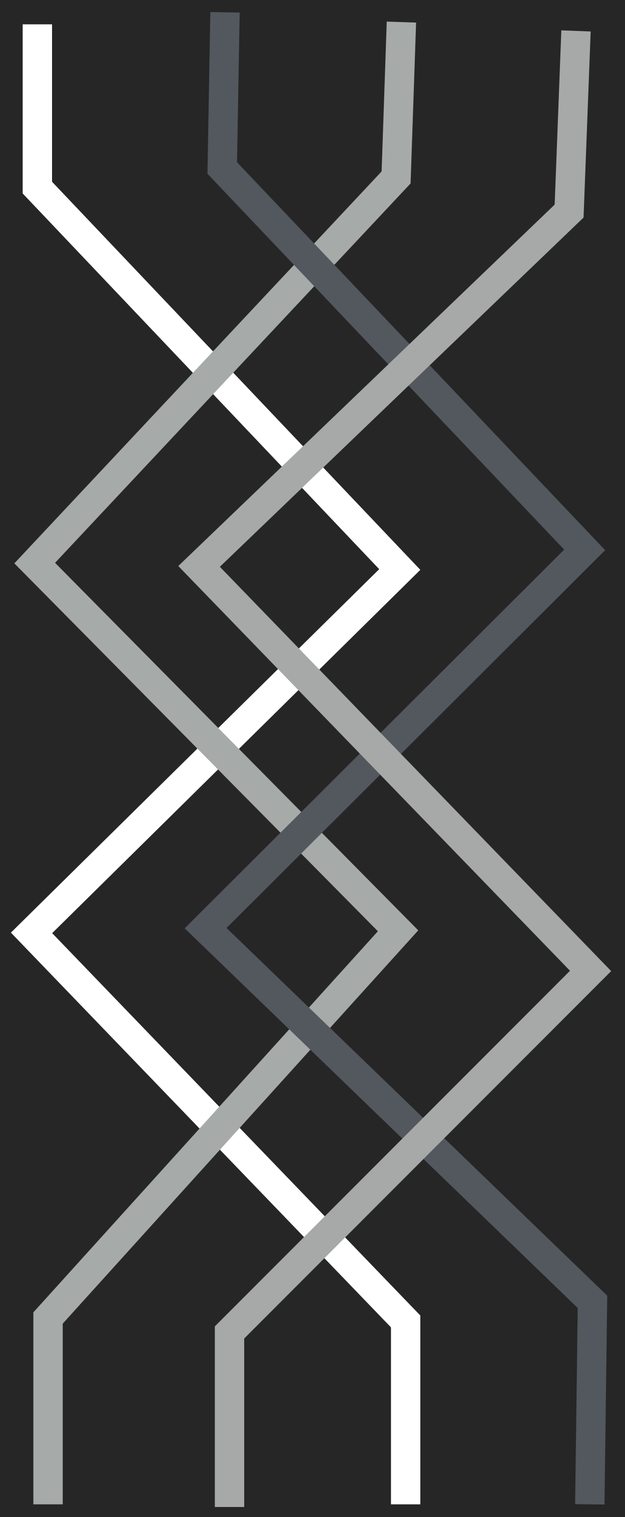
“Simple Made Easy”

simple

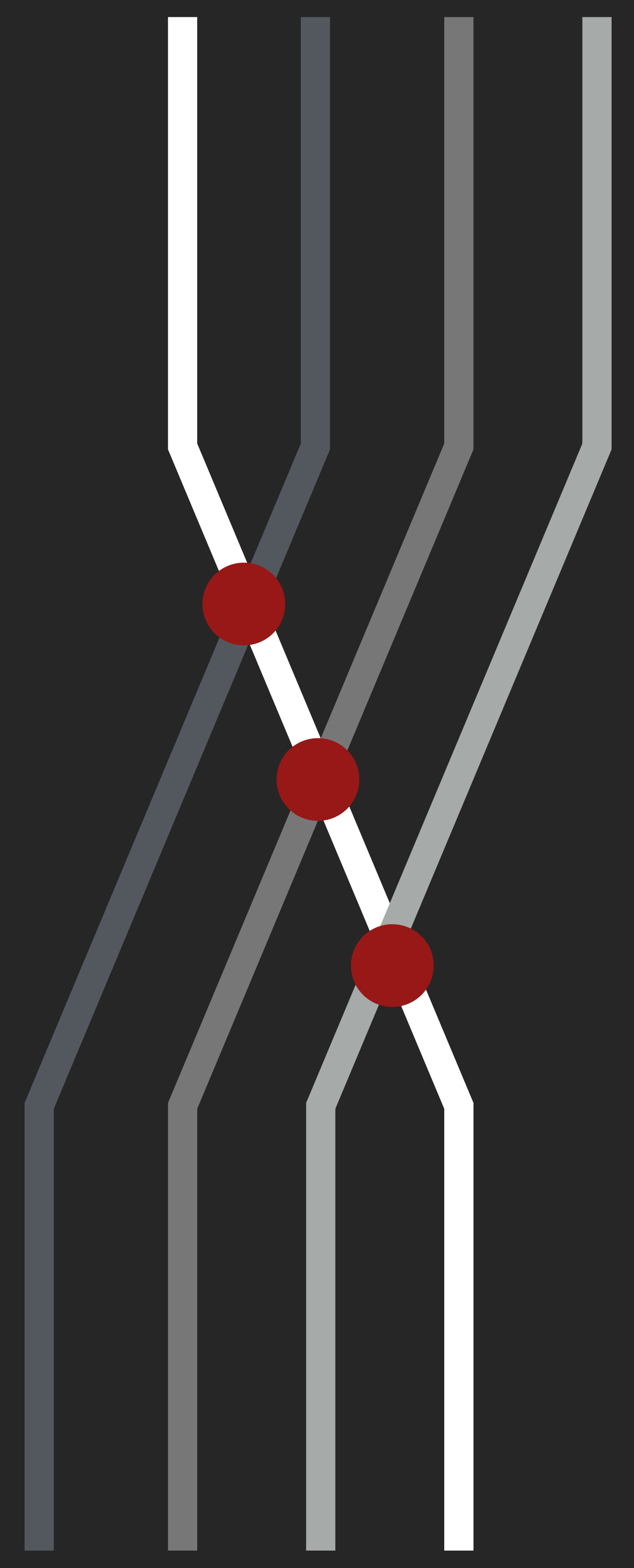


why?

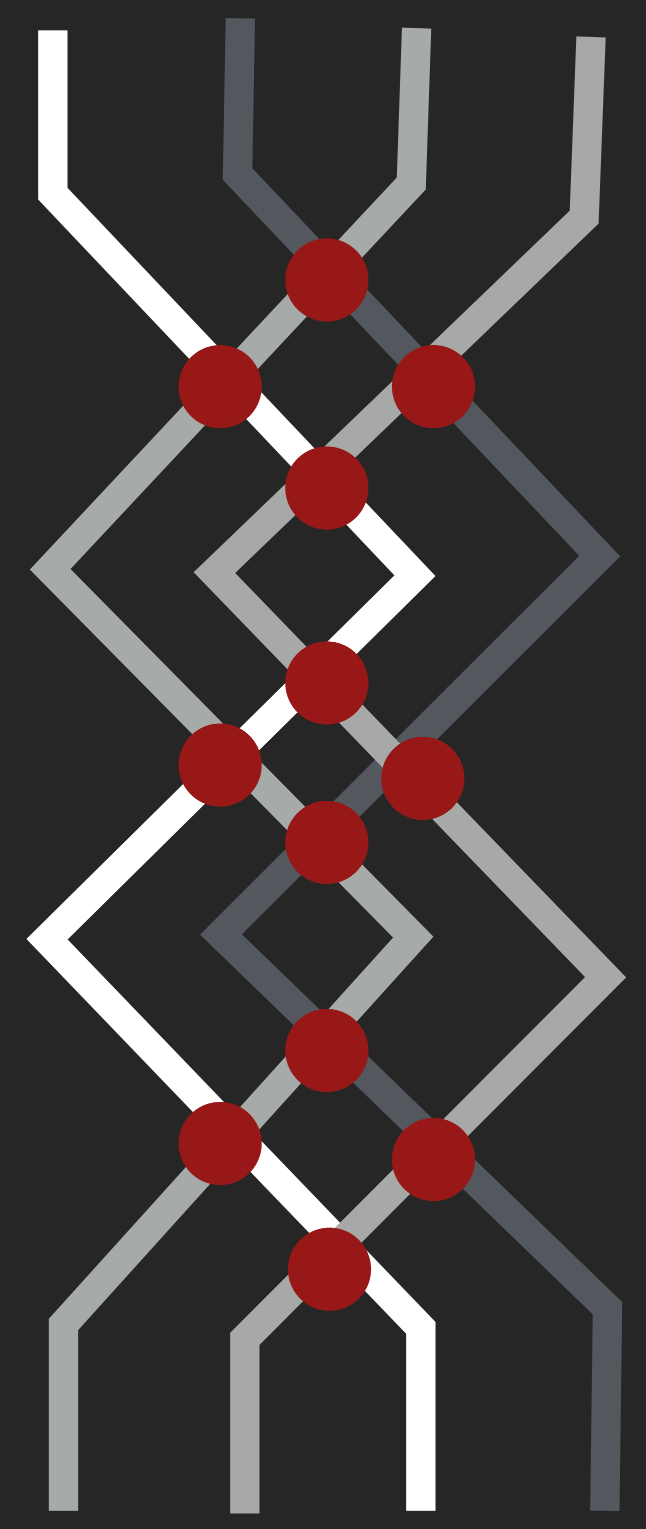
complex



simple



complex



complexity ≠ cardinality

connections

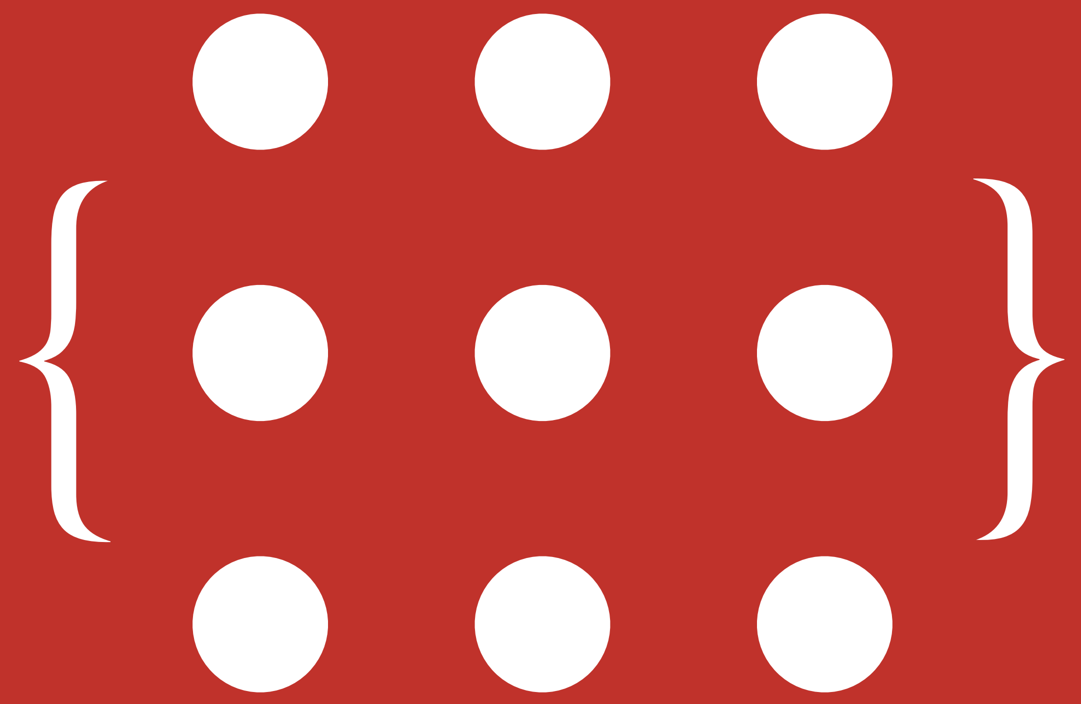
complexity = interlacing

coupling

how do we usually deal with **complexity**?

mutable state

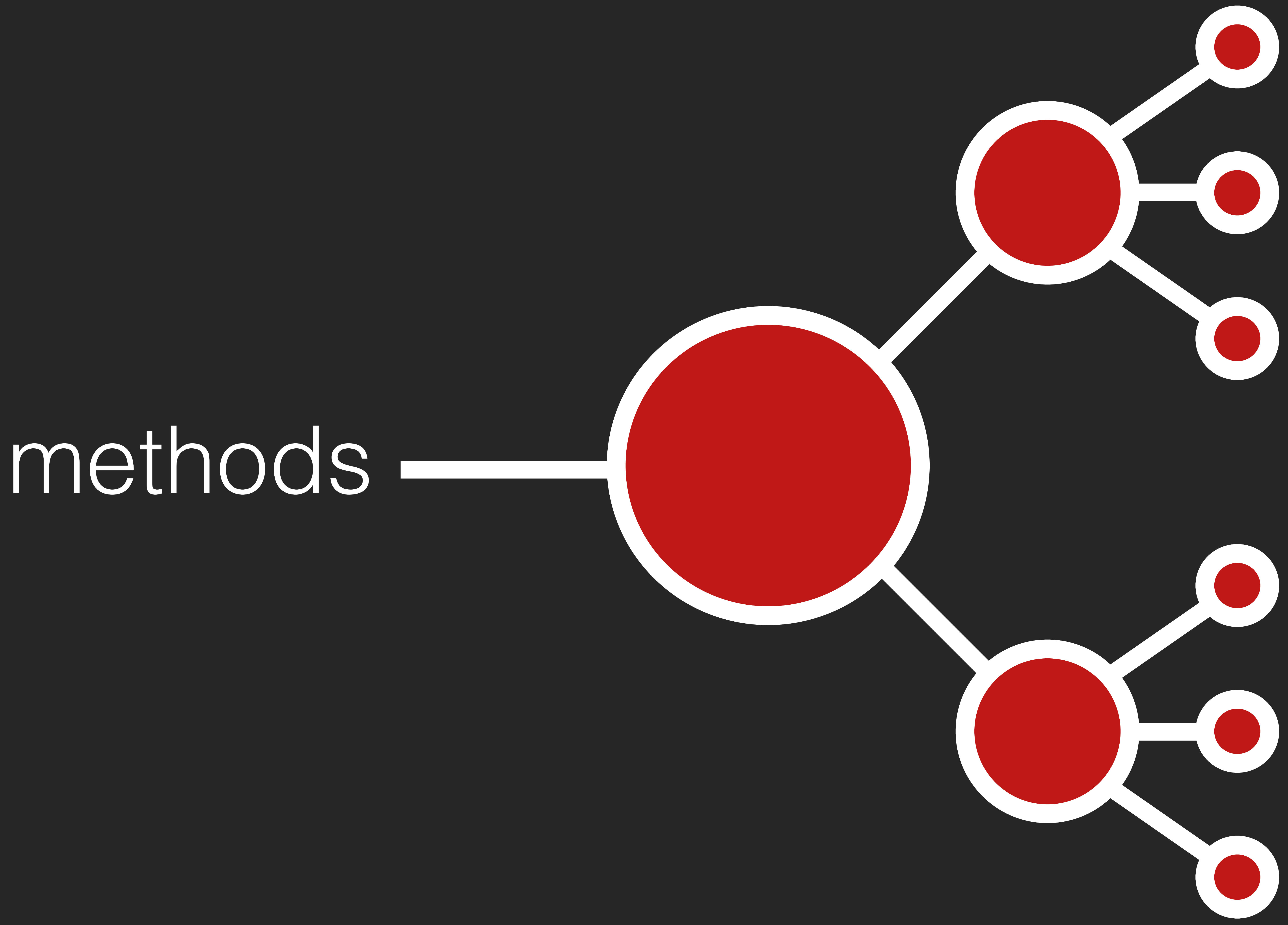
mutable state



object

encapsulates state

object



methods

encapsulation **isolates** complexity

defensive strategy

what would an offensive strategy look like?

can something have **zero** complexity?

immutable values

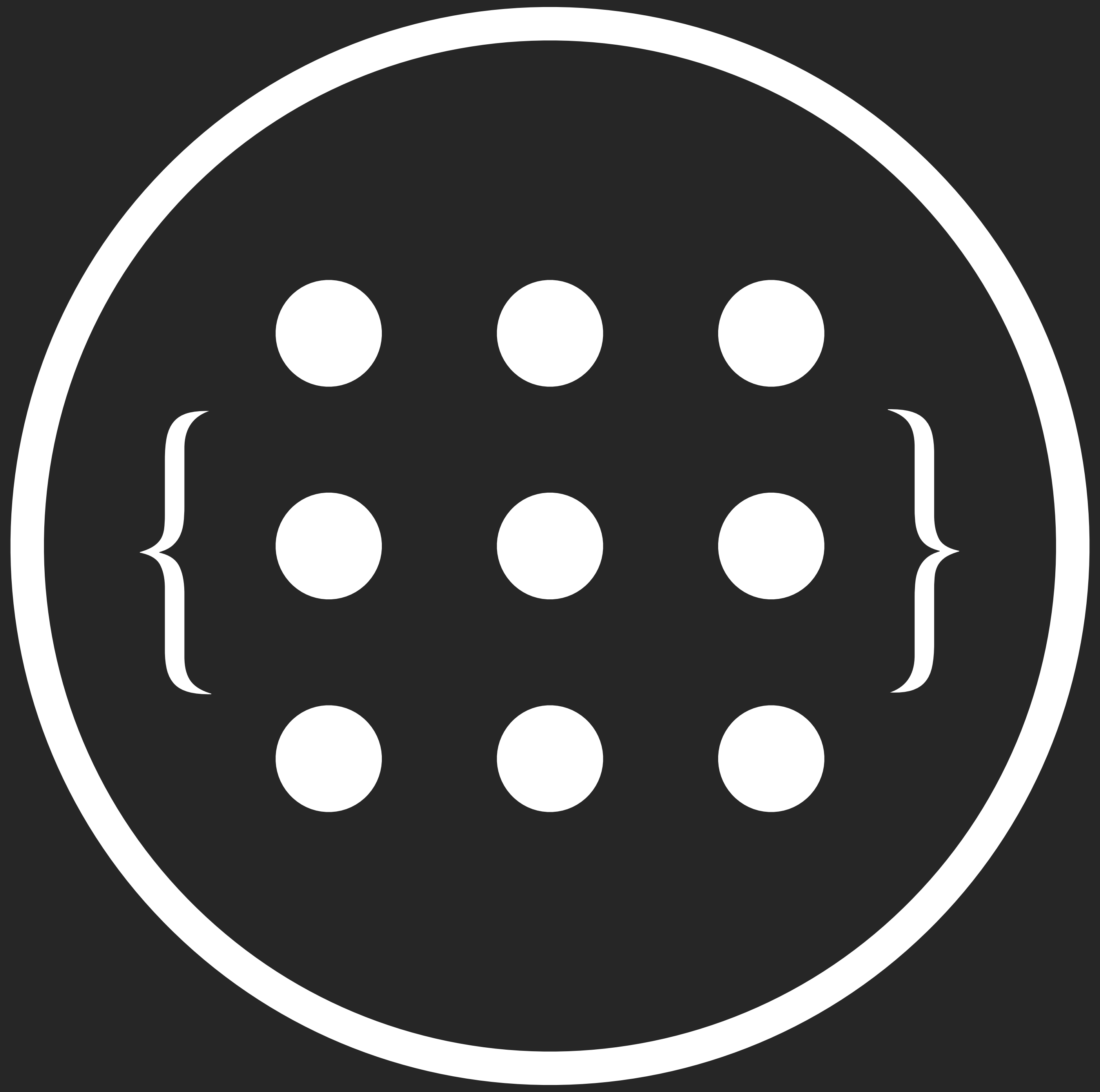
mutable state needed for

1. **performance**
2. communication across **threads**

object

do we need **encapsulation**?

walls are expensive



can we do that?

in **distributed** environments we often work with immutable values are there any **immediate** benefits?

free API

getters

setters

equality

lensing

transformations

transversal

merging

auditing

diffing

serialisation

deserialisation

concurrency

end

questions?