# Clojure

# Clojure 

An introduction

*"To build a brand new language and use lisp syntax on the JVM, you either gotta be a crazy person, or have some really cool ulterior motive. I met Rich and he's not a crazy person." — Neal Ford*

*(ulterior: hidden, secret)*

Who knows:

- Java
- Scala
- Any other functional language
- A LISP
- Clojure?
- git

# Clojure

- Made by Rich Hickey
- Released 2007
- Clojure.org

*“What makes Clojure unique, in my mind, is that it takes a hardline approach to both aesthetics and practicality, perhaps in recognition of the fact that those two need sets were never in opposition at all.” — Michael O. Church*

# Clojure 

*“Clojure is a dynamic, strongly typed, functional, high performance implementation of a Lisp on the JVM”*
*— Neal Ford*

# How does it feel?

# Like learning 

- Somewhat hard to learn
- Changed way of thinking
- Extremely efficient and fun
- Painful to go back

# Clojure


Lisp
Clojure

# Functional Programming

# Lisp

# Clojure

**Functional Programming**

**Lisp**

**Clojure**

**“Better” than others?**

**Functional Programming**

**Lisp**

**Clojure**

**Abstractions**

**“Better” than others?**

**Functional Programming**

**Lisp**

**Clojure**

- **State**
- **Concurrency**
- **Abstractions**
- **“Better” than others?**

**Functional Programming**

**Lisp**

**Clojure**

- **Data**
- **State**
- **Concurrency**
- **Abstractions**
- **“Better” than others?**

**Functional Programming**

**Lisp**

**Metalinguistic Abstraction (Marcos)**

**Clojure**

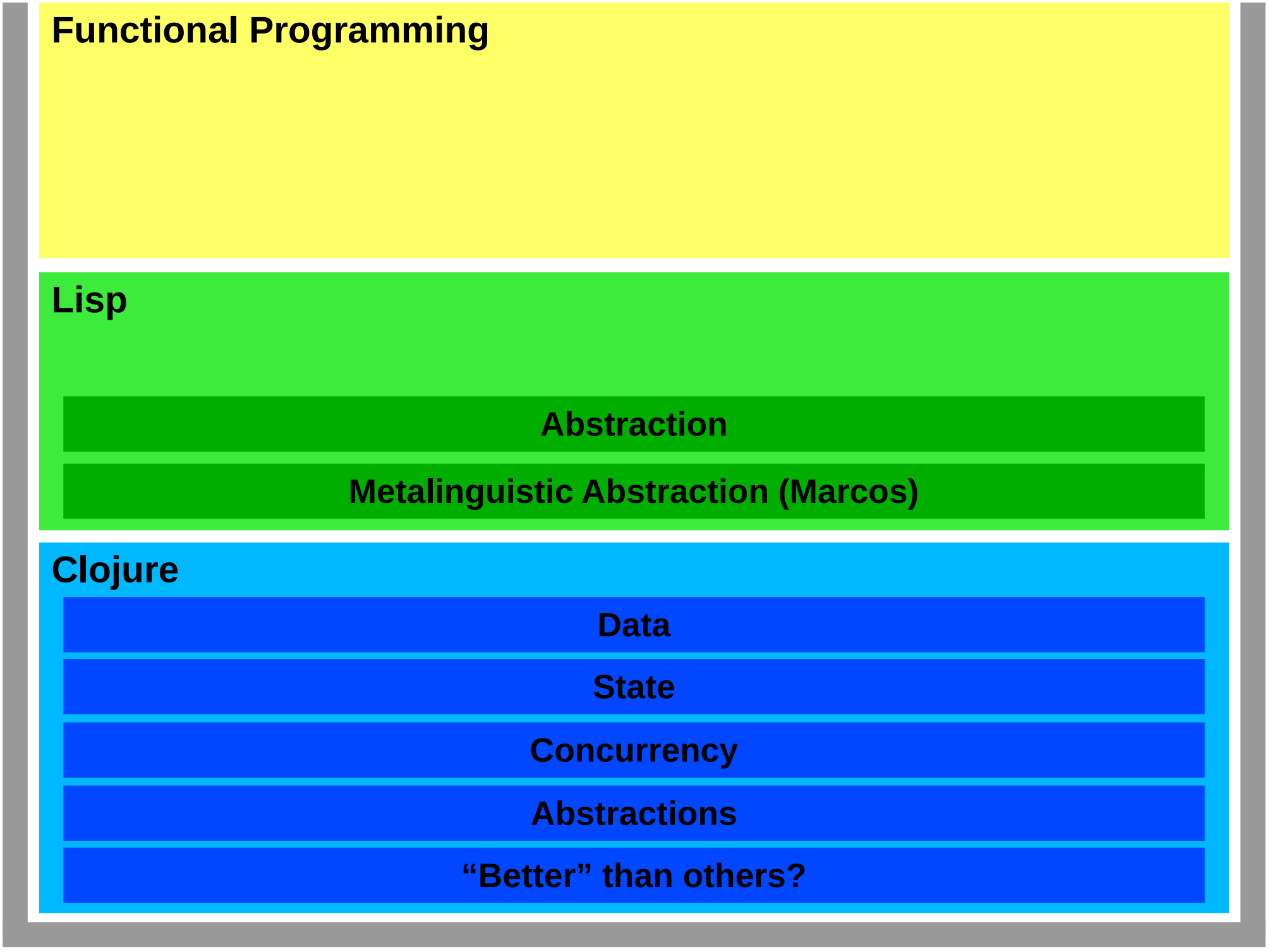
**Data**

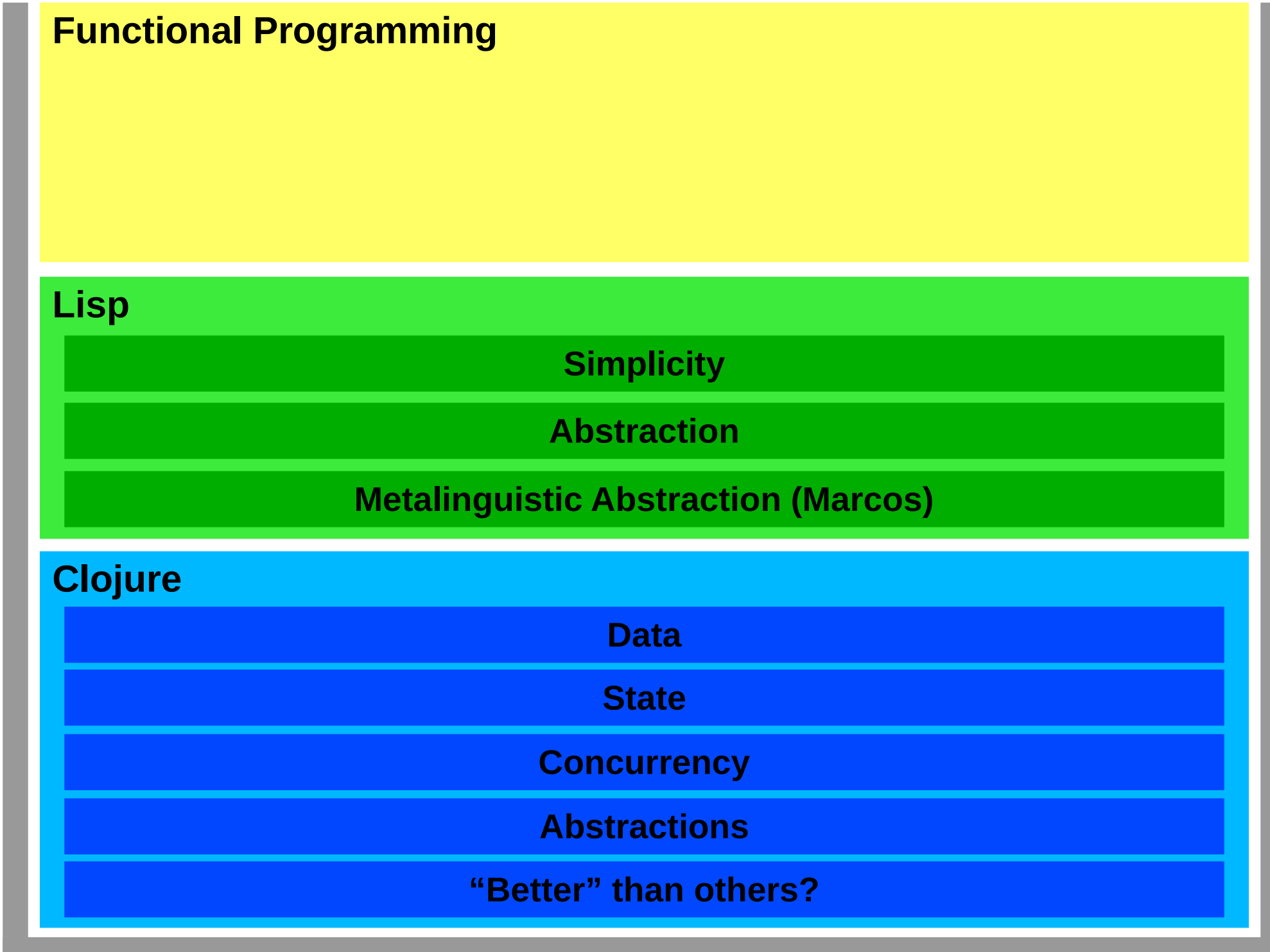
**State**

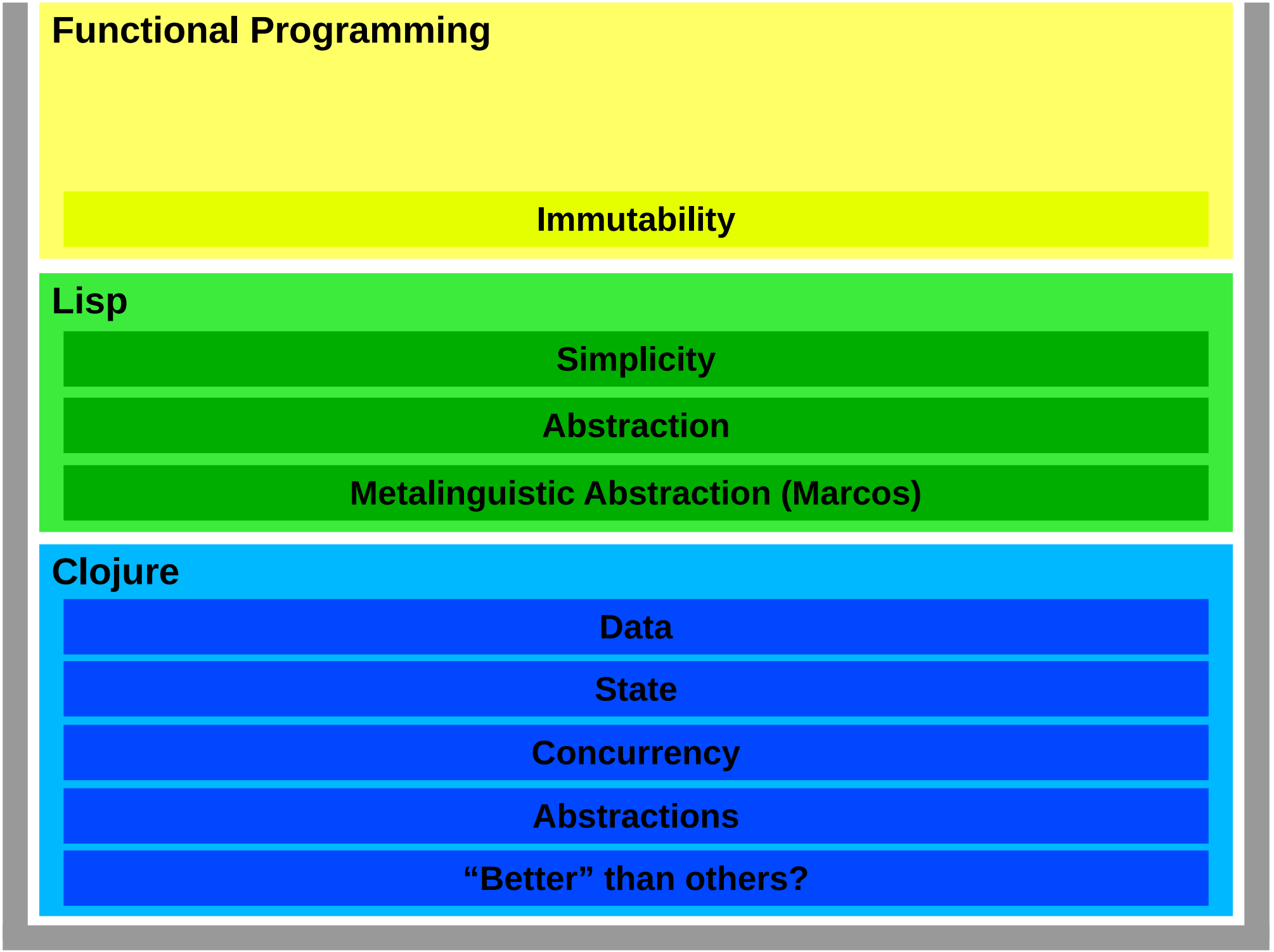
**Concurrency**

**Abstractions**

**“Better” than others?**


Functional Programming
Lisp
Abstraction
Metalinguistic Abstraction (Marcos)
Clojure
Data
State
Concurrency
Abstractions
“Better” than others?


Functional Programming
Lisp
Simplicity
Abstraction
Metalinguistic Abstraction (Marcos)
Clojure
Data
State
Concurrency
Abstractions
“Better” than others?


Functional Programming
Immutability
Lisp
Simplicity
Abstraction
Metalinguistic Abstraction (Marcos)
Clojure
Data
State
Concurrency
Abstractions
“Better” than others?

## Functional Programming

- Purity
- Immutability

## Lisp

- Simplicity
- Abstraction
- Metalinguistic Abstraction (Marcos)

## Clojure

- Data
- State
- Concurrency
- Abstractions
- “Better” than others?

## Functional Programming

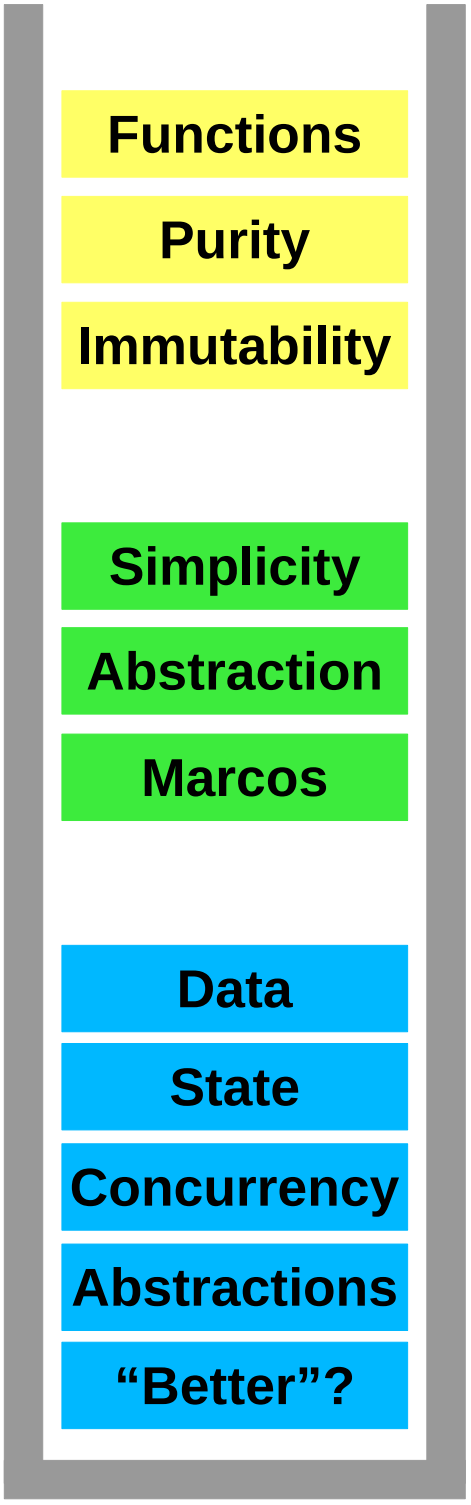
- Functions as First Class Citizens
- Purity
- Immutability

## Lisp

- Simplicity
- Abstraction
- Metalinguistic Abstraction (Marcos)

## Clojure

- Data
- State
- Concurrency
- Abstractions
- “Better” than others?


Functions
Purity
Immutability
Simplicity
Abstraction
Marcos
Data
State
Concurrency
Abstractions
“Better”?

# Functions – Syntax

| Syntax | Construct |
|---|---|
| (“a” “b” “c”) | List |
| [9 8 7] | Vector |
| Foo, + | Symbol |

# Functions – Syntax

- Syntax: (function argument*)

```
(+ 2 3)
```

# Functions – Syntax

- Syntax: (function argument*)

```
(+ 2 3)

(+ 2 3 4 5 6)
```

# Functions – Syntax

- Syntax: (function argument*)

```
(+ 2 3)

(+ 2 3 4 5 6)

(if (> a b) (print “a wins!”)
            (print “b wins!”))
```

# Functions – Syntax

- Syntax: (function argument*)

```
(+ 2 3)

(+ 2 3 4 5 6)

(if (> a b) (print “a wins!”)
            (print “b wins!”))

(. "foobar" toUpperCase)
(.toUpperCase "foobar")
```

# Functions – Syntax

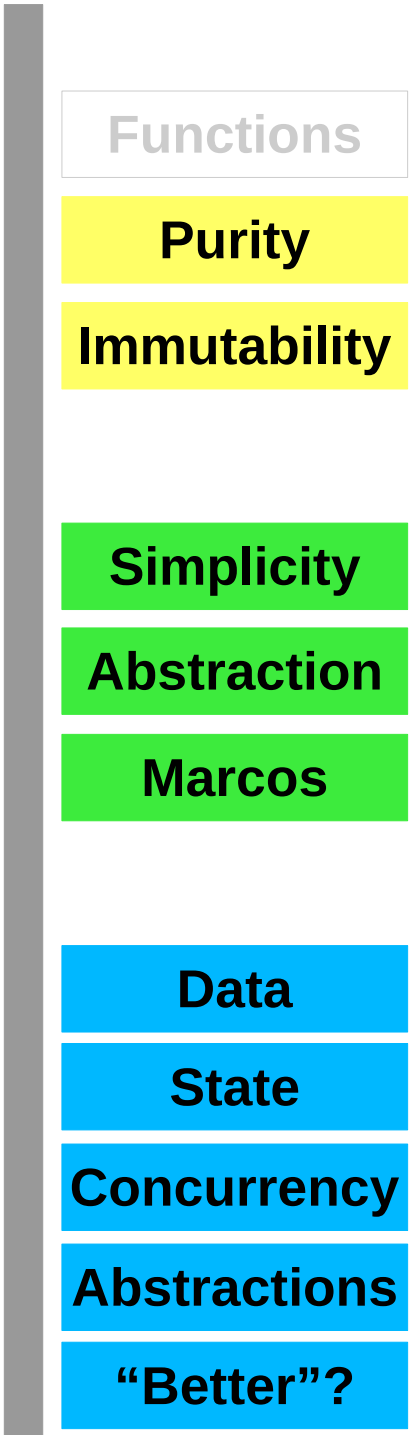
- Syntax: (function argument*)

```
(+ 2 3)

(+ 2 3 4 5 6)

(if (> a b) (print “a wins!”)
            (print “b wins!”))

(. "foobar" toUpperCase)

(defn fact [x]
   (if (<= x 1) 1 (* x (fact (- x 1)))))


(fact 5)
→ 120
```

# Functions as First Class Citizens

- Functions
  - Independent things
  - Pass around as arguments
  - Higher order functions

```
Lists.newArrayList(1, 2, 3, 4, 5).
stream().map(x -> x+1).
collect(Collectors.toList());
```

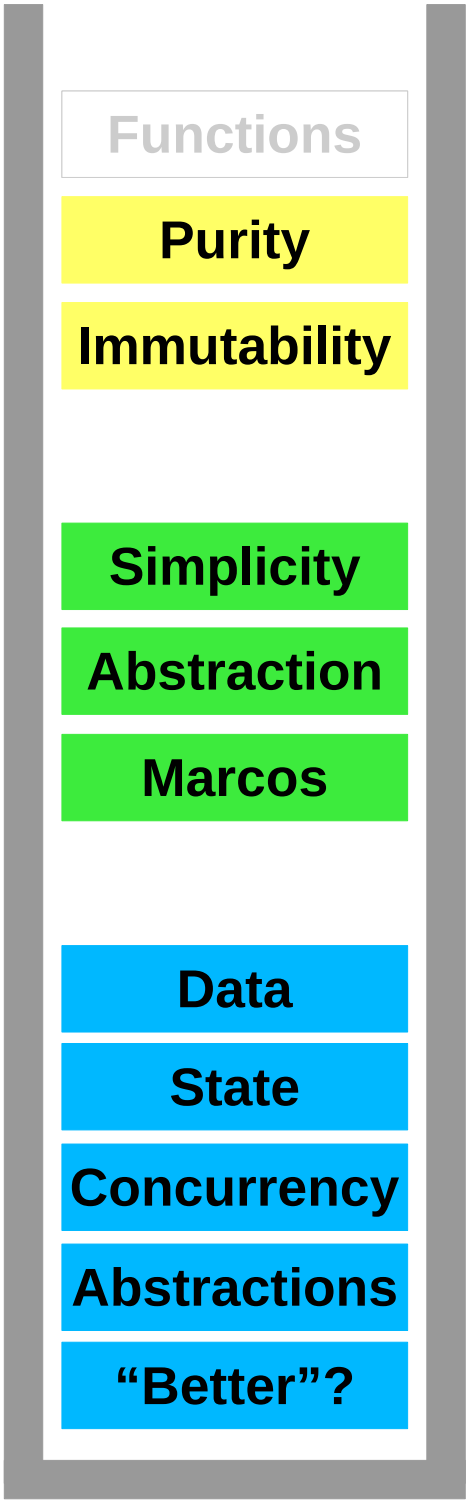


# Functions as First Class Citizens

- Functions
  - Independent things
  - Pass around as arguments
    - → Higher order functions

```
Lists.newArrayList(1, 2, 3, 4, 5)
.stream().map(x -> x+1)
.collect(Collectors.toList());
```

```
(map inc [1 2 3 4 5])
```



# Functions as First Class Citizens (2)

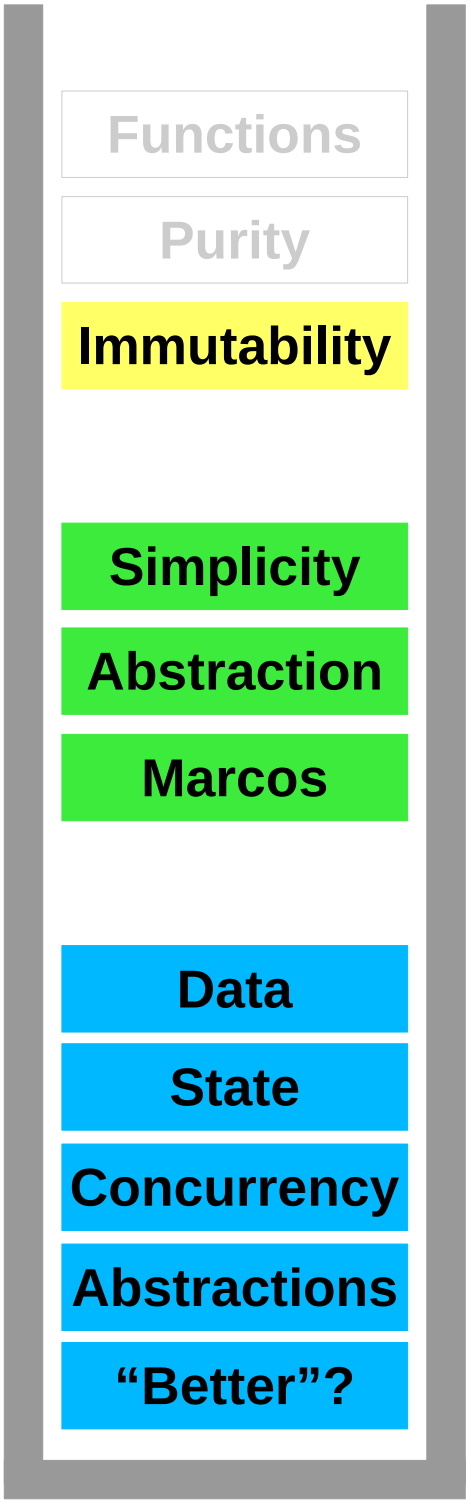
- Another kind of HOF: functions that return functions

```
(defn a-plus-abs-b [a b]
  ((if (> b 0) + -) a b))
```

- Benefits
  - Function reuse
  - Abstraction
  - More flexible

# Purity

- Pure functions
  - No side effects
  - Not dependent on state other than input
  - Same input → same result
- Benefits
  - Optimizations (inline results)
  - Parallel execution
  - Lazyness
  - Testable
  - Cachable (→ memoization)



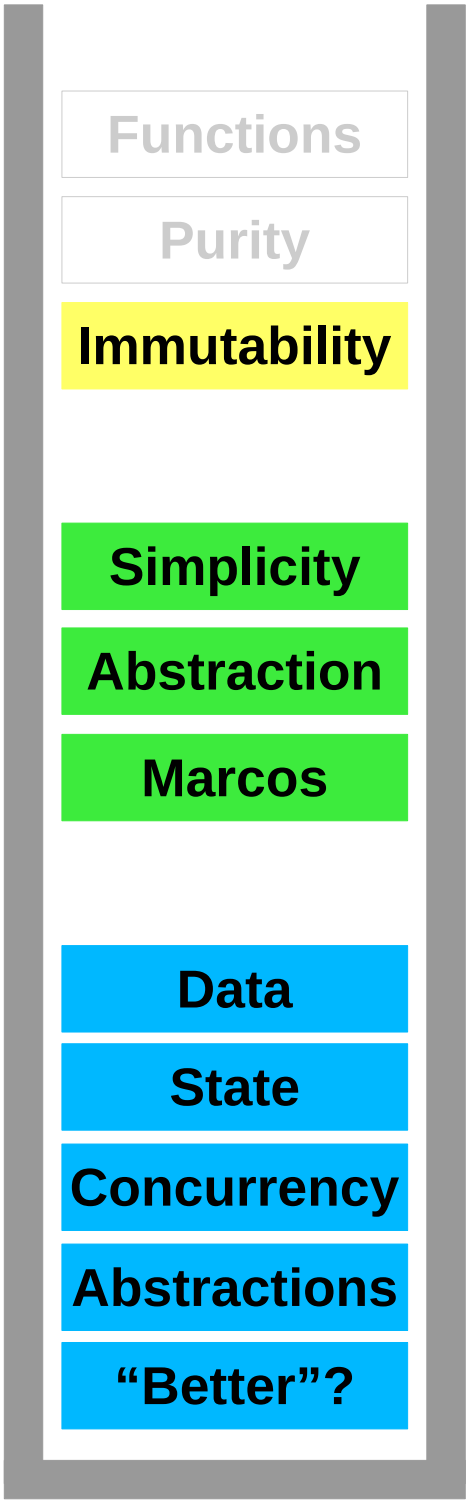
# Purity (2)

- Lazyness → infinite sequences

```
(defn fib [a b]
   (cons a (lazy-seq (fib b (+ b a)))))


(take 15 (fib 1 1))
→ (1 1 2 3 5 8 13 21 34 55 89 144 233
377 610)
```

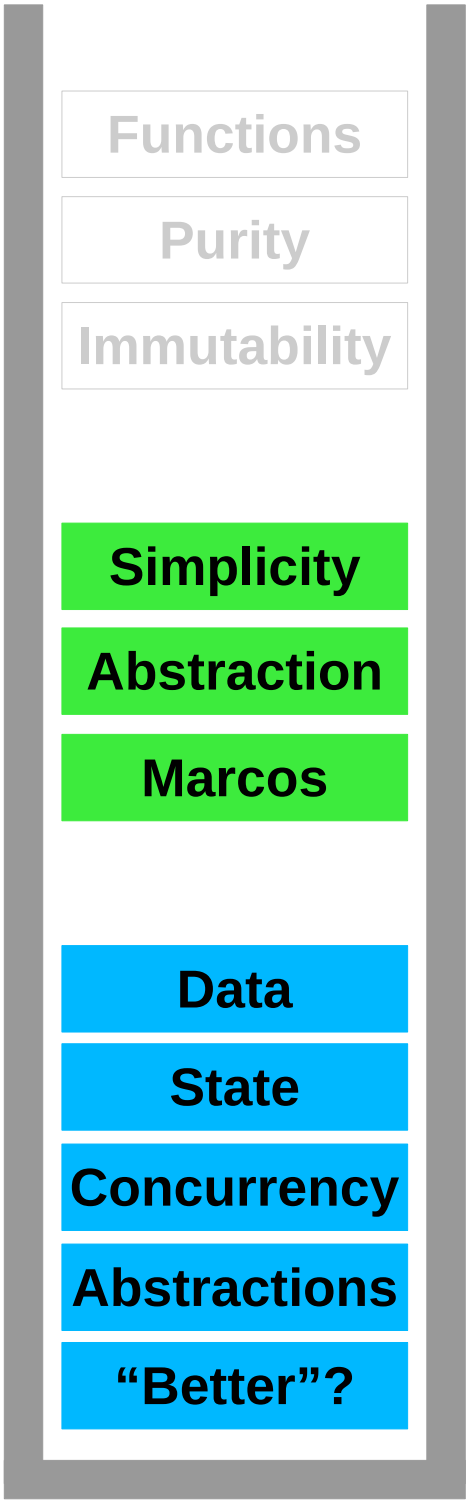
# Purity (3)

- Haskell:
  - Focus on purity
  - Type system enforces this.
    Side effects decoupled using monads
    (e.g. file IO)
- Clojure:
  - Likes purity
  - But no type system that “punishes”
    you for impure functions



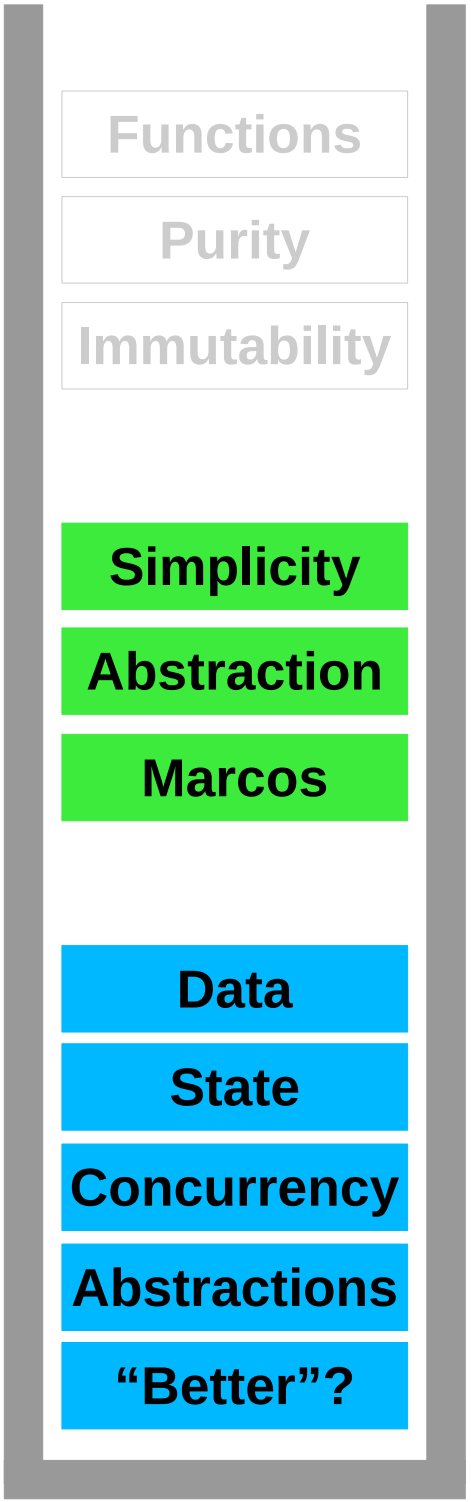
# Immutability

- Important in all FP languages (Scala, Haskell, Clojure, ...)
- Mutability destroys purity
- In Java
  - some things are immutable
  - most are not



# Immutability (2)

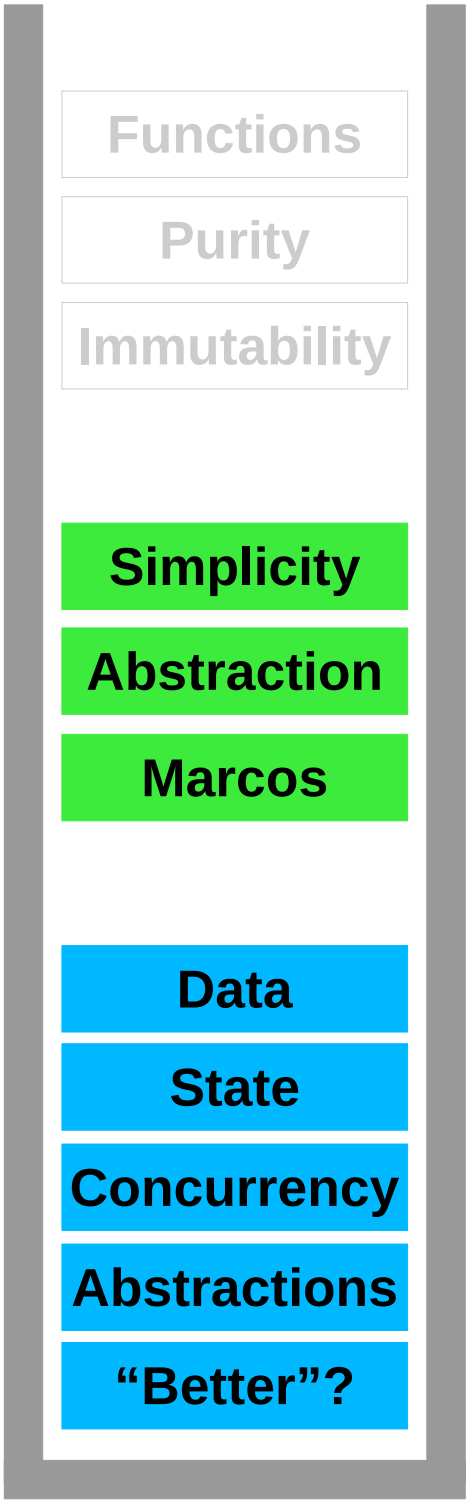
- In Clojure everything is immutable per default
  - Functions, values
  - Data structures
  - You can have mutable objects (i.e. Java interop)
- Application-State?
  - Designated locations for state
  - Passed in as an argument



# Functional Programming

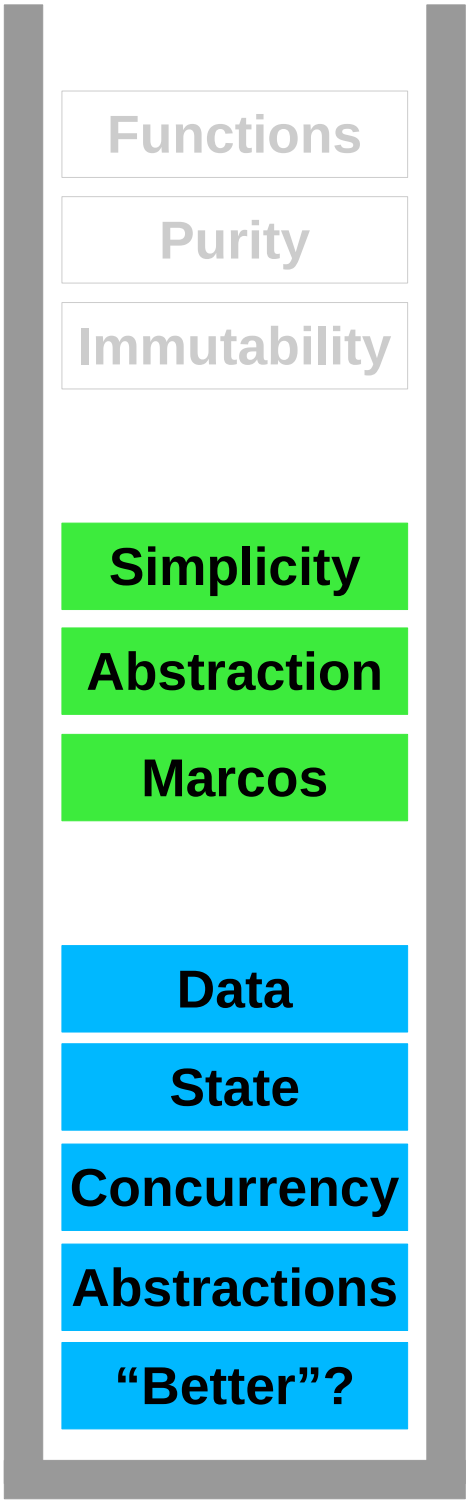
- Functions as first class citizens
- Pure functions
- Immutability

*"OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts." — Michael Feathers*



# Lisp

*“The two most sophisticated projects we have going on in the world right now, are both Clojure projects, on purpose. Because we think this is the only way these projects can be done.”*
*— Neal Ford*



# Simplicity (1)

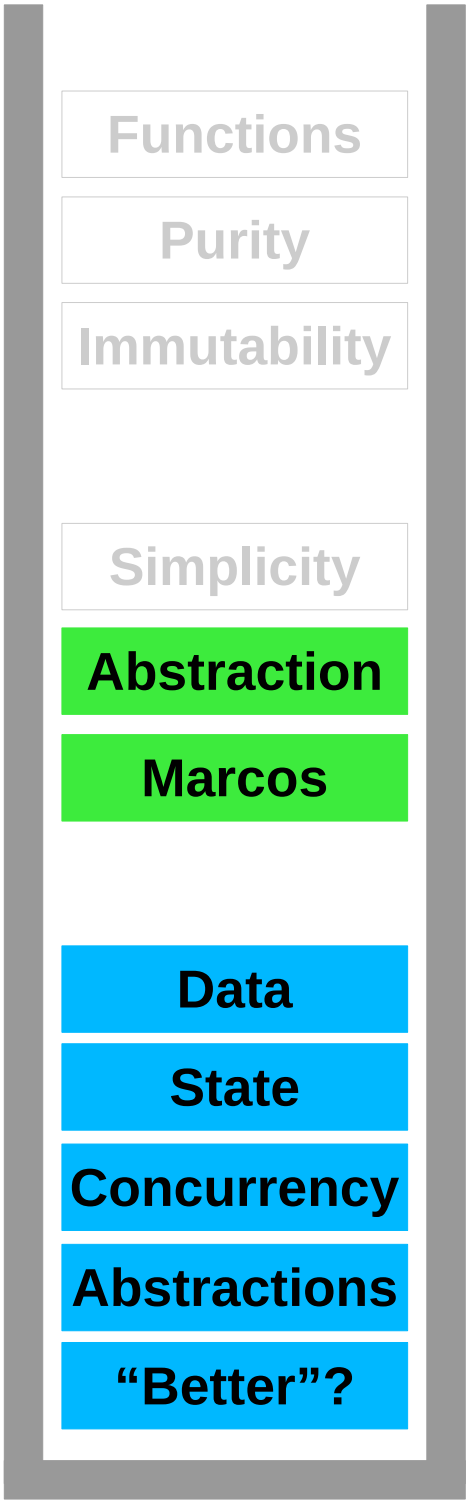
- Simple means unentangled, decoupled, separate → Simple Made Easy

*“Simplicity is hard work. But in the long run the person who has a simpler system is gonna wipe the plate with you, because she'll be able to change things when you are struggling to push elephants around.” — Rich Hickey*

*“When I learned Lisp later in my career, I saw that you could build the same systems with much, much simpler stuff” — Rich Hickey*

# Simplicity (2)

- In Java we're not nearly as simple as we could be:
  - We're not used to
  - Java's syntax does not help
  - Mutablity fosters complexity
- Clojure helps to write simple programs:
  - Immutability and pure functions
  - Basic data structures for data
  - Abstractions for organizing state



# Abstraction

- Separate abstractions for things we entangle with objects
  - Data (→ Basic data structures)
  - Methods (→ Functions)
  - State/Mutablitiy (→ Reference types)
  - Polymorphism (→ Protocols)
  - Inheritance (→ Extend function)
  - Namespaces (→ Namespaces)
- OO makes it easy to couple these things

# Metalinguistic Abstraction (Macros) (1)

- Lisps are the only languages that have such powerful macros

*“Lisp is a programmable programming language” — John Foderaro*

*“Lisp is the red pill.” — John Fraser, on comp.lang.lisp*

# Metalinguistic Abstraction (Macros) (2)

- Special Forms ^= syntax rules baked into the language
- Java → ???
  - Specialized syntax for each purpose
  - Not extensible (helps with learning and readability)
- Lisp
  - A small set of special forms (Scheme 6, Clojure 12)
  - Extensible through macros

# Metalinguistic Abstraction (Macros) (2)

- Special Forms ^= syntax rules baked into the language
- Java → ~140
  - Specialized syntax for each purpose
  - Not extensible (helps with learning and readability)
- Lisp
  - A small set of special forms (Scheme 6, Clojure 12)
  - Extensible through macros

# Metalinguistic Abstraction (Macros) (3)

- Code is also data (lists)
  - You can transform lists programmatically
  - Thus you can create or change code
- Features
  - Compile-time transformations of code
  - Same syntax for macros as for code (homoiconicity)

*“In Clojure meta-programming is just programming. Because Clojure programs are Clojure data structures.” — Neal Ford*

# Macros

- Avoiding wrappers:

```
@Override
public List<ItemPmo> getItems() {
   List<ItemPmo> rows = new
           ArrayList<ItemPmo>();
   for (IItem p:
       getThing().getItemListe()) {
      rows.add(new ItemPmo(p));
   }
   return rows;
}
```

# Metalinguistic Abstraction (Macros) (4)

- Examples
  - Java's foreach loop
  - Abstracting exception handling
- Real macros
  - when
  - “..”

# when

```
(when x (println “x is true”))
```

Functions

Purity

Immutability

Simplicity

Abstraction

Marcos

**Data**

**State**

**Concurrency**

**Abstractions**

**“Better”?**

# when

```
(when x (println “x is true”))
```

```
(defmacro when [test & body]
  (list 'if test
        (cons 'do body)))
```

Functions

Purity

Immutability

Simplicity

Abstraction

Marcos

**Data**

**State**

**Concurrency**

**Abstractions**

**“Better”?**

# when

```
(when x (println “x is true”))
```

```
(defmacro when [test & body]
  (list 'if test
        (cons 'do body)))
```

→ Macroexpansion:

```
(if x
  (do (println “x is true”)))
```

Functions

Purity

Immutability

Simplicity

Abstraction

Marcos

**Data**

**State**

**Concurrency**

**Abstractions**

**“Better”?**

..

```
session.getCurrentUser()
.getAddress().getStreet()
```

```
(. (. (. session getCurrentUser)
getAddress) getStreet)
```

Functions

Purity

Immutability

Simplicity

Abstraction

Marcos

**Data**

**State**

**Concurrency**

**Abstractions**

**“Better”?**

# ..

```
session.getCurrentUser()
.getAddress().getStreet()
```

```
(. (. (. session getCurrentUser)
getAddress) getStreet)
```

→ Macro “..”

```
(.. session getCurrentUser
getAddress getStreet)
```

Functions

Purity

Immutability

Simplicity

Abstraction

Marcos

**Data**

**State**

**Concurrency**

**Abstractions**

**“Better”?**

# .. (2)

```
(defmacro ..
  ([x form] `(. ~x ~form))
  ([x form & more] `(.. (. ~x
~form) ~@more)))
```

# Metalinguistic Abstraction (Macros) (5)

- Benefits
  - Clean up your code
  - Create your own special syntax

*”Design patterns do not exist in the Lisp and Clojure worlds.” — Neal Ford*

*“When I see patterns in my programs, I consider it a sign of trouble – often that I'm generating by hand the expansions of some macro that I need to write." — Paul Graham*

# Metalinguistic Abstraction (Macros) (6)

- Benefits
  - Create DSLs
  - Integrate own constructs

  → change the language to your needs

*”Because it's the hardest of all projects where you need the ability to mold your language towards your problem, not vice versa.”*
*— Neal Ford*

# Lisp

- Simplicity
- Use suitable abstractions
- Molding the syntax (Marcos)

*“Programming in Lisp is like playing with the primordial forces of the universe. It feels like lightning between your fingertips. No other language even feels close.” — Glenn Ehrlich*

# Data - Values

| type | example | Java equivalent |
|---|---|---|
| string | “foo” | String |
| character | \f | Character |
| regex | #”fo*” | Pattern |
| a.p. integer | 42 | Integer/Long<br>BigInteger |
| double | 3.14159 | Double |
| a.p. double | 3.14159M | BigDecimal |
| boolean | true | Boolean |
| nil | nil | null |
| symbol | foo, + | - |
| keyword | :foo | - |

# Data Orientation

*“Objects are a terrible approach for data”*
*— Rich Hickey*

- Objects
  - Mutable by default
  - Methods/interfaces are mini languages
- Clojure's stance
  - Data never changes!
  - Use basic data structures to represent data, nest them
  - Use standard functions to process it

→ Data orientation

# Data Structures

| Data structure | properties | example |
|---|---|---|
| list | Singly-linked,<br>Insert at front | (1 2 3) |
| vector | Indexed,<br>Insert at rear | [“a” “b” “c”] |
| map | Key/value | {:a 100<br>:b 4711} |
| set | Key | #{:a “b” 3} |

- Usable through seq abstraction (conj)
- Heterogeneous
- “Persistent”

# Persistent Data Structures

- Immutable

# Persistent Data Structures

- Immutable
- “Change” creates new version, old versions stay available

# Persistent Data Structures

- Immutable
- “Change” creates new version, old versions stay available
- However not copy on write

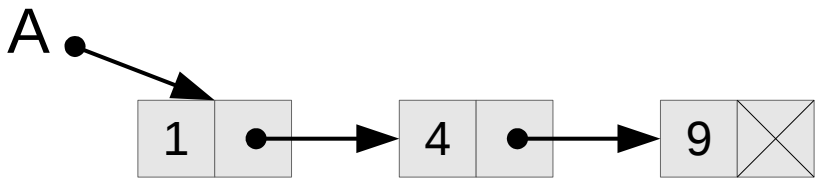
# Persistent Data Structures

- Immutable
- “Change” creates new version, old versions stay available
- However not copy on write
- Similar performance guarantees to mutable data structures

# Persistent Data Structures

- Immutable
- “Change” creates new version, old versions stay available
- However not copy on write
- Similar performance guarantees to mutable data structures

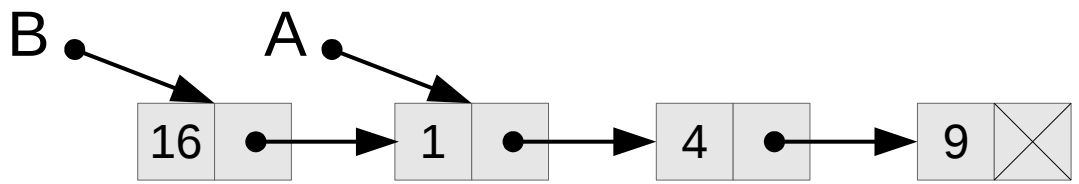
→ can't be full copies



# Persistent Data Structures

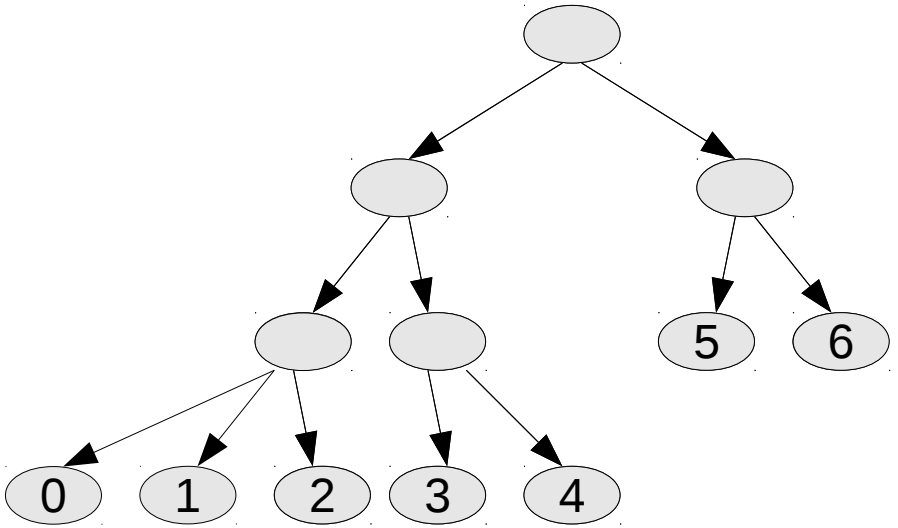
- Immutable
- “Change” creates new version, old versions stay available
- However not copy on write
- Similar performance guarantees to mutable data structures

→ can't be full copies



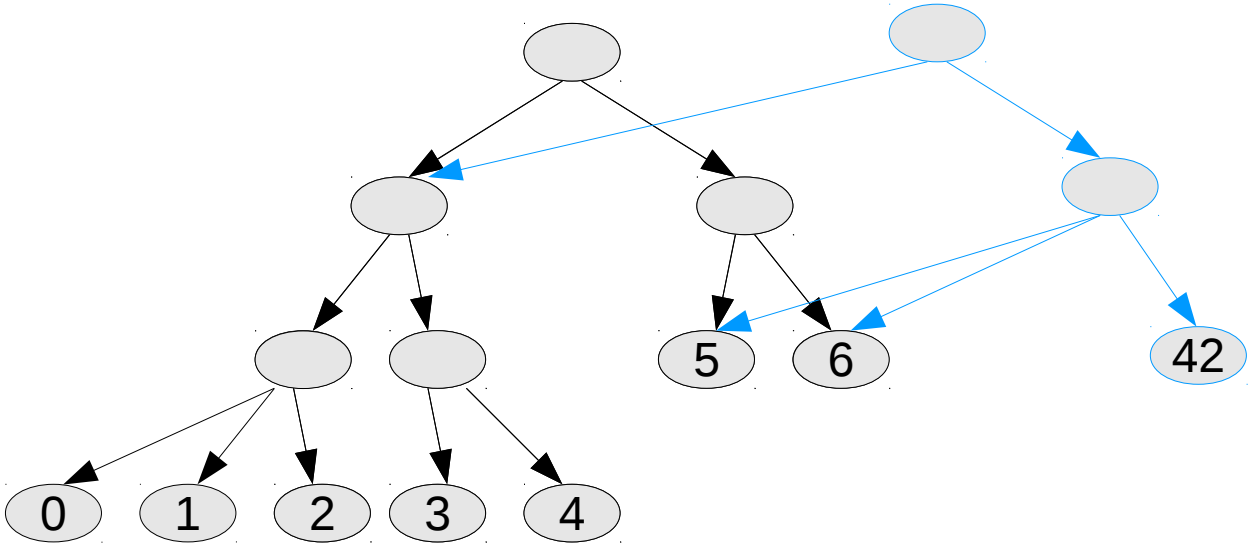
# Persistent Data Structures

- List-based (linked list)
- Tree-based (Map, Set, Vector)



# Persistent Data Structures

- List-based (linked list)
- Tree-based (Map, Set, Vector)
- Structural sharing



# Persistent Data Structures

- Immutability comes at a cost
  - Requires more memory
  - Factor 2-4 slower than mutable data structures
- Payoff:
  - Everything is immutable
  - Worry free concurrency

*"When you are writing a program that uses persistent data structures, you'll be able to sleep at night. You're gonna be happier. Your life is gonna be better. Because there's a huge quantity of things you will no longer have to worry about." — Rich Hickey*

# Persistent Data Structures

- Incidental complexity *because* of mutability

```
private void updateItemCopy() {
   this.itemCopy = new
ArrayList<>(getContainerPmo().getItems(
));}

public void updateFromPmo() {
   if (itemCopy.size() !=
getContainerPmo().getItems().size()) {
…
}}
```

# State

- Epochal Time model

  → Are we there yet?

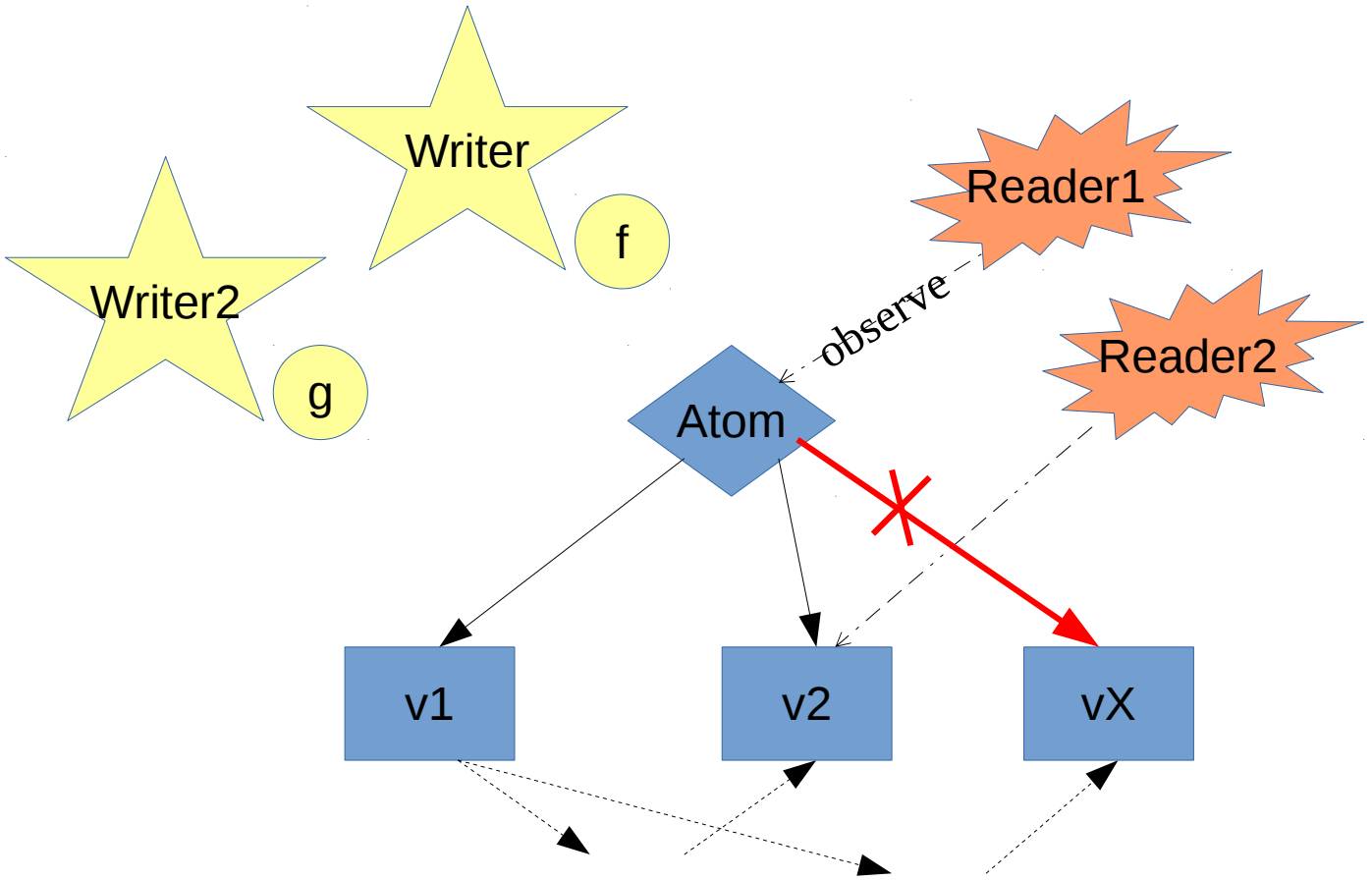
*"The object-oriented model has gotten time wrong. The problem is that there is no standard time management." — Rich Hickey*

# State (2)

- State = value of a “thing” at a point in time
- Reference types (e.g. atom)
  - Represent a “thing” (identity)
  - Point to immutable values
  - Have semantics for value transitions
- Writing:
  - Applying a pure function to the DS
  - The result is the new value the reference points to
- **Reading is massively parallel**

# State (3)

- Atom: Compare And Swap (CAS) semantics



# State (4)

- Usage

```
(def counter (atom 1))
(swap! counter inc)  ;; write op
(deref counter)   ;; read op
→ 2
(swap! counter inc)
@counter
→ 3
```

# Concurrency

- **Atom**: Compare and swap
- **Agent**: like actors, but local and observable
- **Ref:** Software Transactional Management (STM)
  - E.g. move money between accounts
  - All functions succeed, or none
  - Known from databases
  - Clojure provides ACI (no D)

*“Clojure does for concurrency what Java did for memory management.” — Neal Ford*

# Concurrency – STM

```
(def acc1 (ref {:user "Alice"
                :balance 1000}))
(def acc2 (ref {:user "Bob"
                :balance 222}))


(dosync (alter acc1
  (fn [acc] (update acc :balance – 50)))
        (alter acc2
  (fn [acc] (update acc :balance + 50))))


@acc1
→ {:user "Alice" :balance 950}

@acc2
→ {:user "Bob" :balance 272}
```

# Concurrency (2)

- Multiple (inconspicuous?) ideas come together:
  - Values/DS are immutable
  - Efficient creation (persistent DS)
  - Constructs with specialized semantics
  - Macros to integrate them into the language

# Abstractions

- Seq
  - data structures
  - fs
  - Regex
  - ...
- Concurrency
  - Refs
  - Atoms
  - Agents
  - Channels (CSP)

# Abstractions (2)

- Modeling
  - Protocols (think interface)
    - Polymorphism (single dispatch)
    - Extend existing classes (think mixins)
  - Multimethods (mutliple dispatch)
  - Types, Records (combine data & functions)
  - Extend macro
    - mechanism for extending types
    - More general than inheritance (e.g. traits, mixins, multiple inheritance, or your own mechanisms)

# “Better” than others?

- Other Lisps
  - Better integrated data structures
  - Additional persistent data structures
- Most other languages on the JVM
  - Simplicity + Data orientation
  - Functional programming
  - Marcos
  - Concurrency support / STM + CSP
  - Java interoperability

# “Better” than others? (2)

- Ecosystem
  - REPL development flow
  - Leinigen (build tool)
- ClojureScript
  - Clojure to JavaScript compiler
- Datomic: database
  - Functional
  - Immutable (keeps historical data)

# Learning Clojure

- Upsides
  - Learned a lot
  - **Clojure changed my way of thinking**
- Downsides
  - Pain to write Java afterwards
  - Starting to see
    - System design mistakes everywhere
    - Need for macros everywhere

*"Lisp is worth learning for the profound enlightenment experience you will have when you finally get it;*
*that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot."*
*— Eric Raymond, "How to Become a Hacker"*

# Using Clojure

- Amazing mix of useful features
- New concepts
- Simple constructs
- Closer to the language

# Using Clojure (2)

- Undecided / uncertain:
  - No type system (optional via core.typed)
    - No type checking
    - But also: more flexible
  - Data orientation
    - Works well for small systems
    - Does it scale?

# Using Clojure (3)

- Downsides
  - Clojure is not for everyone
  - Introducing Clojure into my workplace is hard (“our customers can't code clojure”)

# Conclusion

- Clojure is a language for experts
- What should we use it for? - What it's good at!
  - Highly concurrent environments (the server-side of applications)
  - Hard problems (projects where you need highly expressive syntax, DSLs)
  - Everything other functional languages are good at (domains that possess the closure property, i.e. hierarchical data)
- IMHO everywhere else

*”People ask me what projects I would choose Clojure for.*
*The answer is: the nastiest, hardest, toughest kinds of projects possible.”*
*— Neal Ford*

# Links

- Rich Hickey's Talks (most notably: Simplicity, Clojure, Time)
  *“Every time I watch one of his talks, I feel like someone has gone in and organized my brain.” — Daniel Higginbotham*
- SICP
- Why LISP?
  *“Lisp is not merely a different notation, it's a fundamentally different way of thinking about what programming is.” - Ron Garret*
- Curious Clojurist, Neal's Master Plan
  *”With Clojure you bring a gun to knife-fight.” — Neal Ford*
- Programming with hand tools
  *”The tools that you use shape how you look at the world.” — Tim Ewald*
- Books:
  - Programming Clojure, Pragmatic Bookshelf (Halloway, Bedra)
  - Clojure Programming, O'Reilly  (Emerick, Carper, Grand)
- My Blog

# Getting started with Clojure

- Clojure.org
  - Getting started (links to IDEs, books etc.)
  - Clojure CheatSheet (Library functions overview, similar to JavaDoc)
- IDEs:
  - Lighttable (free, written in ClojureScript)
  - Counterclockwise: Eclipse Plugin, free
  - Cursive: IntelliJ Plugin
  - Emacs with Cider Plugin (free)
- 4 Clojure (programming puzzles)