The value of Clojure's identity

A walk-through its main features and what makes it special

Lisp

Functional Programming

Why Clojure?

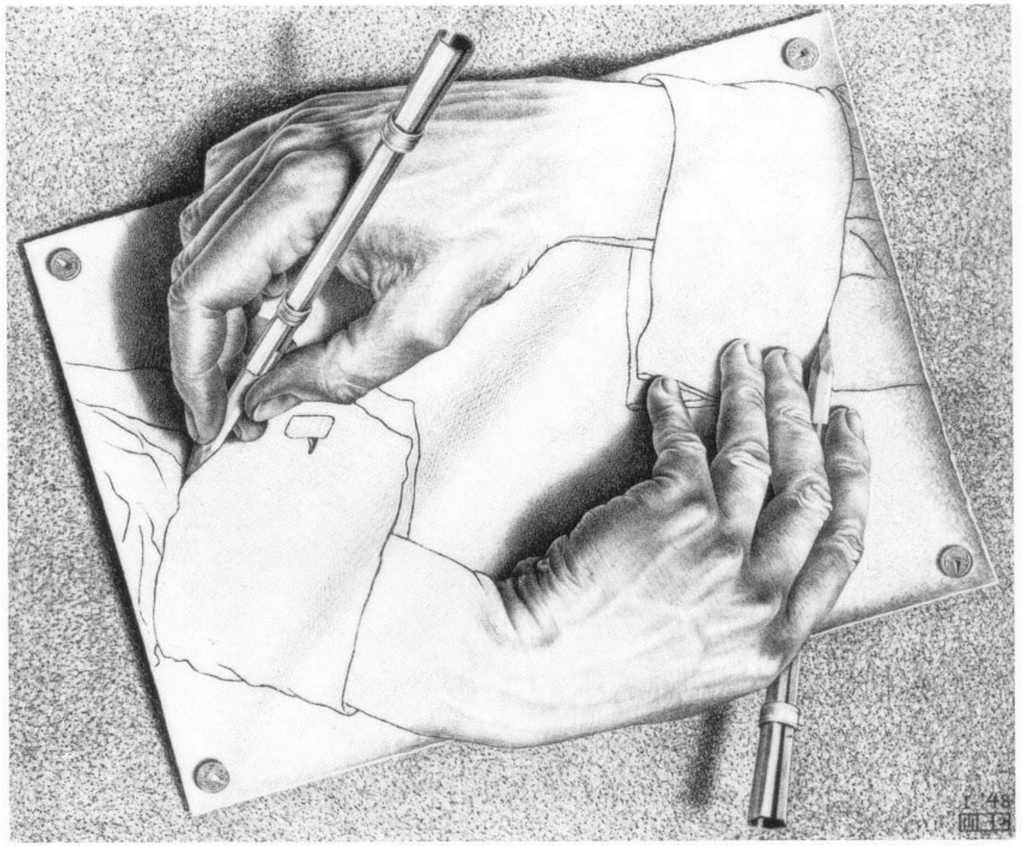
Concurrency

Polymorphism

Hosted

Lisp

Code is Data



Your program is a tree of data structures, instead of text strings

Primitive types

type	example
string	`“some string”`
character	`\c`
integer	`42, 42N`
floating point	`2.71, 2.71M`
boolean	`true, false`
nil	`nil`
symbol	`foo, bar, +`
keyword	`:some, ::key`

Data Structures

Lists	**Vectors**	**Maps**	**Sets**

`(1 2 3)`	`[1 2 3]`	`{:some 1`
		`:other 2`
		`:thing 3}`
			`#{1 2 3}`

Sequential	Sequential & Random Access	Associative	Membership

Some syntax

Prefix notation

```
f("hey!")
```

Calling functions

```
(println "Hello!")
```

list symbol string

Defining functions

```
(defn foo
  "Sums a and b"
  [a b]
  (+ a b))
```

symbol string vector list list

Some syntax

Special forms

```
def, if, do, let, quote, var, fn, loop, recur,
    throw, try, monitor-enter, monitor-exit
```

+ binding (destructuring) and interop forms

Everything else are either functions or macros built on top of these

Other syntactical niceties

- Destructuring
- Metadata
- Host interop
- Macros

```
(let [[a b c & d :as e] [1 2 3 4 5 6 7]]
  [a b c d e])

->[1 2 3 (4 5 6 7) [1 2 3 4 5 6 7]]
```

```
(let [{a :a, b :b, c :c, :as m :or {a 2 b 3}} {:a 5 :c 6}]
  [a b c m])

->[5 3 6 {:c 6, :a 5}]
```

```
(let [{:keys [a b c] :as m :or {a 2 b 3}} {:a 5 :c 6}]
  [a b c m])
```

```
user=> (def ^{:abc "Hello"} v)
#'user/v
user=> (meta #'v)
{:ns #<Namespace user>, :name v, :abc "Hello",
:column 1, :line 1, :file "NO_SOURCE_PATH"}
```

Macros

- Like functions, but operating on the program itself
  - Take in code (data) and output code (data)
- Libraries provide what's usually done at the language level:
  - **Pattern Matching** (core.match)
  - **Goroutines / CSP** (core.async)
  - **Logic Programming** (core.logic)
  - …

Functional Programming

First class functions

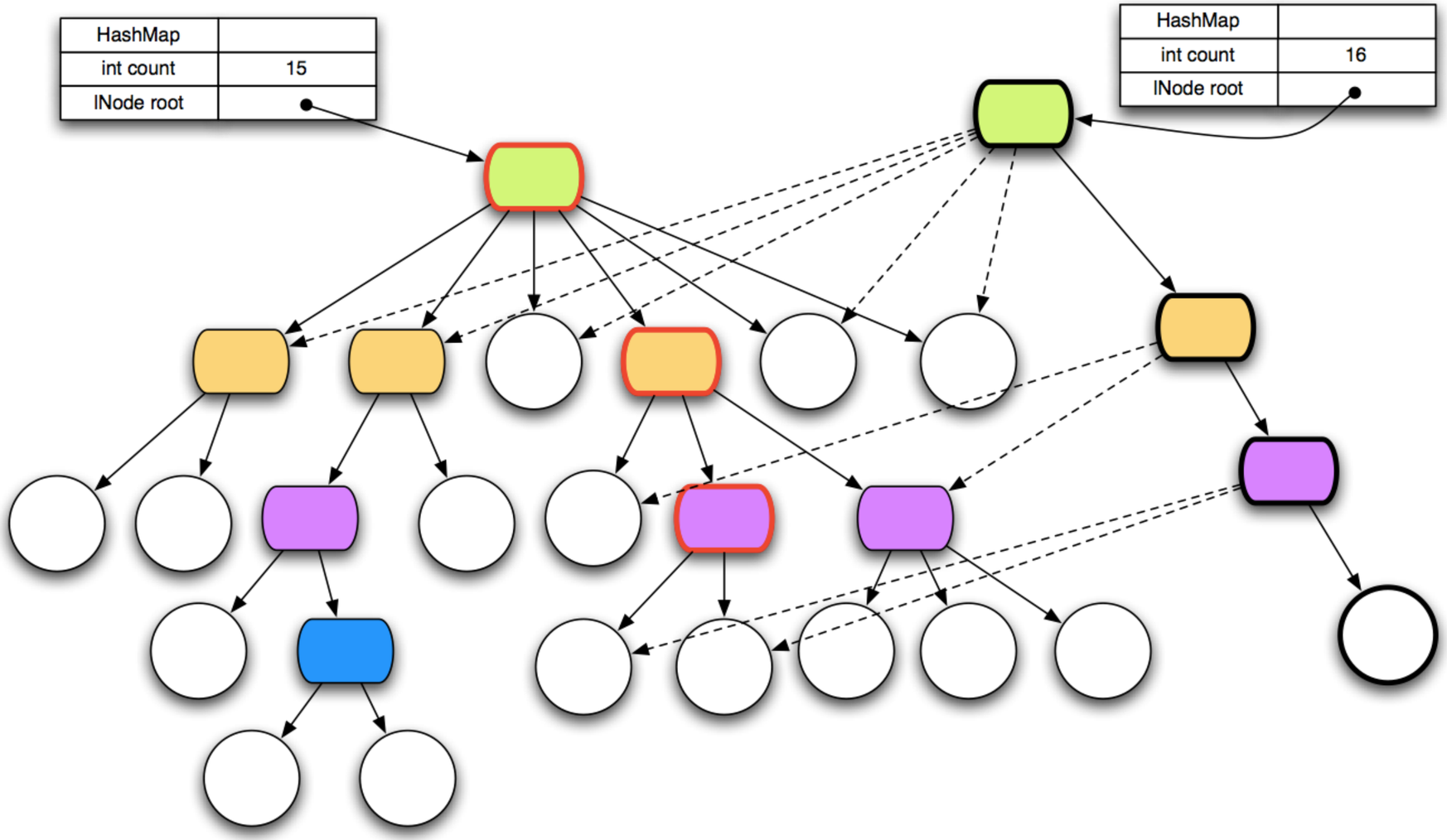
```
(def hello (fn [] "Hello world"))
-> #’user/hello

(hello)
-> "Hello world"

(defn hello [] "Hello world")
-> #'user/hello


(defn make-adder [x]
  (let [y x]
    (fn [z] (+ y z))))
(def add2 (make-adder 2))
(add2 4)
-> 6
```

```
(defn argcount
  ([] 0)
  ([x] 1)
  ([x y] 2)
  ([x y & more] (+ (argcount x y) (count more))))
-> #’user/argcount

(argcount)
-> 0

(argcount 1)
-> 1

(argcount 1 2)
-> 2

(argcount 1 2 3 4 5)
-> 5
```

Immutability

```
(let [my-vector [1 2 3 4]
      my-map {:fred "ethel"}
      my-list (list 4 3 2 1)]
  (list
    (conj my-vector 5)
    (assoc my-map :ricky "lucy")
    (conj my-list 5)
    ;the originals are intact
    my-vector
    my-map
    my-list))
-> ([1 2 3 4 5] {:ricky "lucy", :fred "ethel"} (5 4 3 2 1)
[1 2 3 4] {:fred "ethel"} (4 3 2 1))
```

Persistent data structures

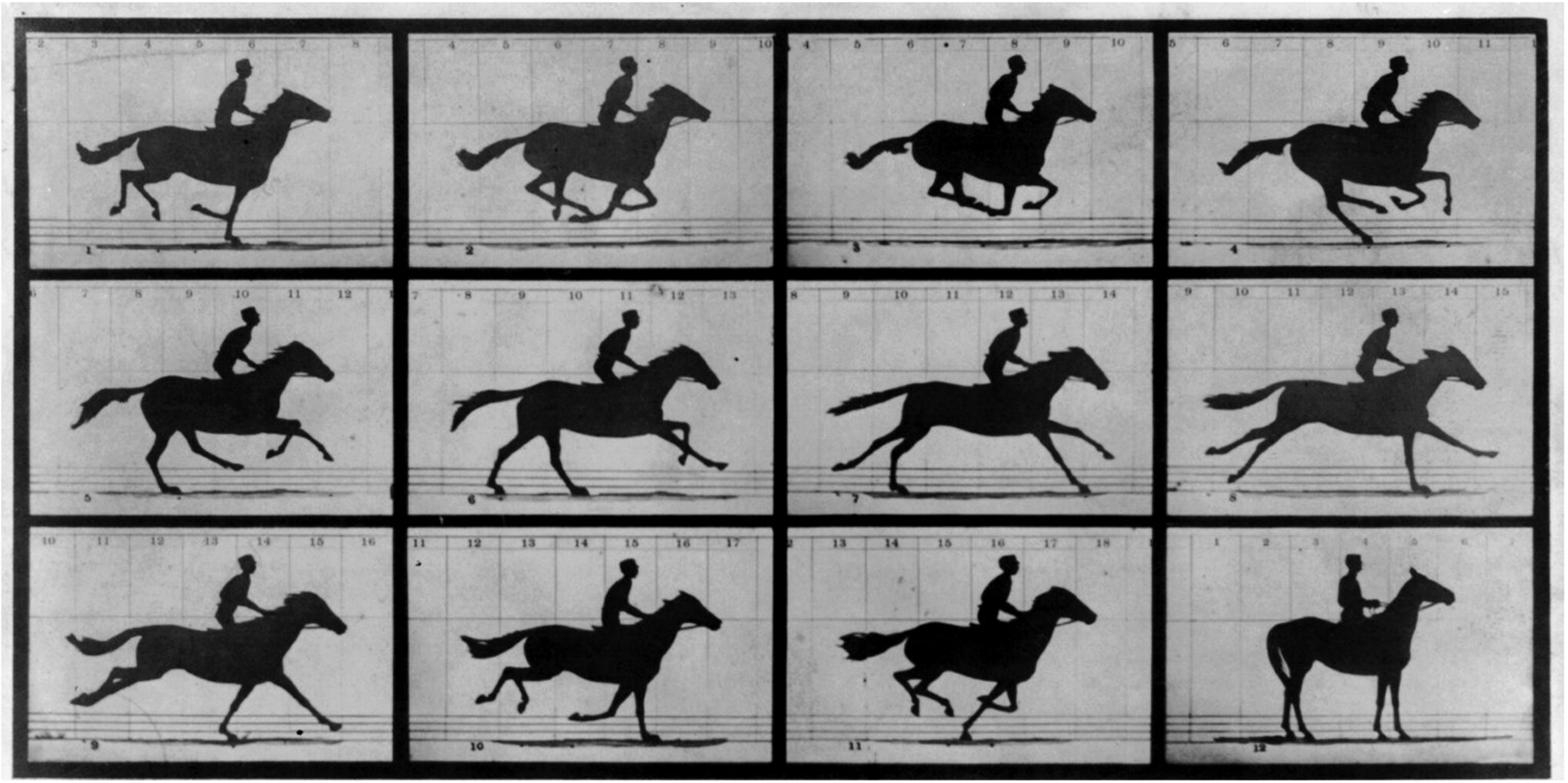


Recursive Looping

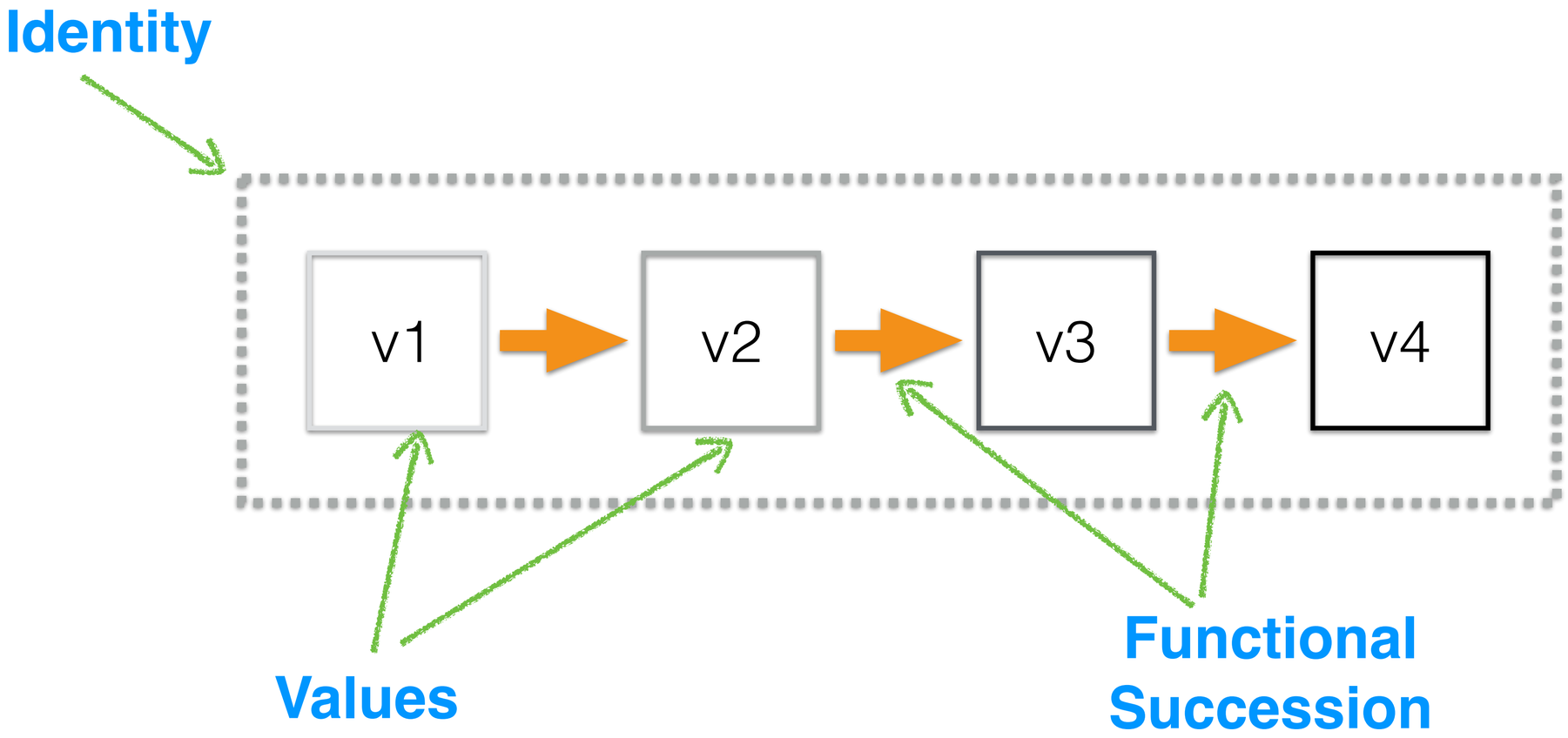
```
(defn my-zipmap [keys vals]
  (loop [my-map {}
         my-keys (seq keys)
         my-vals (seq vals)]
    (if (and my-keys my-vals)
      (recur (assoc my-map (first my-keys) (first my-vals))
             (next my-keys)
             (next my-vals))
      my-map)))
(my-zipmap [:a :b :c] [1 2 3])
-> {:b 2, :c 3, :a 1}
```

Concurrency

Identity and State



Identity and State



Concurrency Models

Vars	Thread-local changes
Atoms	Synchronous and Independent
Refs	Synchronous and Coordinated
Agents	Asynchronous and Independent

Vars

```
user=> (def ^:dynamic x 1)
user=> (def ^:dynamic y 1)
user=> (+ x y)
2
user=> (binding [x 2 y 3]
         (+ x y))
5
user=> (+ x y)
2
```

Atoms

```
user=> (def v (atom 0))
#'user/v

user=> @v
0

user=> (swap! v inc)
1

user=> (swap! v (fn [n] (* (+ n n) 2)))
4
```

Refs

```
(def account1 (ref 100))
(def account2 (ref 0))

user=> @account1
100

(defn transfer [amount from to]
    (dosync
       (alter from - amount)
       (alter to   + amount)))

user=> @account1
100
user=> @account2
0
user=> (transfer 100 account1 account2)
100
user=> @account1
0
user=> @account2
100
```

Agents

```
user=> (def currentsum (agent {:nums [] :sum 0}))
#'user/currentsum

(defn update-currentsum [current s]
    (let [new-nums (conj (:nums current) s)]
        {:nums new-nums
         :sum (reduce + new-nums)}))

user=> (send currentsum update-currentsum 5)
#<Agent @4cdac8 {:nums [], :sum 0}>

user=> (send currentsum update-currentsum 10)
#<Agent @4cdac8 {:nums [5], :sum 5}>

user=> @currentsum
{:nums [5 10], :sum 15}
```

Polymorphism

Multimethods

```
(defmulti encounter (fn [x y] [(:Species x) (:Species y)]))
(defmethod encounter [:Bunny :Lion] [b l] :run-away)
(defmethod encounter [:Lion :Bunny] [l b] :eat)
(defmethod encounter [:Lion :Lion] [l1 l2] :fight)
(defmethod encounter [:Bunny :Bunny] [b1 b2] :mate)
(def b1 {:Species :Bunny :other :stuff})
(def b2 {:Species :Bunny :other :stuff})
(def l1 {:Species :Lion :other :stuff})
(def l2 {:Species :Lion :other :stuff})
(encounter b1 b2)
-> :mate

(encounter b1 l1)
-> :run-away

(encounter l1 b1)
-> :eat

(encounter l1 l2)
-> :fight
```

Protocols

```
(defprotocol P
  (foo [x])
  (bar-me [x] [x y]))

(deftype Foo [a b c]
  P
  (foo [x] a)
  (bar-me [x] b)
  (bar-me [x y] (+ c y)))

(bar-me (Foo. 1 2 3) 42)
-> 45

(foo
 (let [x 42]
   (reify P
     (foo [this] 17)
     (bar-me [this] x)
     (bar-me [this y] x))))
-> 17
```

Hosted

Hosted

- VMs, not OSes, are the platforms of the future, providing:
  - Type system
  - Libraries
  - Memory and other resource management
  - Bytecode + JIT compilation
- Language as platform vs. language + platform
  - Old way - each language defines its own runtime
  - New way (JVM, .Net) - common runtime independent of language
- Platforms are dictated by clients

Implementations

JVM

Clojure

JavaScript

ClojureScript

CLR

Clojure-CLR

Some people working on other backends

Interop

Clojure

```
(.toUpperCase "fred")
-> "FRED"

(.getName String)
-> "java.lang.String"

(System/getProperty "java.vm.version")
-> "1.6.0_07-b06-57"
```

ClojureScript

```
(.the-method target-object args)

(.-property target-object)

(js/alert "Hello World!")
```

Getting started with Clojure

Books

- The Joy of Clojure
- Programming Clojure

Environment

- Leiningen
- Text editor + REPL over IDE
  - Emacs, Vim, LightTable, Sublime

Presentations

- Rich Hickey's keynotes (!)
- ClojureTV on YouTube

People to follow

Chas Emerick, David Nolen, Stuart Sierra, Michael Fogus, Chris Houser, Stuart Halloway & many more!