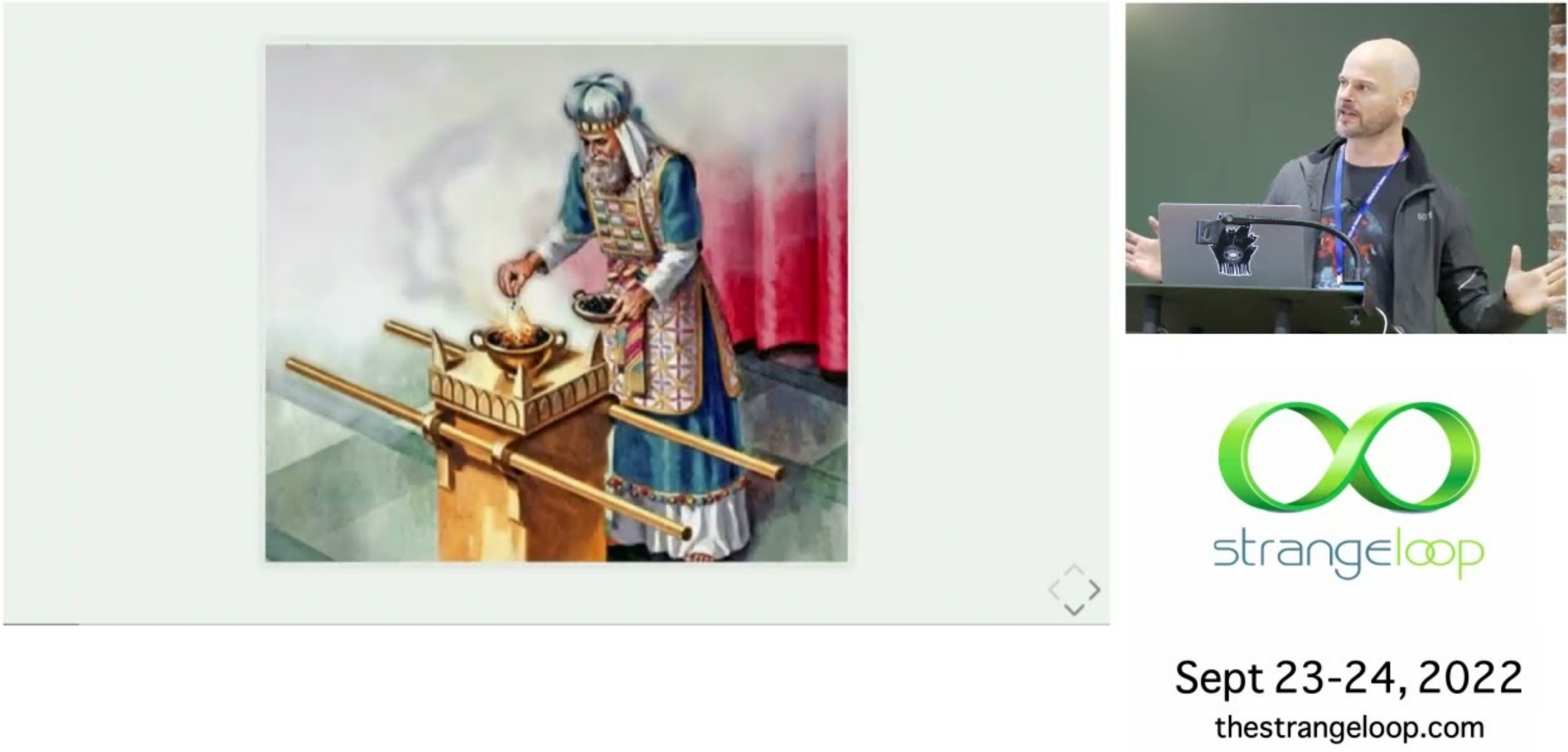
CS 420 Advanced Programming Languages
Fall Semester, 2022
Doc 16 Functions, Some Concurrency
Oct 20, 2022

Stop Writing Dead Programs

Jack Rusher (Strange Loop 2022)

https://www.youtube.com/watch?v=8Ab3ArE8W3s



Sept 23-24, 2022

thestrangeloop.com

Read from inside out

```
(defn calculate
   [a b c d]
   (+ (/ (+ a b)  c) d))
```

let

->

->>

let

Allows you to

    compute partial results
    give results names

Compute average of three numbers

```
(defn average
     [a b c]
     (/ (+ a b c) 3))
```

```
(defn average
  [a b c]
  (let [sum (+ a b c)
        size 3]
    (/ sum size)))
```

Using let

```
(defn calculate
   [a b c d]
   (+ (/ (+ a b)  c) d))
```

```
(defn calculate-2
  [a b c d]
  (let [a+b (+ a b)
        divide-c (/ a+b c)
        plus-d (+ divide-c d)]
    plus-d))
```

-> Threading macro

```
(-> x)
(-> x form1 … formN)
```

Inserts x as second element in form1

Then inserts form1 as second element in form2

etc.

-> Example

```
(def c 5)              (- (/ (+ c 3) 2) 1)


(-> c
    (+ 3)                 (+ c 3)
    (/ 2)                 (/ 8 2)
    (- 1))                (- 4 1)
```

-> Example

```
(def c 5)


(-> c
    (+ 3)
    (/ 2)
    dec)
```

```
(dec (/ (+ c 3) 2))
```

(+ c 3)

(/ **8** 2)

(dec **4**)

-> Example

```
(-> "a b c d"
    .toUpperCase
    (.replace "A" "X")
    (.split " ")
    first)
```

```
(.toUpperCase "a b c d")
(.replace "A B C D"  "A" "X")
(.split "X B C D"  " ")
(first {"X", "B", "C", "D"} )
```

-> Example

```
(-> person :employer :address :city)

(def person
  {:name "Mark Volkmann"
   :address {:street "644 Glen Summit"
             :city "St. Charles"
             :state "Missouri"
             :zip 63304}
   :employer {:name "Object Computing, Inc."
              :address {:street "12140 Woodcrest Dr."
                        :city "Creve Coeur"
                        :state "Missouri"
                        :zip 63141}}})
```

->> Threading macro

```
(->> x)
(->> x form1 … formN)
```

Inserts x as last element in form1

Then inserts form1 as last element in form2

etc.

->> Example

```
(def c 5)

(->> c
    (+ 3)                  (+ 3  c)
    (/ 2)                  (/ 2 8)
    (- 1))                 (- 1 1/4)
```

as-> Allow Threading in different locations

```
(as-> 5 c
    (+ 3 c)
    (/ c 2)
    (- c 1))
```

bind 5 to c	
(+ 3 **5**)	bind 8 to c
(/ **8** 2)	bind 4 to c
(- **4** 1)	return 3

Multiple lines

```
(defn average
   [a b c]
   (println (str "a is " a)
   (+ 1 3)
   (/ (+ a b c) 3))
```

```
(average 1 2 3)
```

returns 2

prints on standard out

```
    a is 1
```

Why not use def & multiple lines?

```
(defn average-bad
  [a b c]
  (def sum (+ a b c))
  (def size 3)
  (/ sum size))
```

```
(defn average
  [a b c]
  (let [sum (+ a b c)
        size 3]
    (/ sum size)))
```

(average-bad 1 2 3)	2
sum	6
size	3

(average 1 2 3)	2
sum	Error
size	Error

def defines global names/values

let defines local names/values

Don't use def inside functions

Symbols, Values & Binding

Symbols reference a value

foo & bar are symbols

They are bound to values

```
(def foo "hi")

(def bar (fn [n] (inc n)))
```

Binding & Shadowing

```
→ (def x 1)

  (defn shadow
    [x]
 ●  (println "Start function x=" x)
    (let [x 20]
      (println "In let x=" x))
    (println "After let x=" x))


  (println "Before function x=" x)
  (shadow 10)
  (println "After function x=")
```

```
Before function x= 1

 Start function x= 10

In let x= 20

After let x= 10

After function x= 1
```

Bindings, Shadowing & Functions

```
(dec 10)

(let [dec "December"
      test (dec 10)]
  test)

     Compile Error
```

```
(dec 10)

(def dec "December")

(dec 10)              Compile Error

(clojure.core/dec 10)
```

```
(def + -)
(+ 4 3)          1
```

juxt

Combines a set of functions
Returns vector applying each function to input

```
(def basic-math (juxt + - * /))
(basic-math 2 5)
```

```
[7 -3 10 2/5]
```

```
(def split-collection (juxt take drop))
(split-collection 4 (range 9))
```

```
[(0 1 2 3) (4 5 6 7 8)]
```

juxt in Sorting

```
((juxt :last :first) {:last "Adams" :first "Zak"} )                  ["Adams" "Zak"]


(sort-by (juxt :last :first) [{:last "Adams" :first "Zak"}           ({:last "Adams", :first "Zak"}
                              {:last "Zen" :first "Alan"}             {:last "Smith", :first "Alan"}
                              {:last "Smith" :first "Alan"}])         {:last "Zen", :first "Alan"})



(sort-by (juxt :first :last) [{:last "Adams" :first "Zak"}          ({:last "Smith", :first "Alan"}
                         {:last "Zen" :first "Alan"}                 {:last "Zen", :first "Alan"}
                         {:last "Smith" :first "Alan"}])             {:last "Adams", :first "Zak"})
```

comp

Takes a sequence of functions

Composes the functions

```
((comp str +) 8 8 8)                           "24"

(def fourth (comp first rest rest rest))
(fourth [:a :b :c :d :e])                       :d
```

nth

Given n can we produce

```
(comp first rest rest rest … rest)
```

where we have n -1 rest's?

Yes We Can!

```
(defn fnth
  [n]
  (apply comp
         (cons first
               (take (dec n) (repeat rest)))))
```

```
((fnth 1) [:a :b :c :d :e])                    :a

((fnth 3) [:a :b :c :d :e])                    :c
```

How does this work?

```
(repeat rest)
```

infinite lazy sequence of rest

```
(take (dec n) (repeat rest))
```

```
'(rest rest … rest)    ;n-1 rest's
```

```
(cons first
      (take (dec n) (repeat rest)))
```

```
'(first rest rest … rest)
```

```
(apply comp
       (cons first
             (take (dec n) (repeat rest))))
```

```
(comp first rest rest … rest)
```

memoize

```
(memoize f)
```

Caches results of function f

Uses cached value next time f is called with same arguments

```
(defn adder
  [x]
  (println "adder" x)
  (inc x))

(def adder-memoized (memoize adder))
```

(adder-memoized 1)	prints 1, returns 2
(adder-memoized 1)	returns 2
(adder-memoized 2)	prints 2, returns 3
(adder-memoized 1)	returns 2

memoize - Cache Size

Cache is a map

Contains return values for each different set of input arguments

clojure.core.cache contains more sophisticated caches

Multi-Methods

dispatch function

```
(defmulti even-odd even?)

(defmethod even-odd true
  [n]
  (str n " is even"))

(defmethod even-odd false
  [n]
  (str n " is odd"))
```

value of dispatch function
handled by this method

Multi-Methods

```
(defmulti even-odd even?)

(defmethod even-odd true
  [n]
  (str n " is even"))

(defmethod even-odd false
  [n]
  (str n " is odd"))
```

```
(even-odd 5)       5 is odd
(even-odd 4)       4 is even
```

Default values

```
(defmulti fibonacci identity)

(defmethod fibonacci 0
  [n]
  0)

(defmethod fibonacci 1
  [n]
  1)

(defmethod fibonacci :default
  [n]
  (+ (fibonacci (dec n)) (fibonacci (- n 2))))
```

```
(fibonacci 1)     1

(fibonacci 10)    55
```

Dispatch Function can be any function

```
(defmulti types class)

(defmethod types java.lang.String
  [x]
  "it is a string")

(defmethod types java.lang.Long
  [x]
  "it is a Long")

(defmethod types :default
  [x]
  "Don't know")
```

```
(types "ca")      "it is a string"
(types 12)        "it is a Long"
(types 12.3)      "Don't know"
```

Multiple Arguments

```
(defmulti by-size (fn [a b] (size a)))

(defmethod by-size :small
  [x y]
  "small")

(defmethod by-size :small
  [x y]
  "small")

(defmethod by-size :medium
  [x y]
  "meduim")

(defmethod by-size :defualt
  [x y]
  "large & other")
```

```
(defn size
  [x]
  (cond
   (< x 5) :small
   (< x 20) :medium
   (< x 100) :large))
```

```
(by-size 2 20)          "small"
(by-size 10 20)         "meduim"
```

Vectors as Match

```
(defmulti by-size (fn [a b] [(size a) (size b)]))

(defmethod by-size [:small :small]
  [x y]
  "small-small")

(defmethod by-size [:small :large]
  [x y]
  "small-large")

(defmethod by-size [:medium :meduim]
  [x y]
  "meduim-medium")

(defmethod by-size :default
  [x y]
  "other")
```

```
(by-size 2 90)     "small-large"
(by-size 10 20)    "other"
```

Warning about defmulti

defmulti is define once

If you need to modify your defmulti need to remove it from the bindings

In previous example used

```
    (ns-unmap *ns* 'by-size)
```

One Last Example

```
(defmulti by-children (fn [[a c b]] [(nil? b) (nil? c)]))

(defmethod by-children  [true true]
  [x]
  "no children")

(defmethod by-children  [true false]
  [x]
  "right child")

(defmethod by-children  [false true]
  [x]
  "left children")

(defmethod by-children  [false false]
  [x]
  "both children")
```

```
(by-children [1 4 nil])    "right child"
(by-children [1 nil nil])   "no children"
```

Open-Closed Principle

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

Wikipedia

Delay

Suspends execution of code until delay is dereferenced

Caches result

Second time dereferenced returns cached result

Thread safe

```
(def wait (delay (println "do it now") (+ 1 2)))

 @wait          prints "do it now", returns 3
 @wait          returns 3
```

realized?

Returns true if a value has been produced for a promise, delay, future or lazy sequence.

```
(def wait (delay (println "do it now") (+ 1 2)))
```

(realized? wait)	false
@wait	prints "do it now", returns 3
(realized? wait)	true
@wait	returns 3

Example - Proxy for Expensive Operation

```
(defn fetch-page
  [url]
  {:url url
   :contents (delay (slurp url))})
```

```
(def result (fetch-page "http://www.eli.sdsu.edu/index.html"))

(:contents result)                    #<Delay@2fcc470c: :pending>

(realized? (:contents result))        false

@(:contents result)                   "<!DOCTYPE html>\n<html lang=\"en\">\n …"
```

@ and deref

```
@(:contents result)
```

```
 (deref (:contents result))
```

They do the same thing

Future

Computes body on another thread

Use @ or deref to get answer

@, deref blocks until computation is done

```
(def long-calculation (future (apply + (range 1e8))))
@long-calculation
```

Future & Delay in ending program

When you end your program there will be a 1 minute delay if you used future

End your program with (shutdown-agents)

```
(def long-calculation (future (apply + (range 1e8))))

@long-calculation

(shutdown-agents)
```

deref with Timeout

```
 (deref (future (Thread/sleep 5000) :done!)
        1000
        :impatient!)
   ;= :impatient!
```

Promise

one-time, single value pipe

```
(def p (promise))
(realized? p)        false
(deliver p 42)       #<core$promise$reify__1707@3f0ba812: 42>
(realized? p)        true
@p                   42
(deliver p 50)       nil
@p                   42
```

Promise

Simple way to send data back from thread

References

Time, State, Identity

Time

Relative moments when an event occurs

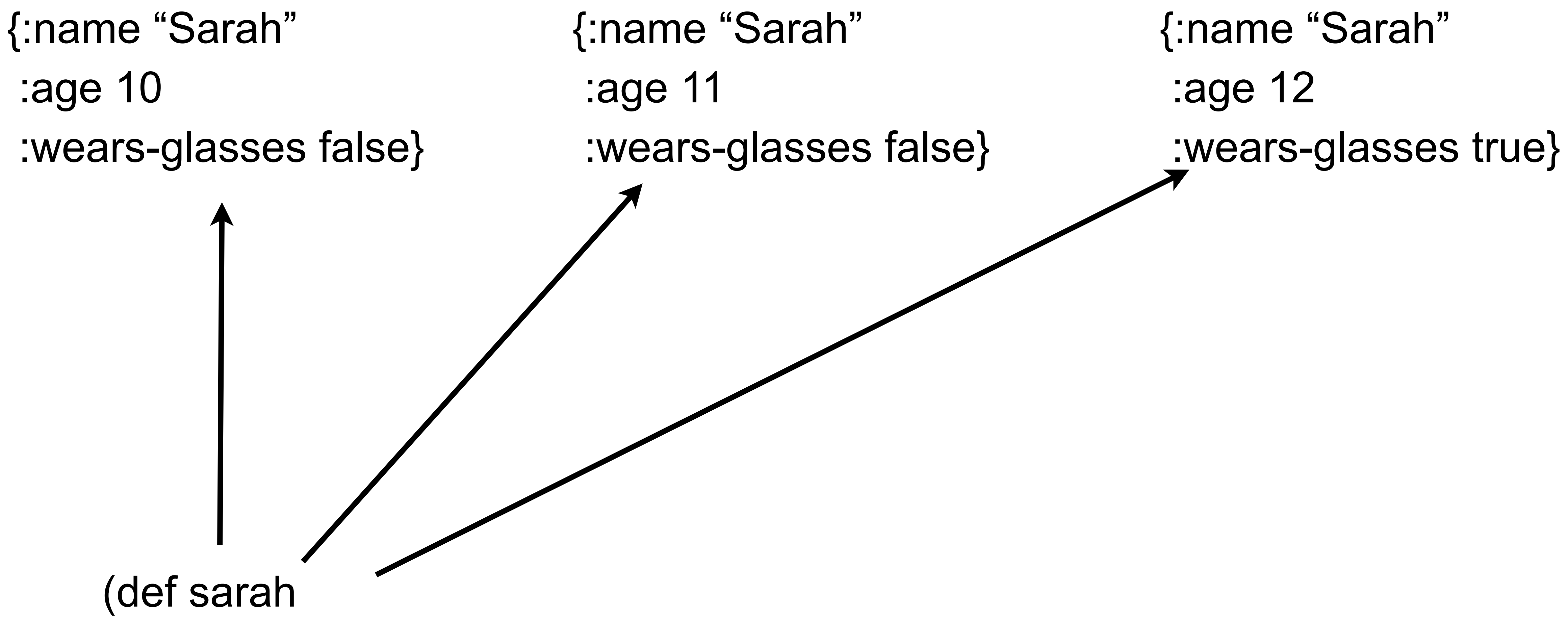
State

Snapshot of entity’s properties at a moment in time

Identity

Logical entity identified by a common stream of states occurring over time

State & Identity



```
{:name “Sarah”
 :age 10
 :wears-glasses false}
```

```
{:name “Sarah”
 :age 11
 :wears-glasses false}
```

```
{:name “Sarah”
 :age 12
 :wears-glasses true}
```

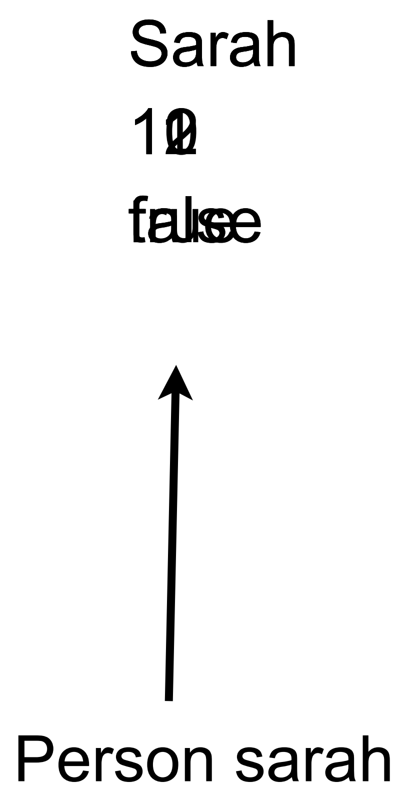
```
(def sarah
```

Java

```
 class Person {
     public String name;
     public int age;
     public boolean wearsGlasses;

     public Person (String name, int age, boolean wearsGlasses) {
       this.name = name;
       this.age = age;
       this.wearsGlasses = wearsGlasses;
    }
}
```

State & Identity

Complexted in Java



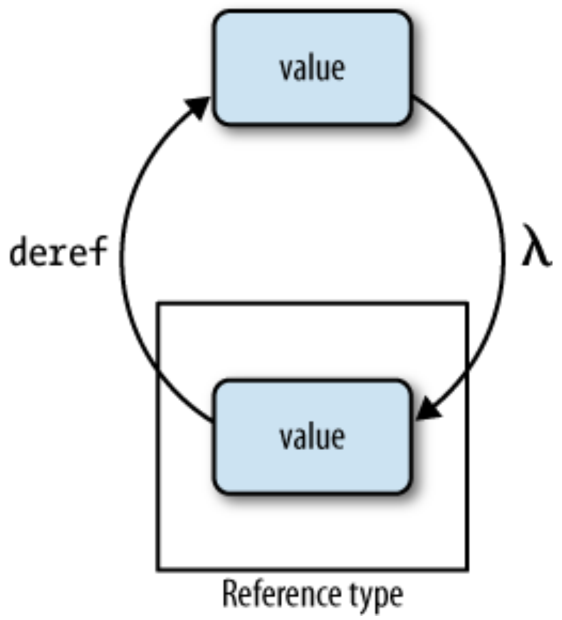
Sarah

10 12

true false

Person sarah

Reference Type Basics



var, ref, atom, agent

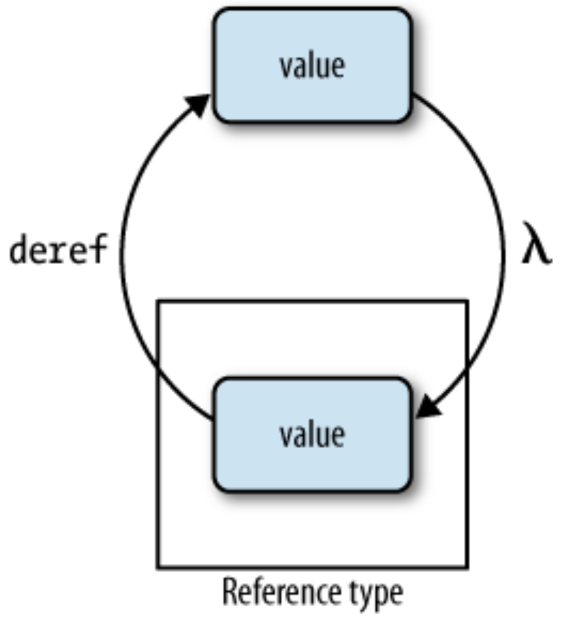
All are pointers

Can change pointer to point to different data

Dereferencing will never block

Each type as different way of setting/changing its value

Reference Type Basics



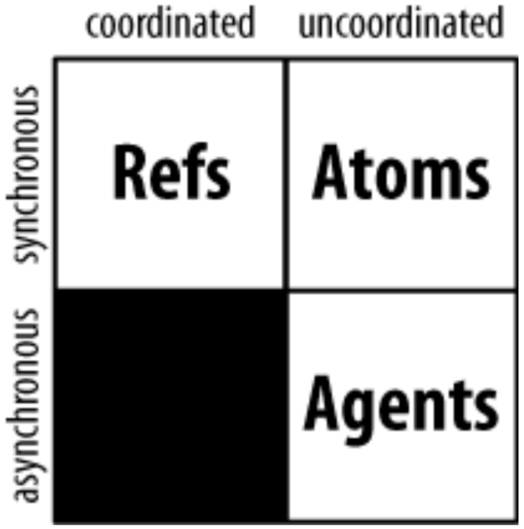
var, ref, atom, agent

Each type

- Can have meta data
- Can have watches (observers)
  - Call specified function when value is change
- Can have validator
  - Enforce constraints on values pointer can point to

Features of each Type

	Ref	Agent	Atom	Var
Coordinated	X			
Asynchronous		X		
Retriable	X		X	
Thread-local				X



	coordinated	uncoordinated
synchronous	Refs	Atoms
asynchronous	■	Agents

Synchronous - block until operation completes

Asynchronous - Non blocking, operation can compete on separate thread

Coordinated - Supports transactions

Thread-local - Changes made are local to current thread

Creating & Referencing Each Type

```
(def ref-example (ref 10))
@ref-example
(deref ref-example)

(def agent-example (agent 10))
@agent-example
(deref agent-example)

(def atom-example (atom 10))
@atom-example
(deref atom-example)

(def var-example 10)
var-example
```

Note the difference

Watches

```
(defn cat-watch
  [key pointer old new]
  (println "Watcher" key pointer old new))

(def cat 4)

(add-watch (var cat) :cat cat-watch)

(def cat 10)

(remove-watch (var cat) :cat)

(def cat 20)
```

Output in Console

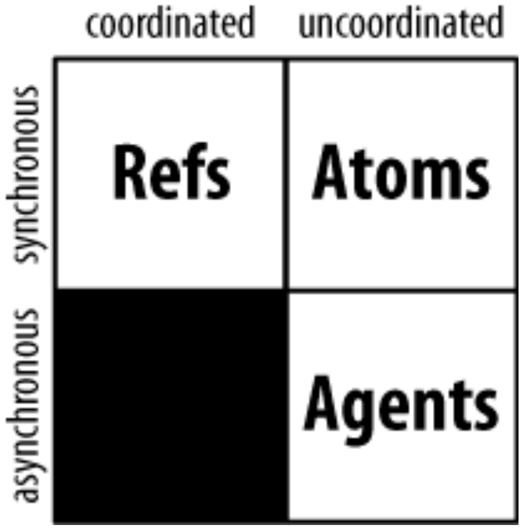
```
Watcher :cat #'user/ca
```

Validator

```
(def cat 4)

(set-validator! (var cat) #(> 10 %))

(def cat 9)

(def cat 20)                    ;;exception
```

Features of each Type

	Ref	Agent	Atom	Var
Coordinated	X			
Asynchronous		X		
Retriable	X		X	
Thread-local				X



	coordinated	uncoordinated
synchronous	Refs	Atoms
asynchronous		Agents

Synchronous - block until operation completes

Asynchronous - Non blocking, operation can compete on separate thread

Coordinated - Supports transactions

Thread-local - Changes made are local to current thread

Atoms

Changes are

- Synchronous
- Uncoordinated
- Atomic

Synchronous

- Code waits until change done

Uncoordinated

- No transaction support

Atomic

- Threads only see old or new value
- Never see partially changed data

Atoms - Methods for change

swap!

Applies function to current state for new state

reset!

Changes state to given value

compare-and-set!

Changes state to given value only if current value is what you think it is

reset!

```
(def a (atom 0))

@a                   0

(reset! a 5)         5

@a                   5
```

swap!

```
(def a (atom 0))

@a                  0

(swap! a inc)       1

@a                  1
```

swap!

```
(def sarah (atom {:name "Sarah" :age 10 :wears-glasses? false}))

(swap! sarah update-in [:age] + 3)          {:name "Sarah", :age 13,
                                            :wears-glasses? false}

@sarah                                      {:name "Sarah", :age 13,
                                            :wears-glasses? false}
```

swap! is Atomic

```
(swap! sarah (comp #(update-in % [:age] inc)
                   #(assoc % :wears-glasses? true)))
```

Compound operation on sarah

What happens if other thread reads sarah during swap!

It gets the old value

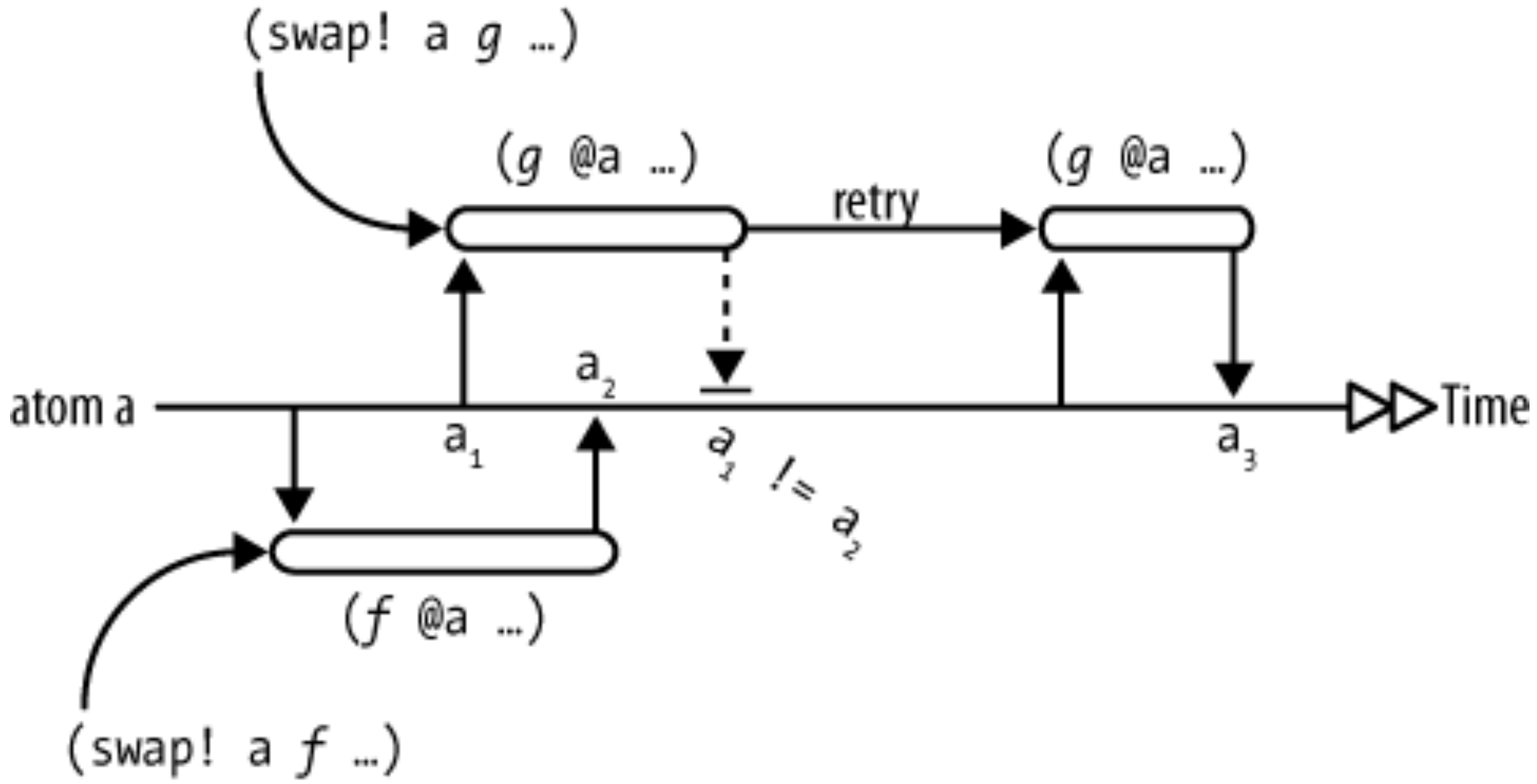
swap! is Atomic

```
(swap! sarah (comp #(update-in % [:age] inc)
                   #(assoc % :wears-glasses? true)))
```

What happens if other thread modifies sarah during swap!

It retries until it can read the new value

Then modifies sarah


(swap! a g …)
(g @a …)
retry
(g @a …)
atom a
a_1
a_2
a_1 != a_2
a_3
Time
(f @a …)
(swap! a f …)