relevance

clojure

stuart halloway
http://thinkrelevance.com

functional

lisp

concurrency

embraces JVM

elegant

example:
refactor apache commons isBlank

initial implementation

```
public class StringUtils {
  public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (int i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- type decls

```
public class StringUtils {
  public isBlank(str) {
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- class

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  for (i = 0; i < strLen; i++) {
    if ((Character.isWhitespace(str.charAt(i)) == false)) {
      return false;
    }
  }
  return true;
}
```

+ higher-order function

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  every (ch in str) {
    Character.isWhitespace(ch);
  }
  return true;
}
```

- corner cases

```
public isBlank(str) {
  every (ch in str) {
    Character.isWhitespace(ch);
  }
}
```

lispify

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

functional

is

simpler

	imperative	**functional**
functions	1	1
classes	1	0
exit points	3	1
variables	2	0
branches	3	0
boolean ops	1	0
function calls	3	2
total	***14***	***4***

java interop

java new

java	`new Widget("foo")`
clojure	`(new Widget "foo")`
clojure sugar	`(Widget. "red")`

access static members

java	`Math.PI`
clojure	`(. Math PI)`
clojure sugar	`Math/PI`

access instance members

java	`rnd.nextInt()`
clojure	`(. rnd nextInt)`
clojure sugar	`(.nextInt rnd)`

chaining access

java	`person.getAddress().getZipCode()`
clojure	`(. (. person getAddress) getZipCode)`
clojure sugar	`(.. person getAddress getZipCode)`

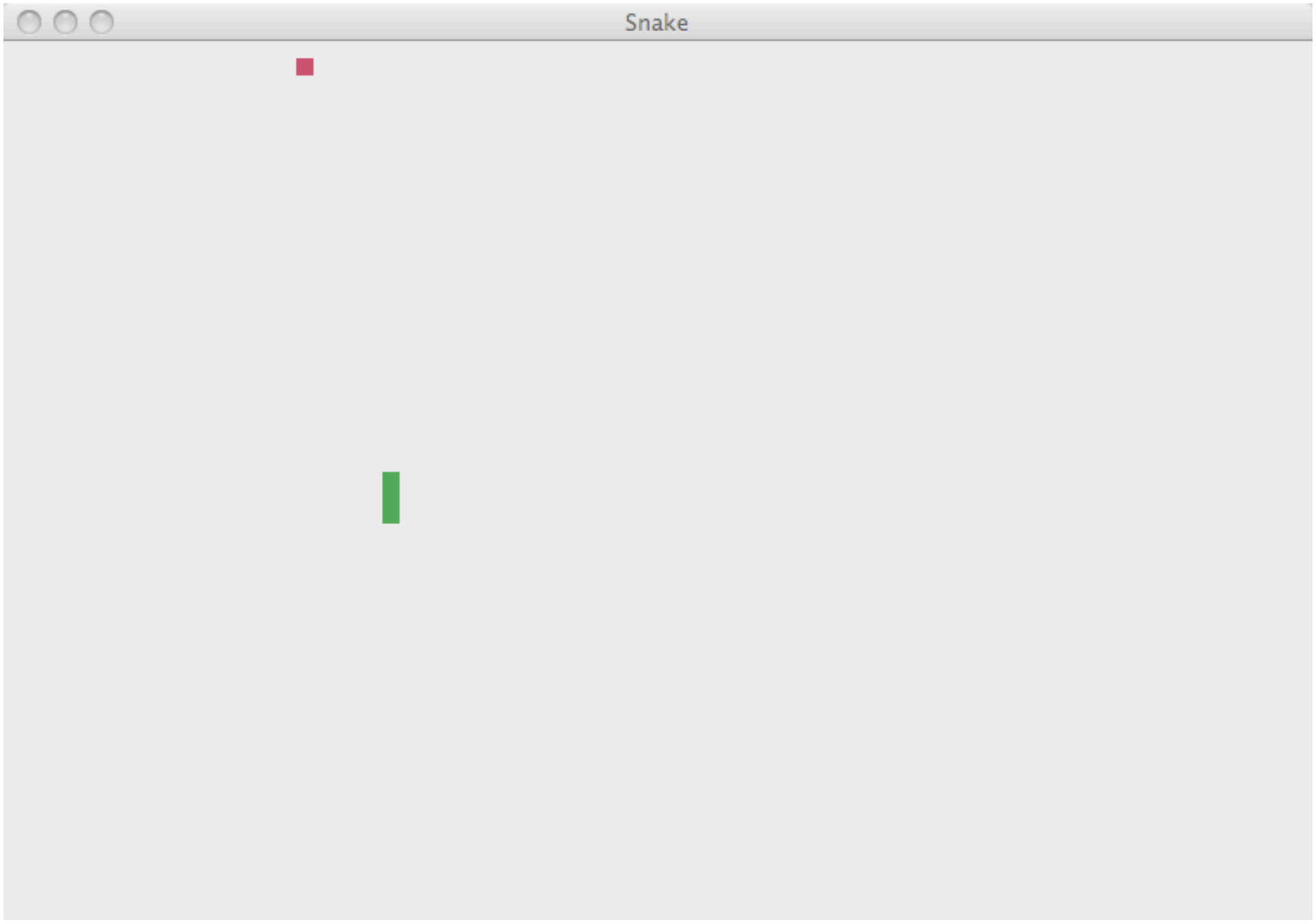
parenthesis count

java	( ) ( ) ( ) ( )
clojure	( ) ( ) ( )

atomic data types

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
regex	`#"fo*"`	Pattern
a. p. integer	`42`	Integer/Long/BigInteger
double	`3.14159`	Double
a.p. double	`3.14159M`	BigDecimal
boolean	`TRUE`	Boolean
nil	`nil`	`null`
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`	N/A

game break!



Sample Code:
http://github.com/stuarthalloway/programming-clojure

sequences

literal sequences

type	properties	example
list	singly-linked, insert at front	`(1 2 3)`
vector	indexed, insert at rear	`[1 2 3]`
map	key/value	`{:a 100` `:b 90}`
set	key	`#{:a :b}`

first / rest / cons

```
(first [1 2 3])
-> 1

(rest [1 2 3])
-> (2 3)

(cons "hello" [1 2 3])
-> ("hello" 1 2 3)
```

take / drop

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

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

map / filter / reduce

```
(range 10)
-> (0 1 2 3 4 5 6 7 8 9)

(filter odd? (range 10))
-> (1 3 5 7 9)

(map odd? (range 10))
-> (false true false true false true
false true false true)

(reduce + (range 10))
-> 45
```

sort

```
(sort [ 1 56 2 23 45 34 6 43])
-> (1 2 6 23 34 43 45 56)
```

```
(sort > [ 1 56 2 23 45 34 6 43])
-> (56 45 43 34 23 6 2 1)
```

```
(sort-by #(.length %)
  ["the" "quick" "brown" "fox"])
-> ("the" "fox" "quick" "brown")
```

interpose

```
(interpose \, ["list" "of" "words"])
-> ("list" \, "of" \, "words")

(apply str
  (interpose \, ["list" "of" "words"]))
-> "list,of,words"

(use 'clojure.contrib.str-utils)
(str-join \, ["list" "of" "words"]))
-> "list,of,words"
```

predicates

```
(every? odd? [1 3 5])
-> true

(not-every? even? [2 3 4])
-> true

(not-any? zero? [1 2 3])
-> true

(some nil? [1 nil 2])
-> true
```

conj / into

```
(conj '(1 2 3) :a)
-> (:a 1 2 3)

(into '(1 2 3) '(:a :b :c))
-> (:c :b :a 1 2 3)

(conj [1 2 3] :a)
-> [1 2 3 :a]

(into [1 2 3] [:a :b :c])
-> [1 2 3 :a :b :c]
```

infinite sequences

```
(set! *print-length* 5)
-> 5

(iterate inc 0)
-> (0 1 2 3 4 ...)

(cycle [1 2])
-> (1 2 1 2 1 ...)

(repeat :d)
-> (:d :d :d :d :d ...)
```

example:
refactor apache
commons
indexOfAny

indexOfAny behavior

```
StringUtils.indexOfAny(null, *)                = -1
StringUtils.indexOfAny("", *)                  = -1
StringUtils.indexOfAny(*, null)                = -1
StringUtils.indexOfAny(*, [])                  = -1
StringUtils.indexOfAny("zzabyycdxx",['z','a']) = 0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) = 3
StringUtils.indexOfAny("aba", ['z'])           = -1
```

indexOfAny impl

```
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars)
{
  if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
    return -1;
  }
  for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    for (int j = 0; j < searchChars.length; j++) {
      if (searchChars[j] == ch) {
        return i;
      }
    }
  }
  return -1;
}
```

simplify corner cases

```
public static int indexOfAny(String str, char[] searchChars)
{
  when (searchChars)
    for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);
      for (int j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

- type decls

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      for (j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

+ when clause

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      when searchChars(ch) i;
    }
  }
}
```

+ comprehension

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for ([i, ch] in indexed(str)) {
      when searchChars(ch) i;
    }
  }
}
```

lispify!

```
(defn index-filter [pred coll]
  (when pred
    (for [[idx elt] (indexed coll) :when (pred elt)] idx)))
```

functional

is

simpler

	imperative	**functional**
functions	1	1
classes	1	0
exit points	3	1
variables	3	0
branches	4	0
boolean ops	1	0
function calls*	6	3
total	***19***	***5***

functional

is

more general!

reusing index-filter

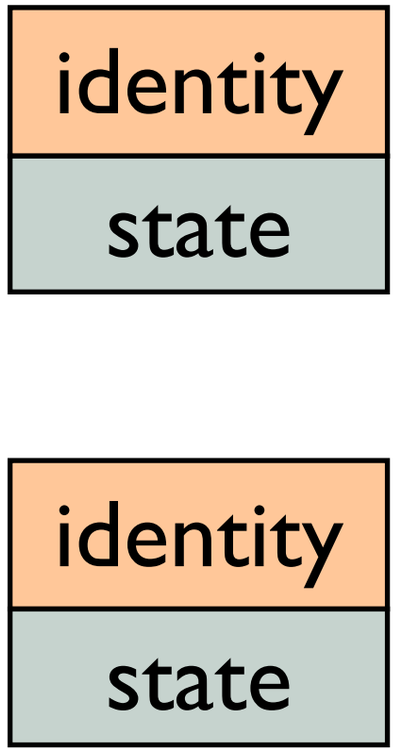
```
; idxs of heads in stream of coin flips
(index-filter #{:h}
[:t :t :h :t :h :t :t :t :h :h])
-> (2 4 8 9)

; Fibonaccis pass 1000 at n=17
(first
  (index-filter #(> % 1000) (fibo)))
-> 17
```

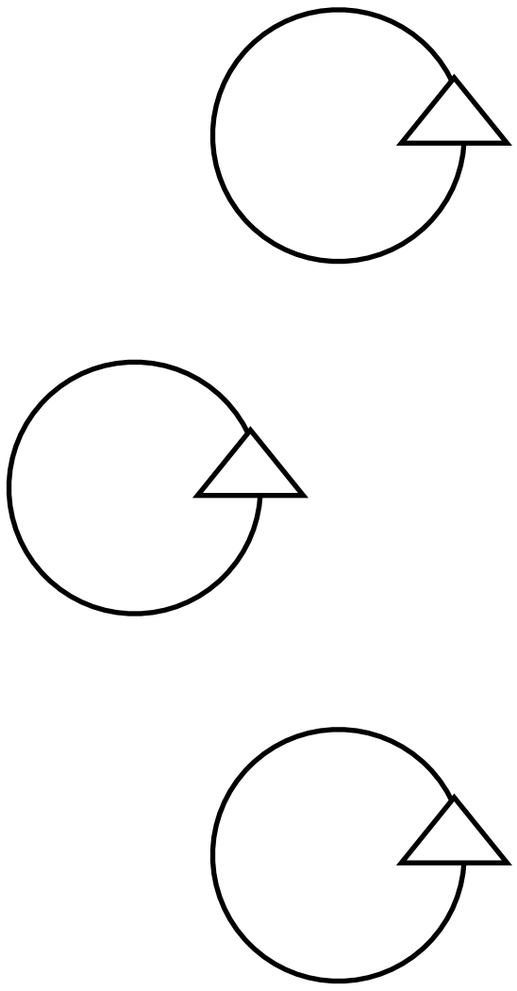
imperative	**functional**
searches strings	searches *any sequence*
matches characters	matches *any predicate*
returns first match	returns *lazy seq of all matches*

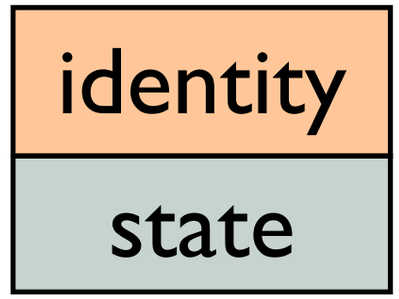
concurrency

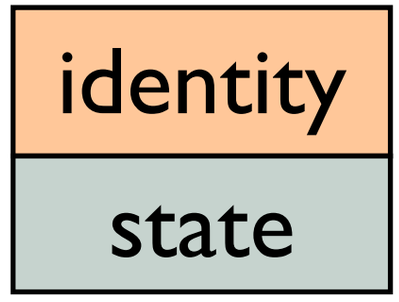
traditional oo



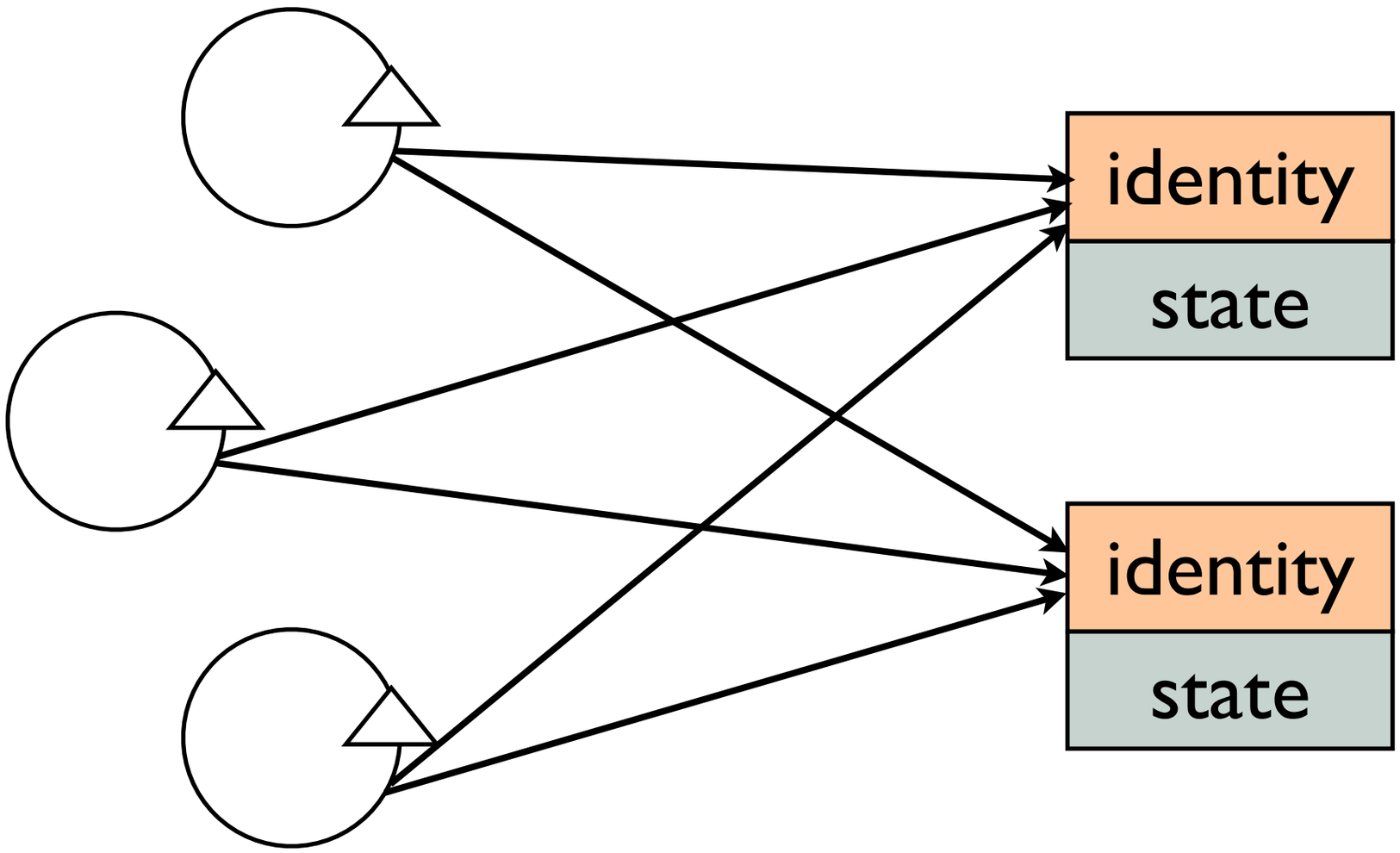
traditional oo



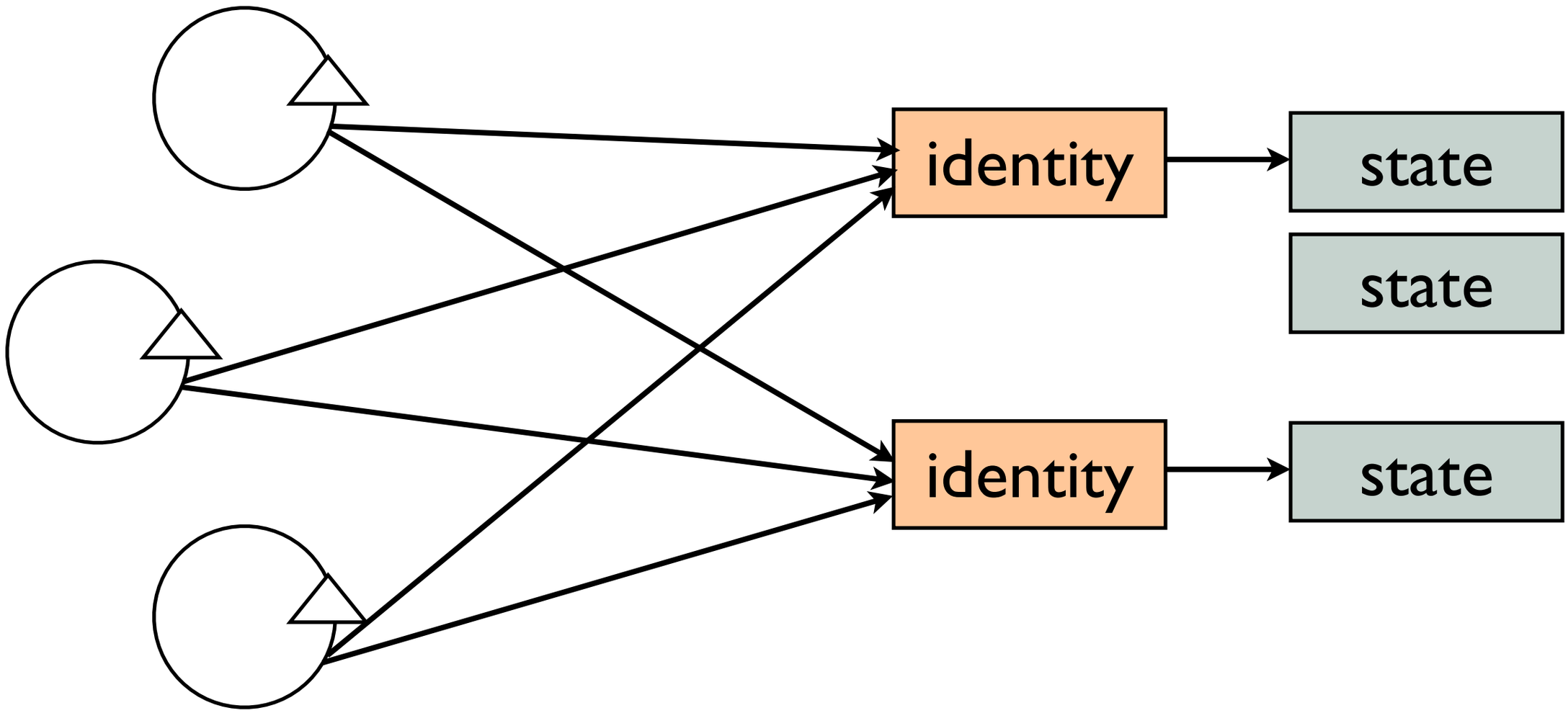




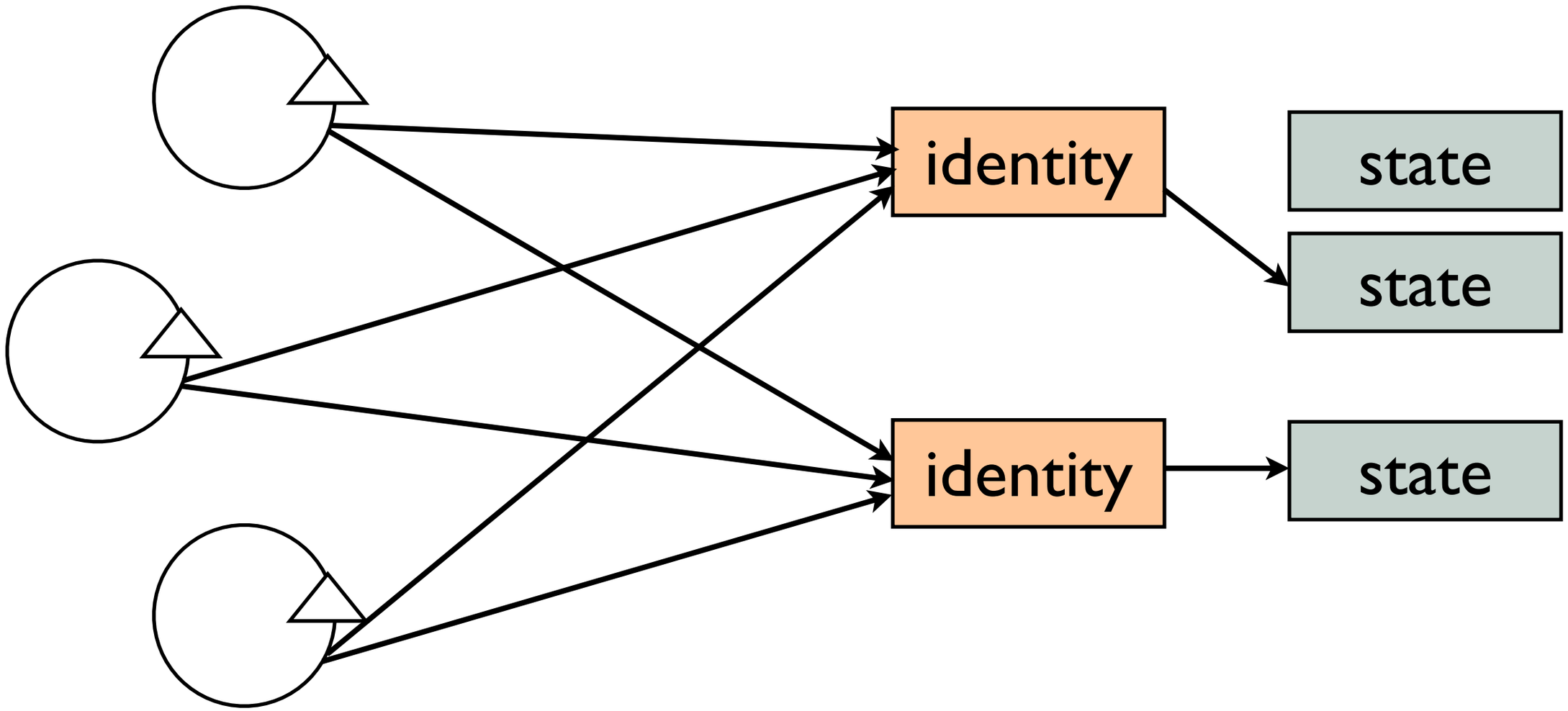
traditional oo



clojure



clojure



concurrency options

concurrency options

refs / stm

concurrency options

refs / stm

atoms

concurrency options

refs / stm

atoms

agents

concurrency options

refs / stm

atoms

agents

dynamic vars

concurrency options

refs / stm

atoms

agents

dynamic vars

locking / java.util.concurrent

refs and stm

threadsafe chat

identity

```
(def messages (ref ()))

(defn add-message [msg]
  (dosync (alter messages conj msg)))
```

scope a transaction

update fn

validate *updates*,

not objects

create a function

that checks every item...

```
(def validate-message-list
  (partial
    every?
    #(and (:sender %) (:text %))))
```

for some criteria

```
(def messages
  (ref
    ()
    :validator validate-message-list))
```

and associate fn with updates to a ref

atoms: uncoordinated updates

atom vs. ref

function	ref	atom
create	`ref`	`atom`
deref	`deref/@`	`deref/@`
update	`alter`	`swap!`
set	`ref-set`	`reset!`

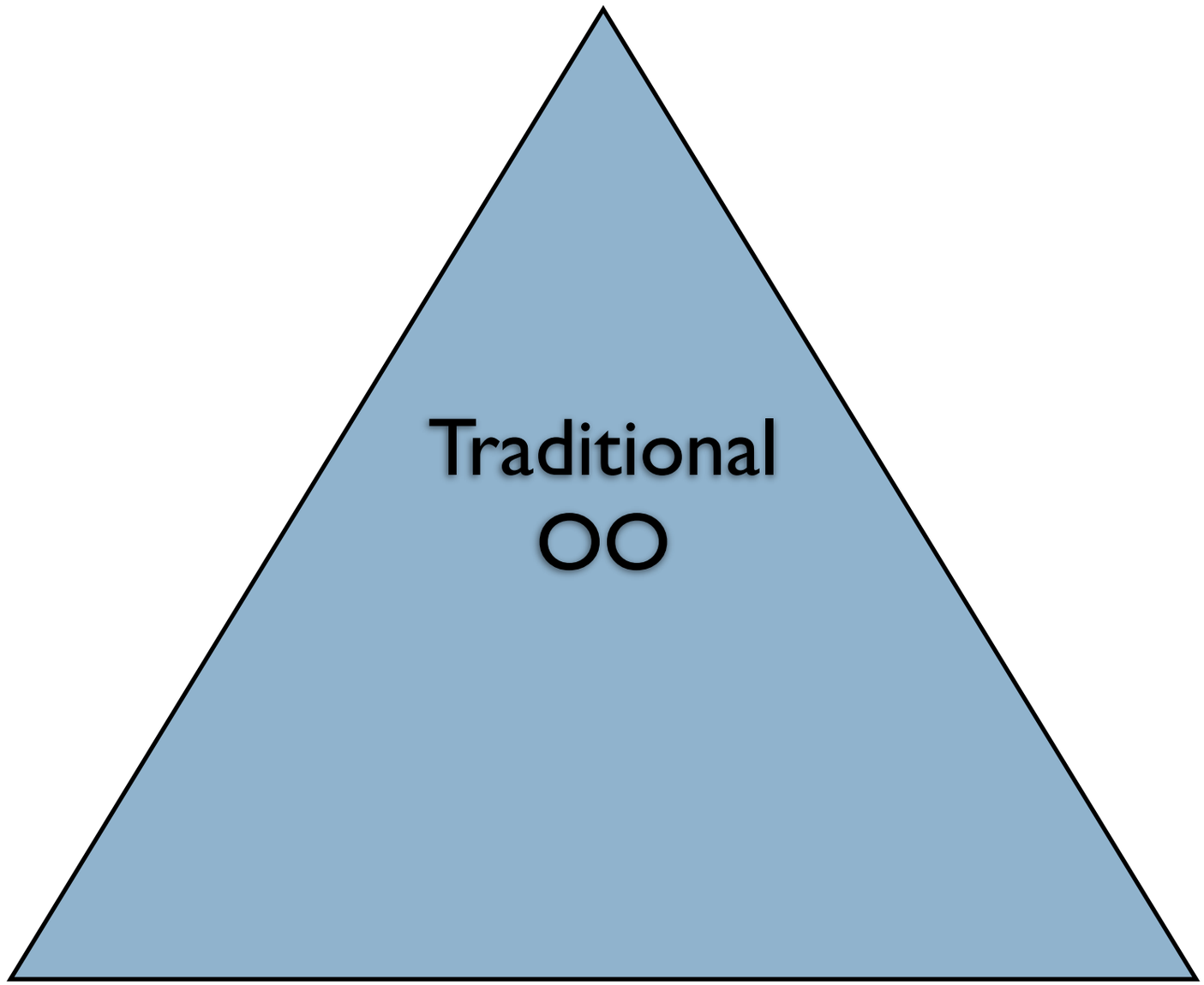
agents: asynchronous updates

sending from a tx

```
(defn add-message-with-backup [msg]
  (dosync
   (let [snapshot (alter messages conj msg)]
     (send-off backup-agent (fn [filename]
                              (spit filename snapshot)
                              filename))
     snapshot)))
```

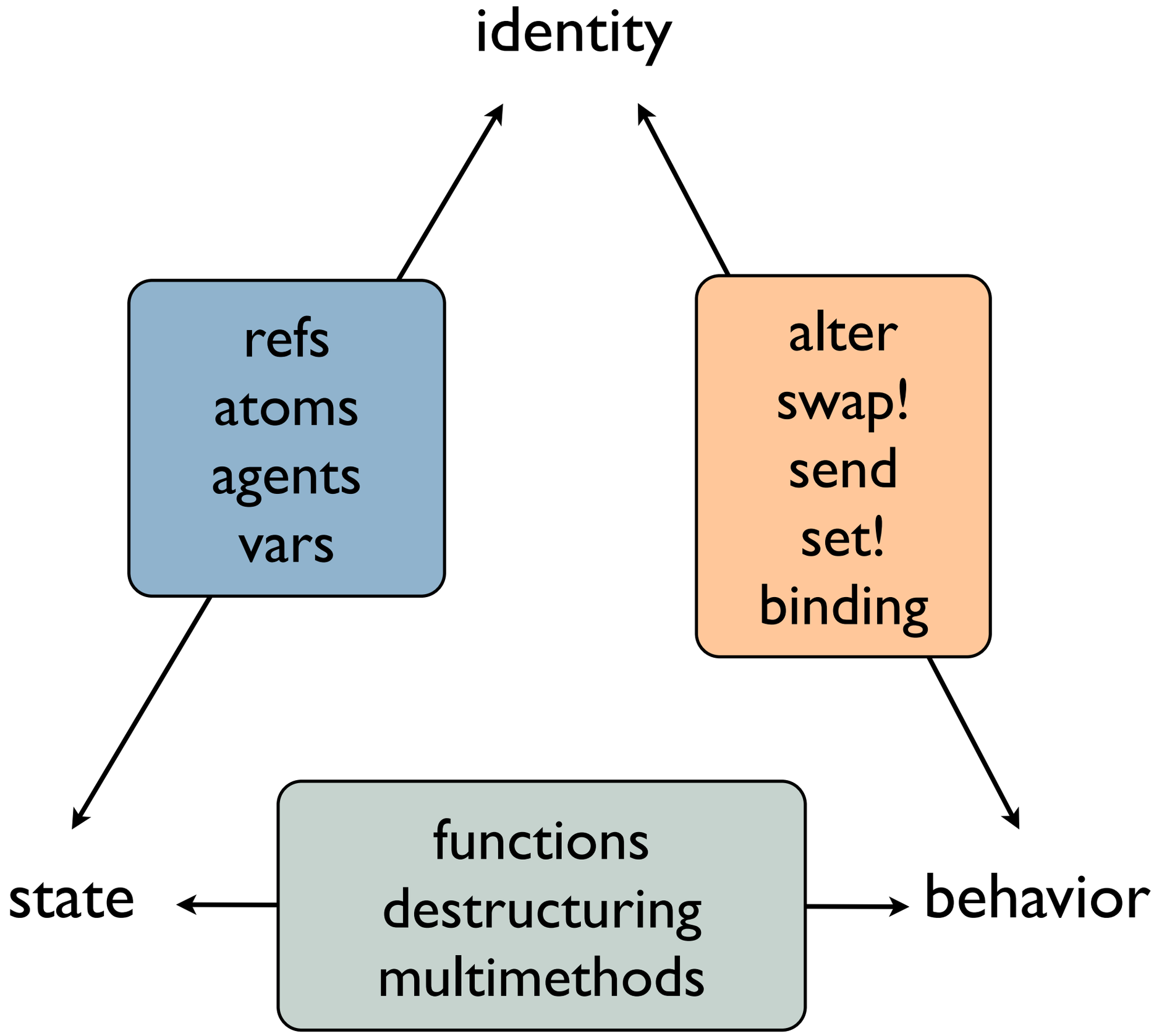
what about objects?

identity



state

behavior


identity
refs
atoms
agents
vars
alter
swap!
send
set!
binding
state
functions
destructuring
multimethods
behavior

Email: stu@thinkrelevance.com
Office: 919-442-3030
Twitter: twitter.com/stuarthalloway
Facebook: stuart.halloway
Github: stuarthalloway

Talks: http://blog.thinkrelevance.com/talks
Blog: http://blog.thinkrelevance.com

If you read the book, please post a review on Amazon:
http://bit.ly/12o06l

The Pragmatic Programmers

Programming Clojure

Stuart Halloway

Edited by Susannah Davidson Pfalzer

http://tinyurl.com/clojure