

closure

stuart halloway
<http://thinkrelevance.com>

1

functional

2

lisp

3

concurrency

4

embraces JVM

5

elegant

6

example:
refactor apache
commons isBlank

7

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;  
    }  
}
```

8

- 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;  
    }  
}
```

9

- 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;  
}
```

10

+ higher-order function

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

11

- corner cases

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

12

lispify

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

13

functional
is
simpler

14

	imperative	functional
functions	1	1
classes	1	0
internal exit points	2	0
variables	2	0
branches	3	0
boolean ops	1	0
function calls	3	2
total	13	3

15

java interop

16

java new

java	<code>new Widget("foo")</code>
clojure	<code>(new Widget "foo")</code>
clojure sugar	<code>(Widget. "red")</code>

17

access static members

java	<code>Math.PI</code>
clojure	<code>(. Math PI)</code>
clojure sugar	<code>Math/PI</code>

18

access instance members

java	<code>rnd.nextInt()</code>
clojure	<code>(. rnd nextInt)</code>
clojure sugar	<code>(.nextInt rnd)</code>

19

chaining access

java	<code>person.getAddress().getZipCode()</code>
clojure	<code>(. (. person getAddress) getZipCode)</code>
clojure sugar	<code>(.. person getAddress getZipCode)</code>

20

parenthesis count

java	<code>()()()()</code>
clojure	<code>()()()</code>

21

atomic data types

type	example	java equivalent
string	<code>"foo"</code>	String
character	<code>\f</code>	Character
regex	<code>#"fo*"</code>	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

22

sequences

23

literal sequences

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

24

first / rest / cons

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

25

take / drop

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

26

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
```

27

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")
```

28

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"
```

29

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
```

30

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]
```

31

infinite sequences

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

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

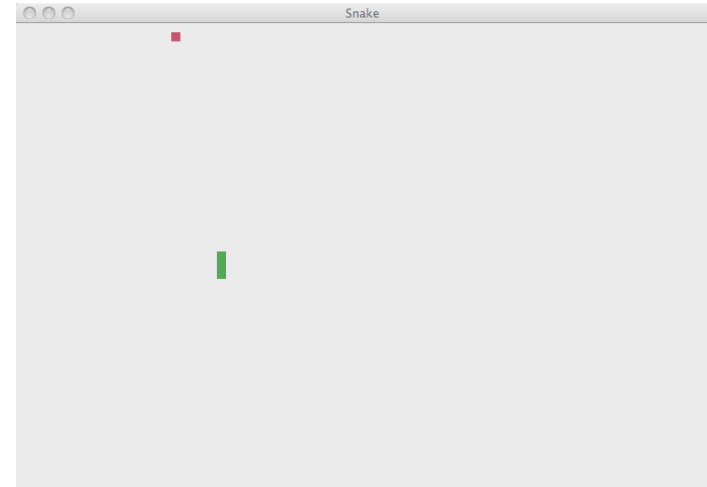
(cycle [1 2])
-> (1 2 1 2 1 ...)

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

32

game break!

33



Sample Code:

<http://github.com/stuarthalloway/programming-clojure>

34

assume snake
is a sequence
of points

first point is
head

```
(defn describe [snake]
  (println "head is " (first snake))
  (println "body is" (rest snake)))
```

rest is body

35

destructure
first element
into head

capture
remainder as a
sequence

```
(defn describe [[head & body]]
  (println "head is " head)
  (println "body is" body))
```

destructure remaining
elements into body

36

snake is more than location

```
(defn create-snake []  
  {:body (list [1 1])  
   :dir [1 0]  
   :type :snake  
   :color (Color. 15 160 70)})
```

37

2. nested destructure
to pull head and body from the
:body value

```
(defn describe [{[head & body] :body}]  
  (println "head is " head)  
  (println "body is" body))
```

1. destructure map,
looking up the :body

38

losing the game

```
(defn lose? [{[head & body] :body}]  
  (includes? body head))
```

39

example:
refactor apache
commons
indexOfAny

40

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
```

41

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;
}
```

42

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;
            }
        }
    }
}
```

43

- 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;
            }
        }
    }
}
```

44

+ when clause

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

45

+ comprehension

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

46

lispify!

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

47

functional
is
simpler

48

	imperative	functional
functions	1	1
classes	1	0
internal exit points	2	0
variables	3	0
branches	4	0
boolean ops	1	0
function calls*	6	3
total	18	4

49

functional
is
more general!

50

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)

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

51

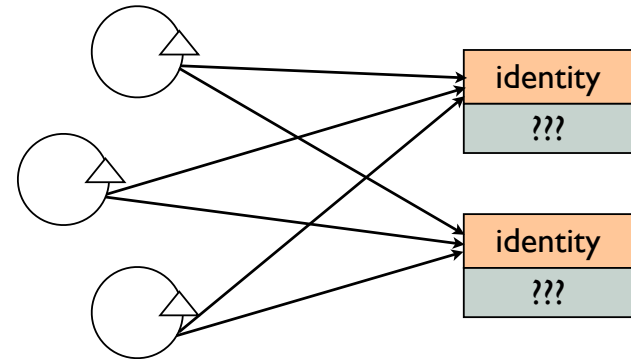
imperative	functional
searches strings	searches <i>any sequence</i>
matches characters	matches <i>any predicate</i>
returns first match	returns <i>lazy seq of all matches</i>

52

concurrency

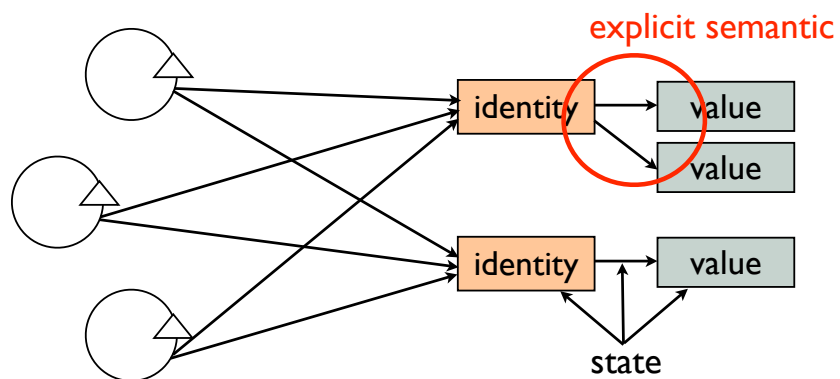
53

traditional oo



54

closure



55

terms

value: immutable data in a persistent data structure

identity: reference to a series of causally related values over time

state: the value of an identity at a time

56

concurrency options

	shared	isolated
synchronous/ coordinated	refs/stm	-
synchronous/ autonomous	atoms	vars
asynchronous/ autonomous	agents	-

57

refs and stm

58

ref example: chat

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

identity

scope a transaction

update fn

59

validate *updates*,
not objects

60

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

61

unified update model

function	ref	atom	agent
create	ref	atom	agent
deref	deref/@	deref/@	deref/@
update	alter	swap!	send send-off
set	ref-set	reset!	-

62

atoms:
uncoordinated
updates

63

atom example: brian's brain

```
(defn new-board
  "Create a new board with about half the cells set to :on."
  ([] (apply new-board dim-board))
  ([dim-x dim-y]
   (for [x (range dim-x)]
     (for [y (range dim-y)]
       (if (< 50 (rand-int 100)) :on :off))))))

(def status (atom {:iterations 0}))
(defn new-stage []
  (atom (new-board)))

(defn update-stage
  "Update the automaton (and associated metrics)."
  [stage]
  (swap! stage step)
  (swap! status update-in [:iterations] inc))
```

64

agents:
asynchronous
updates

65

tying agent to a tx

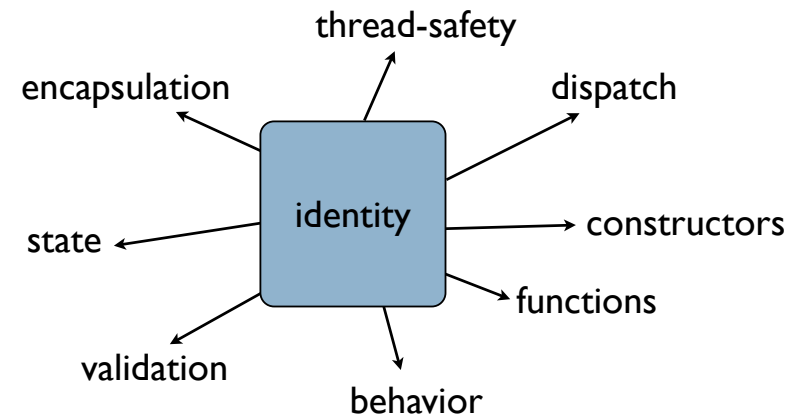
```
(defn add-message-with-backup [msg]
  (dosync
    (let [snapshot (alter messages conj msg)]
      (send-off backup-agent (fn [filename]
                              (spit filename snapshot)
                              filename))
      snapshot))))
```

66

what about
objects?

67

OO: identity drives everything



68

Clojure is a la carte

Programming Clojure



Stuart Halloway

Edited by Susannah Davidson Pfeizer

Email: stu@thinkrelevance.com
Office: 919-442-3030
Twitter: twitter.com/stuarthalloway
Facebook: [stuart.halloway](https://www.facebook.com/stuart.halloway)
Github: [stuarthalloway](https://github.com/stuarthalloway)

Talks: <http://blog.thinkrelevance.com/talks>
Blog: <http://blog.thinkrelevance.com>
Book: <http://tinyurl.com/clojure>