

CS 596 Functional Programming and Design  
Fall Semester, 2014  
Doc 13 Test, Exceptions, Multimethods  
Oct 16, 2014

Copyright ©, All rights reserved. 2014 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/openpub/>) license defines the copyright on this document.

# Unit Tests

# Testing

If it is not tested it does not work

Ralph Johnson

If it is not tested it does not exist

Kent Beck

# What is wrong with print statements?

Class assignment

5- 30 tests

1.5 year project with 10 programers

10,000 tests

How do you run the tests?

How do you see the results of the tests?

Can you run the tests each time you make a change

Print statements do not scale

# What to Test

Everything that could possibly break

Test values

- Inside valid range

- Outside valid range

- On the boundary between valid/invalid

GUIs are very hard to test

- Keep GUI layer very thin

- Unit test program behind the GUI, not the GUI

# Common Things Programs Handle Incorrectly

Adapted with permission from “A Short Catalog of Test Ideas” by Brian Marick,  
<http://www.testing.com/writings.html>

## Strings

Empty String

## Collections

Empty Collection

Collection with one element

Collection with duplicate elements

Collections with maximum possible size

## Numbers

Zero

The smallest number

Just below the smallest number

The largest number

Just above the largest number

# sdsu-rotate-test

```
(deftest sdsu-rotate-test
  (testing "sdsu-rotate"
    (are [n list answer] (= answer (sdsu-rotate n list))
      0 [] []
      1 [] []
      3 [1] [1]
      1 [1 2 3 4] [2 3 4 1]
      2 [1 2 3 4] [3 4 1 2]
      3 [1 2 3 4] [4 1 2 3]
      4 [1 2 3 4] [1 2 3 4]
      5 [1 2 3 4] [2 3 4 1]
      -1 [1 2 3 4] [4 1 2 3]
      -2 [1 2 3 4] [3 4 1 2]
      -3 [1 2 3 4] [2 3 4 1]
    )))
```

```
(deftest sdsu-roman-numeral-test
  (testing "sdsu-roman-numeral"
    (are [n answer] (= answer (sdsu-roman-numeral n))
      1 "I"
      4 "IV"
      30 "XXX"
      40 "XL"
      90 "XC"
      400 "CD"
      900 "CM"
      1901 "MCMXI"
    )))
```



:reloading (assignment3.core assignment3.core-test)

Testing assignment3.core-test

FAIL in (sdsu-dna-count-test) (core\_test.clj:50)

sdsu-dna-count

expected: (= {:T 3, :G 1, :C 2} (sdsu-dna-count "TGCTTC"))

actual: (not (= {:T 3, :C 2, :G 1} {:A 0, :T 3, :C 2, :G 1}))

FAIL in (sdsu-sum-test) (core\_test.clj:22)

sdsu-sum

expected: (= 233168 (sdsu-sum 3 5 1000))

actual: (not (= 233168 266333))

FAIL in (sdsu-sum-test) (core\_test.clj:22)

sdsu-sum

expected: (= 20 (sdsu-sum 2 4 10))

actual: (not (= 20 32))

FAIL in (sdsu-sum-test) (core\_test.clj:22)

sdsu-sum

expected: (= 45 (sdsu-sum 1 1 10))

actual: (not (= 45 90))

Ran 7 tests containing 39 assertions.

4 failures, 0 errors.

Failed 4 of 39 assertions

Finished at 13:08:14.537 (run time: 1.871s)

# Leiningen Projects Include Testing

Sets up requirements for tests

Clojure has testing framework

Similar to JUnit

```
lectureexample/  
  doc/  
  resources/  
  src/  
    lectureexample/  
      core.clj  
  test/  
    lectureexample/  
      core_test.clj  
LICENSE  
project.clj  
README.md
```

# Generated test file: core\_test.clj

```
(ns lectureexample.core-test
  (:require [clojure.test :refer :all]
            [lectureexample.core :refer :all]))
```

```
(deftest test-add-ten
  (testing "FIXME, I fail."
    (is (= 0 1))))
```

deftest - defines the test

testing - optional, label for the output

is - testing method

# are - Shortcut for multiple is

```
(deftest test-add-ten
  (is (= (add-ten [1 2 3] []) [11 12 13]))
  (is (= (add-ten [1] []) [11]))
  (is (= (add-ten [] []) []))
  (is (= (add-ten nil []) []))
```

```
(deftest test-add-ten
  (are [list result] (= (add-ten list []) result)
    [1 2 3] [11 12 13]
    [1] [11]
    [] []
    nil []))
```

# Light Table & Tests

Light Table does not run your tests for you :(

Will see two different ways to run the tests

# Semi-Manual

```
(ns lectureexample.core-test
  (:require [clojure.test :refer :all]
            [lectureexample.core :refer :all]))
```

```
(defn reload-tests
  []
  (use 'lectureexample.core :reload-all)
  (use 'lectureexample.core-test :reload-all)
  (run-tests 'lectureexample.core-test))
```

:reload-all  
reload definitions of your code

run-tests - runs the test

(reload-tests)

```
(deftest test-add-ten
  (is (= (add-ten [1 2 3] []) [11 12 13]))
  (is (= (add-ten [1] []) [11]))
  (is (= (add-ten [] []) []))
  (is (= 1 2))
  (is (thrown? clojure.lang.ArityException (add-ten [1 2]))))
```

# How to Run test Automatically

`lein-test-refresh`

Leiningen plug-in

Runs tests when your source code files change

Need to run Leiningen command

Need to configure project

# project.clj

```
(defproject lectureexample "0.1.0-SNAPSHOT"  
  :description "FIXME: write description"  
  :url "http://example.com/FIXME"  
  :license {:name "Eclipse Public License"  
           :url "http://www.eclipse.org/legal/epl-v10.html"}  
  :dependencies [[org.clojure/clojure "1.6.0"]  
                [org.clojure/tools.trace "0.7.8"]]  
  :main ^:skip-aot lectureexample.core  
  :target-path "target/%s"  
  :profiles {:uberjar {:aot :all}}  
  :plugins [[com.jakemccrary/lein-test-refresh "0.5.1"]])
```



# Starting lein-test-refresh

In terminal/command line

Go to project directory

```
cd lectureexample/
```

Run lein test-refresh

```
lein test-refresh
```

Every time you save a source file test-refresh reload code & runs test

# Some Java

# Accessing Static Methods & Fields

Static Fields

Class/fieldName

Math/PI

Float/MAX\_VALUE

Static Methods

(Class/methodName arg1 arg2 ...)

(Double/parseDouble "3.14159")

(Integer/toBinaryString 3)

# Accessing Java instance methods

`(.instanceMethod object arg1 ...)`

`(.toUpperCase "cat")`

`(.isEmpty [1 2 3])`

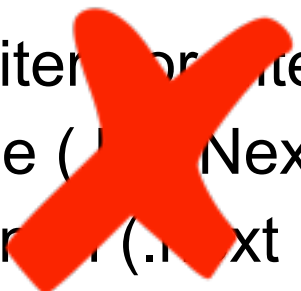
`(.size [1 2 3])`

`(.get [1 2 3] 1)`

# Examples

```
(defn decimal-to-hex [x]
  (-> x
    Integer/parseInt
    (Integer/toString 16)
    .toUpperCase))
```

```
(def iterator [1 2 3])
(while (next iterator)
  (print (.next iterator)))
```



# Exceptions

```
(defn as-int
  [s]
  (try
    (Integer/parseInt s)
    (catch NumberFormatException e
      (.printStackTrace e))
    (finally
      (println "Attempted to parse as integer: " s))))
```

# Raising an Exception

```
(throw (IllegalStateException. "I don't know what to do!"))
```

# Common Exceptions

`java.lang.IllegalArgumentException`

`java.lang.UnsupportedOperationException`

`java.lang.IllegalStateException`

`java.io.IOException`

Text claims that these handle 90% of cases where you need exceptions



# When to Use Exceptions?

Googles answer:

Exceptions should be used for situation where a certain method or function could not execute normally.

Does this mean nil nodes in a tree?

# Multimethods

# Example

```
(defmulti even-odd even?)
```

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

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

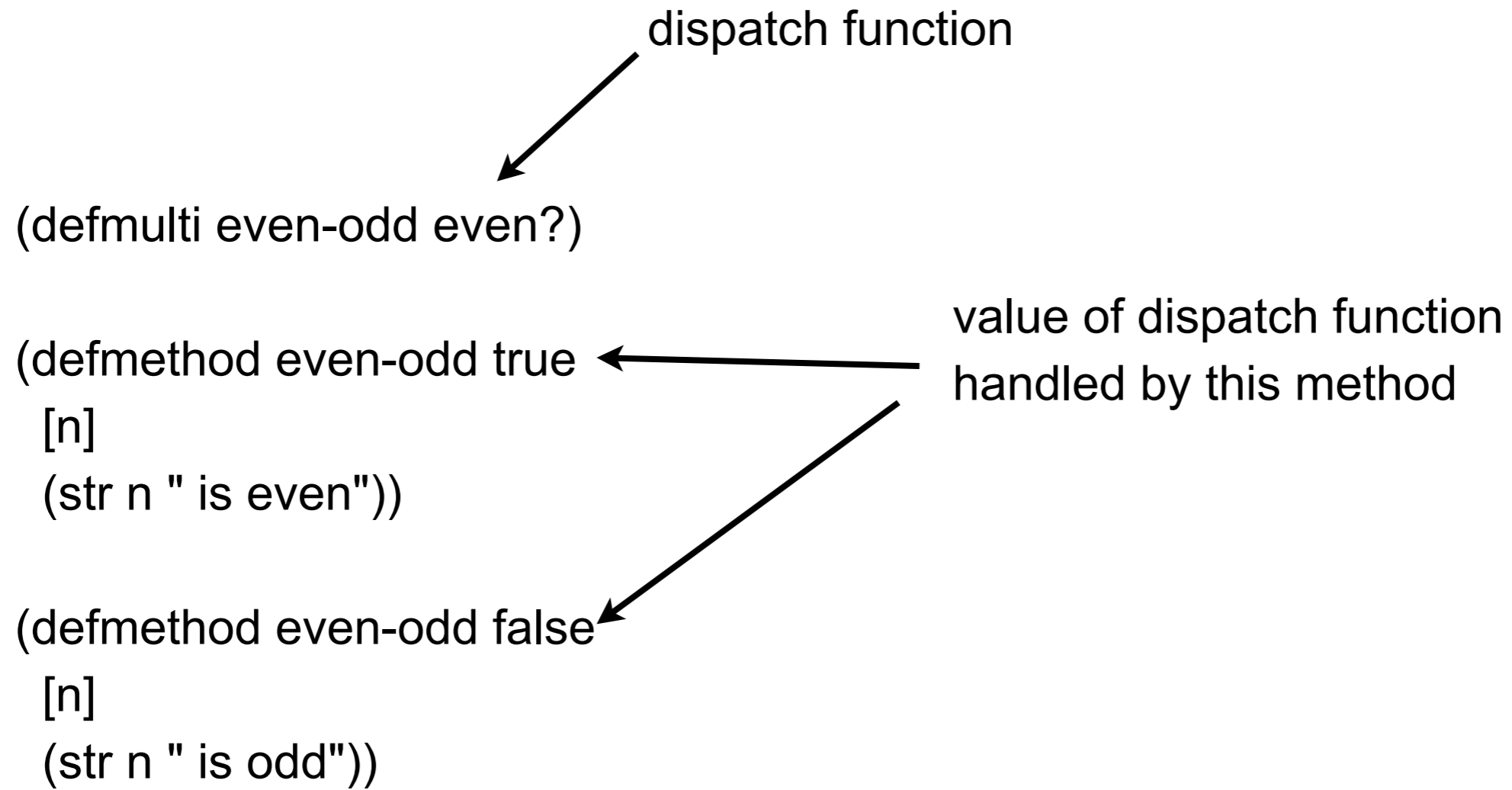
```
(even-odd 5)
```

5 is odd

```
(even-odd 4)
```

4 is even

# Example



# 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)	(types "ca")	"it is a string"
	(types 12)	"it is a Long"
(defmethod types java.lang.String [x] "it is a string")	(types 12.3)	"Don't know"
(defmethod types java.lang.Long [x] "it is a Long")		
(defmethod types :default [x] "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]  
  "medium")
```

```
(defmethod by-size :default  
  [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)
```

```
"medium"
```

# Vectors as Match

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

```
(by-size 2 90) "small-large"
```

```
(by-size 10 20) "other"
```

```
(defmethod by-size [:small :small]
```

```
  [x y]
```

```
  "small-small")
```

```
(defmethod by-size [:small :large]
```

```
  [x y]
```

```
  "small-large")
```

```
(defmethod by-size [:medium :medium]
```

```
  [x y]
```

```
  "medium-medium")
```

```
(defmethod by-size :default
```

```
  [x y]
```

```
  "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

# Some Solutions

# rotate

```
(defn sdsu-rotate
  [n sequ]
  {:pre [(integer? n) (or (seq? sequ) (vector? sequ) (nil? sequ))]}
  (let [sequ-len (count sequ)]
    (if (zero? sequ-len)
        sequ
        (if (neg? n)
            (sdsu-rotate (- sequ-len (mod (- n) sequ-len)) sequ)
            (concat (drop (mod n sequ-len) sequ)(take (mod n sequ-len) sequ)))))))
```

# rotate

```
(defn sdsu-rotate  
  [n xs]  
  (let [z (mod n (count xs))]  
        (concat (drop z xs) (take z xs))))
```

```
(defn sdsu-rotate  
  [n xs]  
  (apply concat (reverse (split-at (mod n (count xs)) xs))))
```

# Sum multiples of 3 & 5 less than 1000

```
(defn multiple-of-3-or-5? [n]
  (or (= 0 (mod n 3))
      (= 0 (mod n 5))))
```

```
(apply + (filter multiple-of-3-or-5? (range 1000)))
```

```
(defn multiple-of-3-or-5? [n]
  (or (zero? (rem n 3))
      (zero? (rem n 5))))
```

```
(reduce + (filter multiple-of-3-or-5? (range 1000)))
```

# Using Lazy

```
(defn sdsu-sum  
  [n1 n2 max]  
  (reduce + (distinct (concat (range n1 max n1) (range n2 max n2))))))
```



# Palindrome

```
(defn palindrome?  
  [n]  
  (let [string-n (str n)]  
    (= (seq string-n) (reverse string-n))))
```

```
(defn- generate-numbers  
  [digits]  
  
  (for [x (range (int (Math/pow 10 digits)) (Math/pow 10 (dec digits)) -1 )  
        y (range (int (Math/pow 10 digits)) (dec x) -1 )]  
    (* x y)))
```

```
(defn sdsu-palindrome  
  [number]  
  (let [numbers (generate-numbers number)]  
    (reduce max (filter palindrome? numbers))))
```

# DNA

```
(defn sdsu-dna-count
  [s]
  (when (string? s)
    (into {}
      (for [[k v] (frequencies s)]
        [(keyword (str k)) v])))))
```

# digits

```
(defn sdsu-digits
  [n b]
  {:pre [(integer? n) (>= n 0) (integer? b) (pos? b)]}
  (if (zero? n)
      [0]
      ((fn acc
         [number base-b-representation]
         (if (zero? number)
             (vec base-b-representation)
             (acc (int (/ number b)) (conj base-b-representation (mod number b)))))) n ())))
```

```

(defn sdsu-roman-numeral
  [n]
  {:pre [(integer? n) (< n 4000) (pos? n)]}
  ((fn acc [
    remainder      ; Remaining (unrepresented) decimal part of the number
    roman-rep      ; Roman numeral representation built so far
  ]
    (cond
      (>= remainder 1000) (acc (- remainder 1000) (str roman-rep "M" ))
      (>= remainder 900)  (acc (- remainder 900)  (str roman-rep "CM"))
      (>= remainder 500)  (acc (- remainder 500)  (str roman-rep "D" ))
      (>= remainder 400)  (acc (- remainder 400)  (str roman-rep "CD"))
      (>= remainder 100)  (acc (- remainder 100)  (str roman-rep "C" ))
      (>= remainder 90)   (acc (- remainder 90)   (str roman-rep "XC"))
      (>= remainder 50)   (acc (- remainder 50)   (str roman-rep "L" ))
      (>= remainder 40)   (acc (- remainder 40)   (str roman-rep "XL"))
      (>= remainder 10)   (acc (- remainder 10)   (str roman-rep "X" ))
      (>= remainder 9)    (acc (- remainder 9)    (str roman-rep "IX"))
      (>= remainder 5)    (acc (- remainder 5)    (str roman-rep "V" ))
      (>= remainder 4)    (acc (- remainder 4)    (str roman-rep "IV"))
      (>= remainder 1)    (acc (- remainder 1)    (str roman-rep "I" ))
      :else roman-rep)) n ""))

```

# References

vars, atoms, agents, refs

# Reference Type Basics

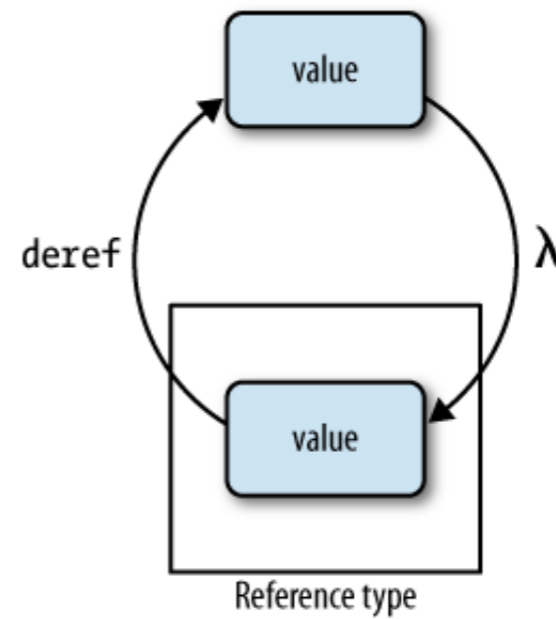
var, ref, atom, agent

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

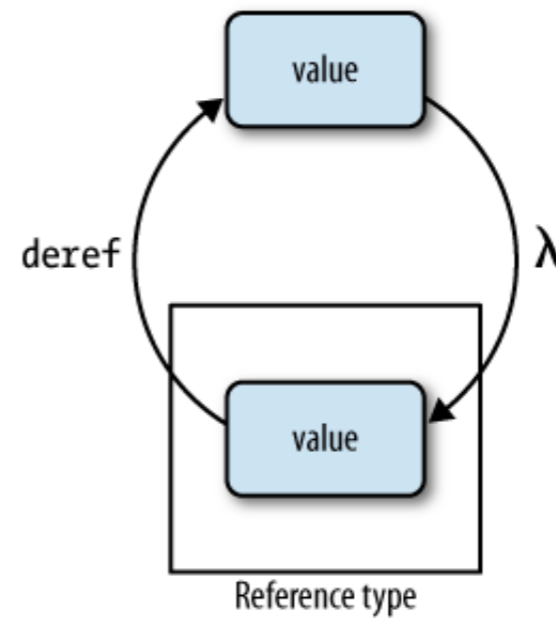
- Can have meta data

- 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

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/cat 4 10
```

# Validator

```
(def cat 4)
```

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

```
(def cat 9)
```

```
(def cat 20)                ;;exception
```