CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 4 Data & Functions
Sep 9, 2014

Maps (Hash Table)

Key-value map

Keys - any value

Values - any value

Fast insert & find

Very common

```
{:first-name "Roger"
 :last-name "Whitney" }
```

```
{:first-name "Roger",
 :last-name "Whitney" }
```

```
{:name {:first "Roger" :last "Whitney" }
 :phone-numbers
    ["111-2222" "222-3333"]}
```

```
{ "a" 1, 2 "b", [4 3] :me}
```

```
{  }
```

Maps (Hash Table)

(get {:a 1} :a)	1
({:a 1} :a)	1
(:a {:a 1})	1
({2 "b"} 2)	"b"
(2 {2 "b"})	Error
(conj {:a 1 :b 2} {:a 3} {:c 4})	{:c 4, :a 3, :b 2}
(merge {:a 1 :b 2} {:a 3 :c 4})	{:c 4, :a 3, :b 2}
(assoc {:a 1 :b 2} :a 3 :c 4)	{:c 4, :a 3, :b 2}

Lists

Linked List

Fast insert & remove at front

'(1 2 3)

'("cat" {:a 1})

'(+ 1 2)

Lists

(list 8 4 2)	(8 4 2)
(nth '("a" "b" "c") 2)	"c"
('("a" "b" "c") 2)	Error
(.indexOf '("a" "b" "c") "b")	1
(peek '("a" "b" "c"))	"a"
(pop '("a" "b" "c"))	("b" "c")
(conj '(1 2 3) 4)	(4 1 2 3)
(class '(1))	clojure.lang.PersistentList

Naming Conventions

Clojure	Java
all-lower-case words-separated-by-hyphen	camelCase

Why the Single Quote

'(+ 1 2) verses (+ 1 2)

All Clojure programs are just lists

Reader/interpreter/compiler evaluates all lists

Single quote turns off evaluation of the list

Homoiconicity - Code-as-Data

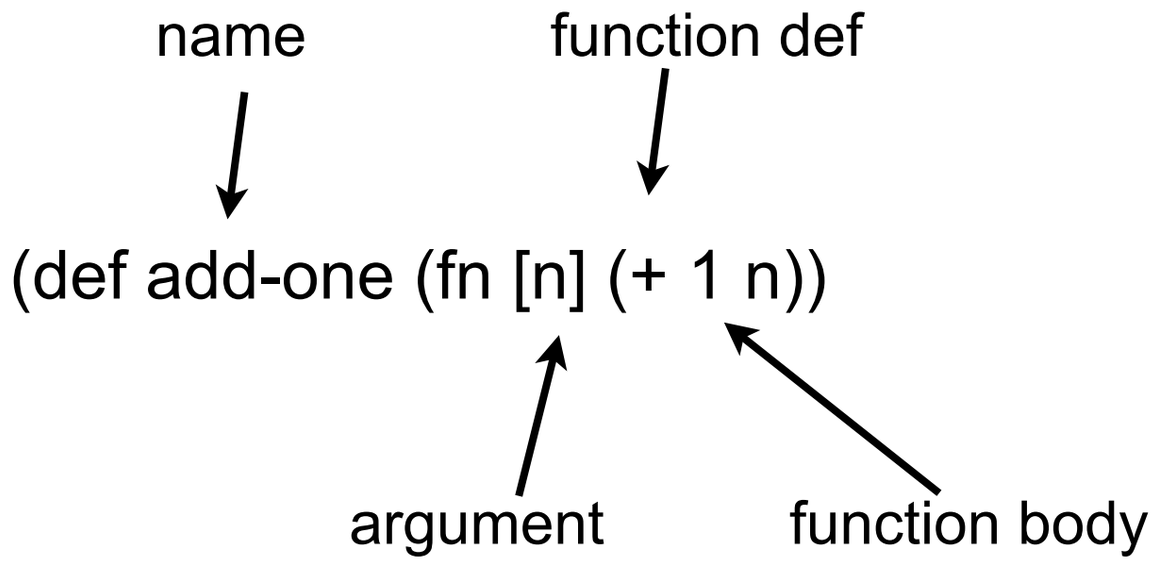
Clojure programs are represented by Clojure data structures

List structure is the Clojure syntax

Makes it easy for Clojure programs to modify Clojure programs

Macros

Defining a function



(add-one 5)

Defining a function - Compact version

```
(def add-one (fn [n] (+ 1 n)))
```

```
(defn add-one
    [n]
    (+ 1 n))
```

```
(add-one 5)
```

Valid function names

Function definitions are just Clojure data structures

Function names are just symbols

So any valid symbol can be used as a function name

```
(defn பன்னிரெண்டு-சேர்க்க
  [n]
  (+ 12 n))
```

Multiple Arguments

```
(defn sum
      [a b c d]
  (+ a b c d))
```

```
(defn foo-bar
      [a b]
  (if (< a b)
        "smaller"
        (+ a b)))
```

Defn Format

```
(defn function-name
    "Doc string"
    [arg1 arg2 … argN]
    (form1)
    (form2)
    …
    (formN))
```


We will see later how to add metadata to a function

Doc Strings

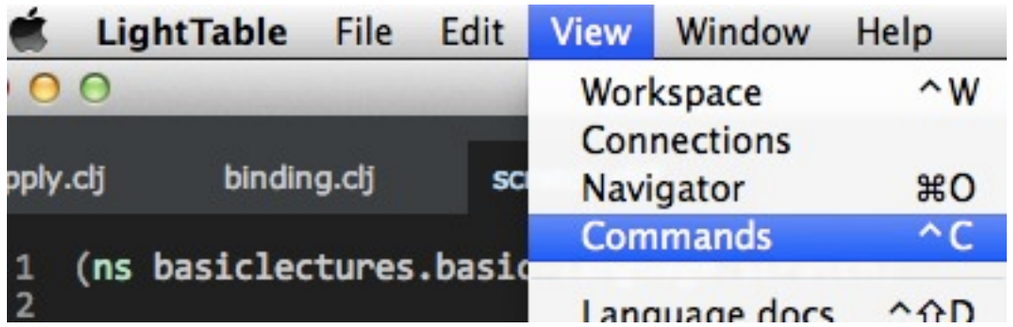
(doc pop)
(clojure.repl/doc pop)

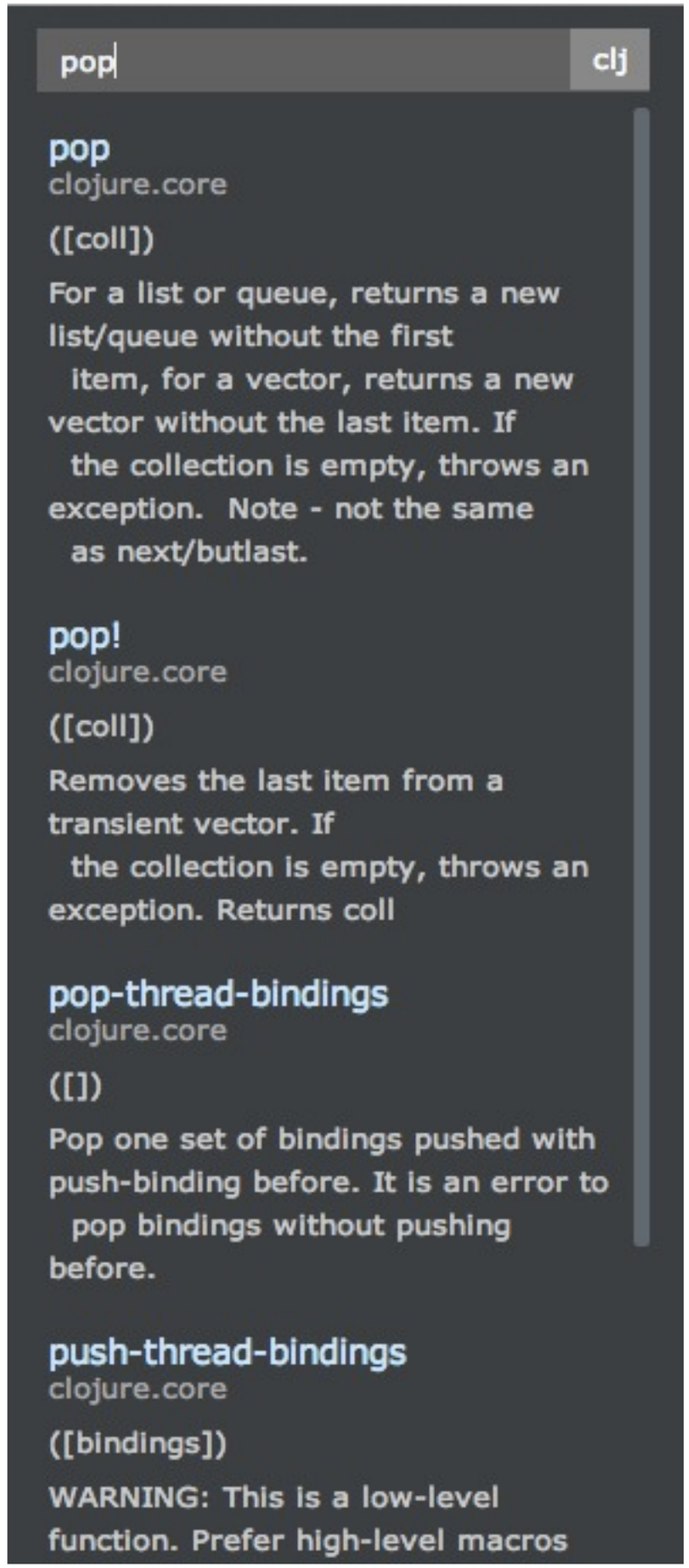
Prints doc string in REPL

(find-doc "pop")
(clojure.repl/find-doc "pop"

Finds functions related to "pop"

find-doc in Light Table



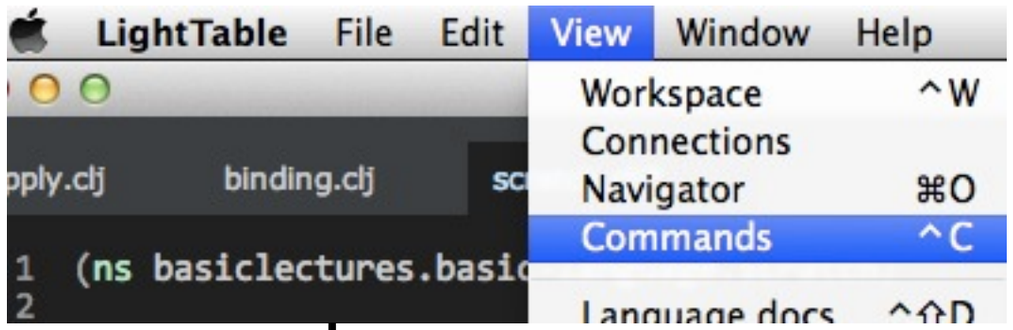


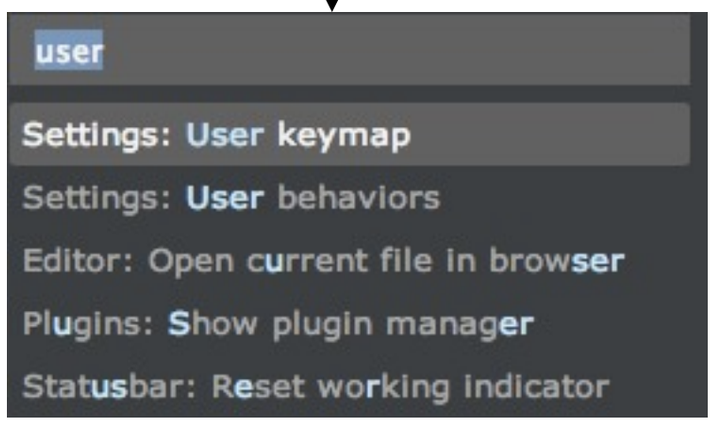
doc in Light Table

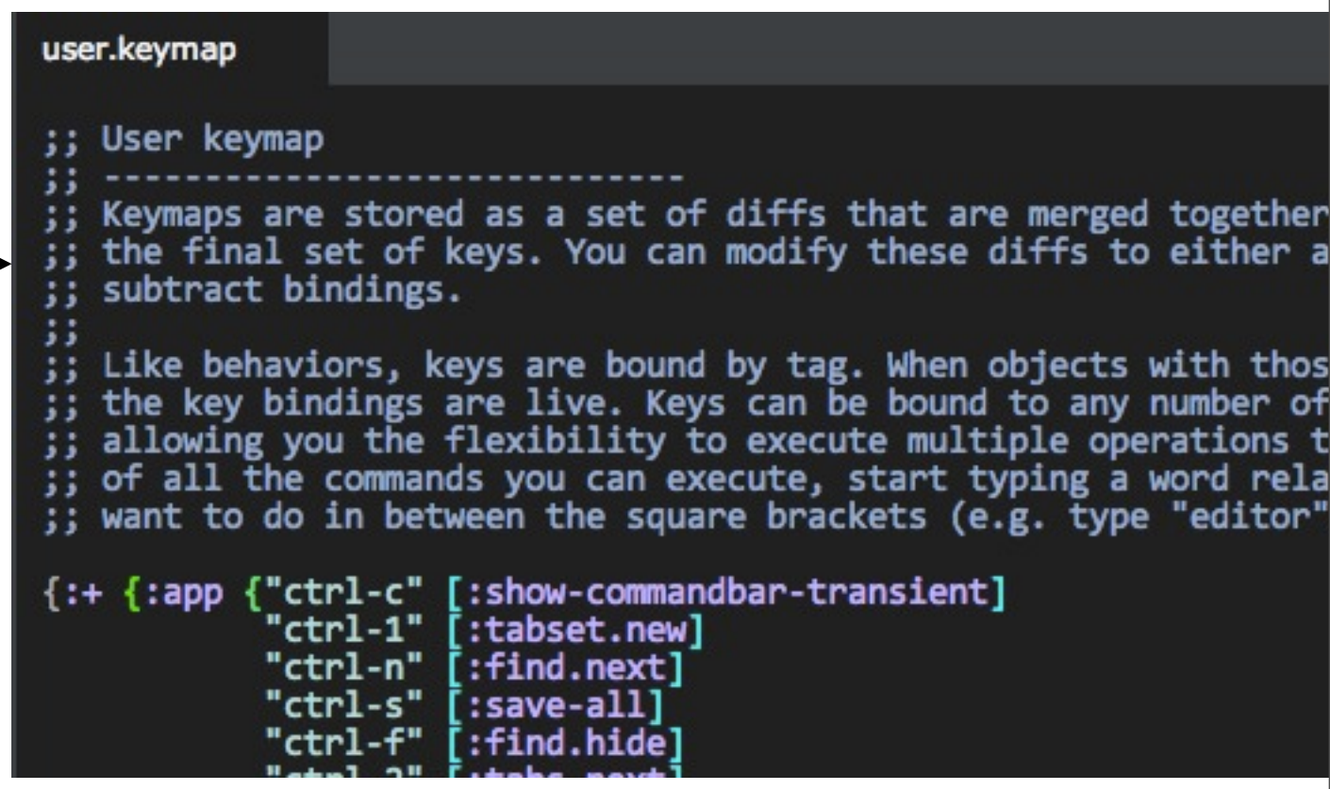
```
3
4 (pop [1234])
```

```
pop
clojure.core

([coll])

For a list or queue, returns a new list/queue without the first
  item, for a vector, returns a new vector without the last item. If
  the collection is empty, throws an exception.  Note - not the same
  as next/butlast.
```

Configuring Light Table







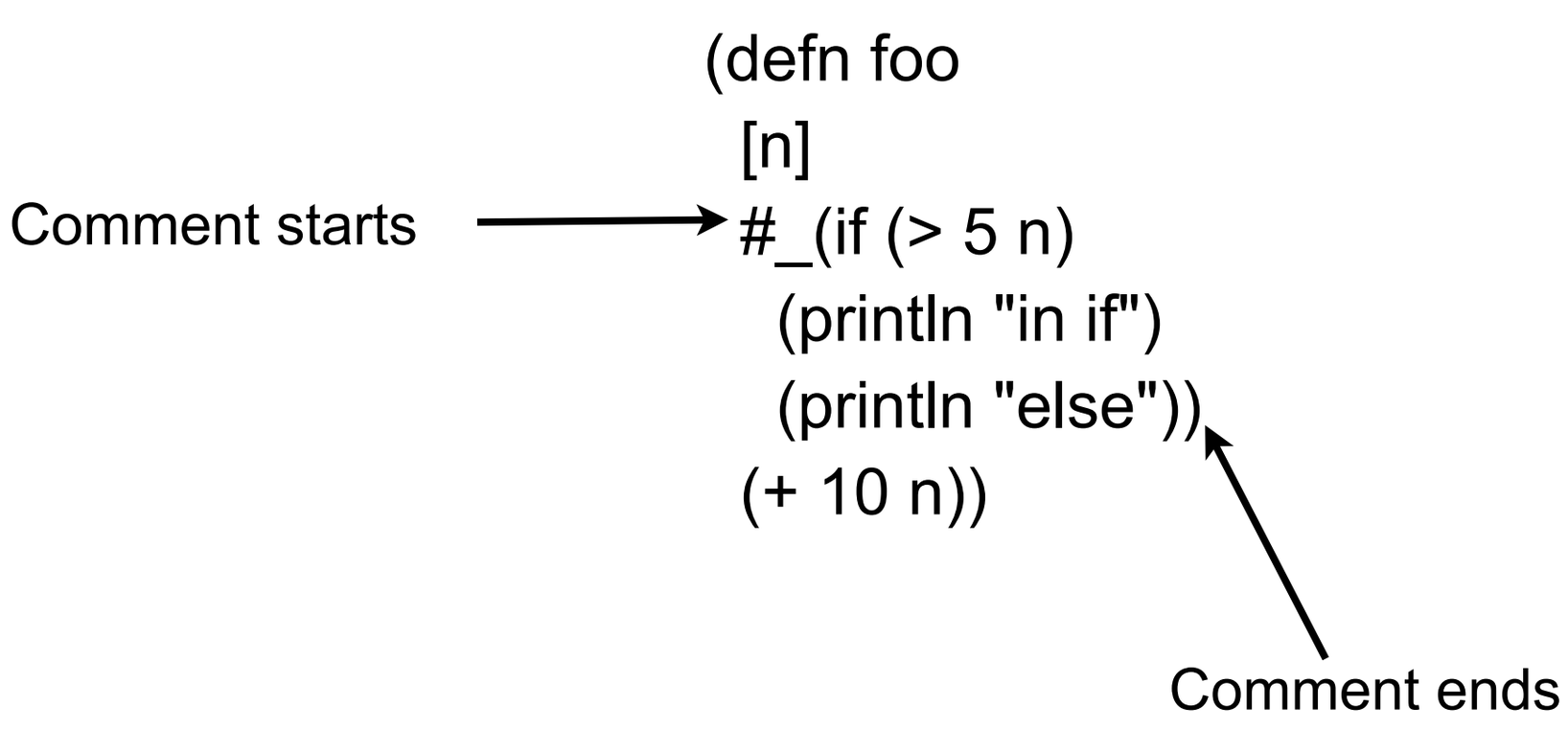
Some Useful keymaps

```
{:+ {:app {"ctrl-c" [:show-commandbar-transient]
           "ctrl-1" [:tabset.new]
           "ctrl-n" [:find.next]
           "ctrl-s" [:save-all]
           "ctrl-f" [:find.hide]
           "ctrl-2" [:tabs.next]
           "ctrl-i" [:instarepl]
           "ctrl-w" [:workspace.show]
           "ctrl-z" [:window.zoom-in]
           "ctrl-shift-z" [:window.zoom-out]
           "ctrl-m"  [:window.maximize]
           "ctrl-t" [:toggle-console]}

      :editor {"ctrl-r" [:clear-inline-results]
               "ctrl-d" [:editor.doc.toggle]
               "ctrl-a" [:paredit.select.parent]
               "ctrl-l" [:paredit.grow.left]
               "ctrl-;" [:paredit.shrink.left]}}}
```

Comments

; a semi-colon starts a comment that goes to end of the line

#_ when prepended to a form makes the entire form a comment



Explain This

```
(defn foo
  [n]
  "How does this work? Not a compile error."
  (if (> 5 n)
    (println "in if")
    (println "else"))
  "This is not a doc comment"
  (+ 10 n))
```

And This?

```
(defn foo
  [n]
  (if (> 5 n)
    "What happens now?"
    (println "in if")
    (println "else"))
  "This is not a doc comment"
  (+ 10 n))
```

Recall

```
(defn function-name
    "Doc string"
    [arg1 arg2 … argN]
    (form1)
    (form2)
    …
    (formN))
```

Clojure Form

Clojure expression

symbols
keywords
literals
lists
vectors
maps
sets

```
(defn foo
  [n]
  "How does this work? Not a compile error."
  (if (> 5 n)
    (println "in if")
    (println "else"))
  "This is not a doc comment"
  (+ 10 n))
```

Anonymous Function - Lambda

Function not bound to symbol

(fn [args] (form1) (form2)…(formn))

(fn [a b] (< (first a) (first b)))

((fn [a b] (< (first a) (first b))) [2 3] [5])

```
((fn [a b]
     (println a  b)
     (< (first a) (first b))) [2 3] [5])
```

Short Syntax for Lambda

```
(fn [a b] (< (first a) (first b)))
```

```
#(< (first %1) (first %2))
```

%n -> n'th argument

```
#(+ 2 %)
```

if only one argument can use %

Passing Functions as Arguments

```
(sort < [3 1 2])

(sort > [3 1 2])

(sort (fn [a b] (< a b))  [3 1 2])

(sort #(< %1 %2)  [3 1 2])

(sort (fn [a b] (compare (str a) (str b))) [ 4 3 16])

(sort  #(compare (str %1)  (str %2))  [4 3 16])
```

Closure

function + reference to its environment

```
(defn adder
  [n]
  #(+ n %))

(def add-5 (adder 5))

(add-5 10)                    Returns 15
```

OO data & Functional Data

Person

- First name
- Last name
- age
- List of phone numbers

Phone Number

- Number
- Type - mobile, work, home, etc

PhoneNumber

```
public class PhoneNumber {
    private String number;
    private String type;

    public PhoneNumber(String type, String number) {
        this.type = type;
        this.number = number;
    }

    public String getNumber()  { return number; }

    boolean isMobile() { type.equals("moblie");  }
    etc.
}
```

Person Class

```
public class Person {
    private int age;
    private String firstName;
    private String lastName;
    private ArrayList phoneNumbers;

    public Person(String first,String last, int age) {
        this.firstName = first;
        this.lastName = last;
        this.age = age;
        phoneNumbers = new ArrayList();
    }

    public int age() { return age; }
    public void age(int newAge) { age = newAge;}

    etc.
```

Sample Use

```
Person example = new Person("Sachin", "Tendulkar", 40);

int lastYearsAge = example.age();
example.age(41);
```

age gives access to the age value in a person

age is like a key in a hash table

Converting Objects to Clojure data

Class	Map
Field name	keyword as key in map
new Person("Sachin", "Tendulkar", 40);	{:first-name "Sachin" :last-name "Tendulkar" :age 40 :phone-numbers {}}

Some Functions

```
(defn make-person
  [first-name last-name age]
  {:first-name first-name
   :last-name last-name
   :age age
   :phone-numbers {}})
```

```
(defn increase-age
  [person-map]
  (update-in person-map [:age] inc))
```

```
(defn add-number
  [person-map phone-type number]
  (assoc-in person-map [:phone-numbers phone-type] number ))
```

Examples

```
(def test-person (make-person "Sachin" "Tendulkar" 40))
```

```
(add-number test-person :mobile "619-111-2222")

        {:first-name "Sachin", :last-name "Tendulkar", :age 40,
        :phone-numbers {:mobile "619-111-2222"}}
```

```
(increase-age test-person)

        {:first-name "Sachin", :last-name "Tendulkar", :age 41,
         :phone-numbers {}}
```

Read from inside out

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

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)
(/ 8 2)
(- 4 1)
```

-> Example

```
(def c 5)

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

```
(+ 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"} )
```


From Clojure Docs

-> 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)

(/ 2)

(- 1))

(+ 3 c)

(/ 2 **8**)

(- 1 **1/4**)

as-> Allow Threading in different locations

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

bind 5 to c

```
(+ 3 5)
(/ 8 2)
(- 4 1)
```

bind 8 to c

bind 4 to c

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

(def foo "hi")

foo & bar are symbols

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

They are bound to values

Expession	Evaluated Result
foo	"hi"
'foo	foo
bar	fn
(bar 12)	13

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
```