CS 420 Advanced Programming Languages
Fall Semester, 2022
Doc 15 Clojure Lists, Battleship & Functions
Oct 6, 2022

Copyright ©, All rights reserved. 2022 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

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

Why Does the Parenthesis Come First?

(max 2 4 1) verses max(2, 4, 1)

All Clojure (and Lisp) programs are valid Clojure (Lisp) data structures

```
(defn nthfirst
  "Drop the last n elements"
  [coll n]
  (-> coll
      reverse
      (nthrest n)
      reverse))
```

Why is this Important?

Clojure & Lisp programs can generate code and run the new code

If a program is to learn, it needs to change

Lisp-based languages allow programs to change their code

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

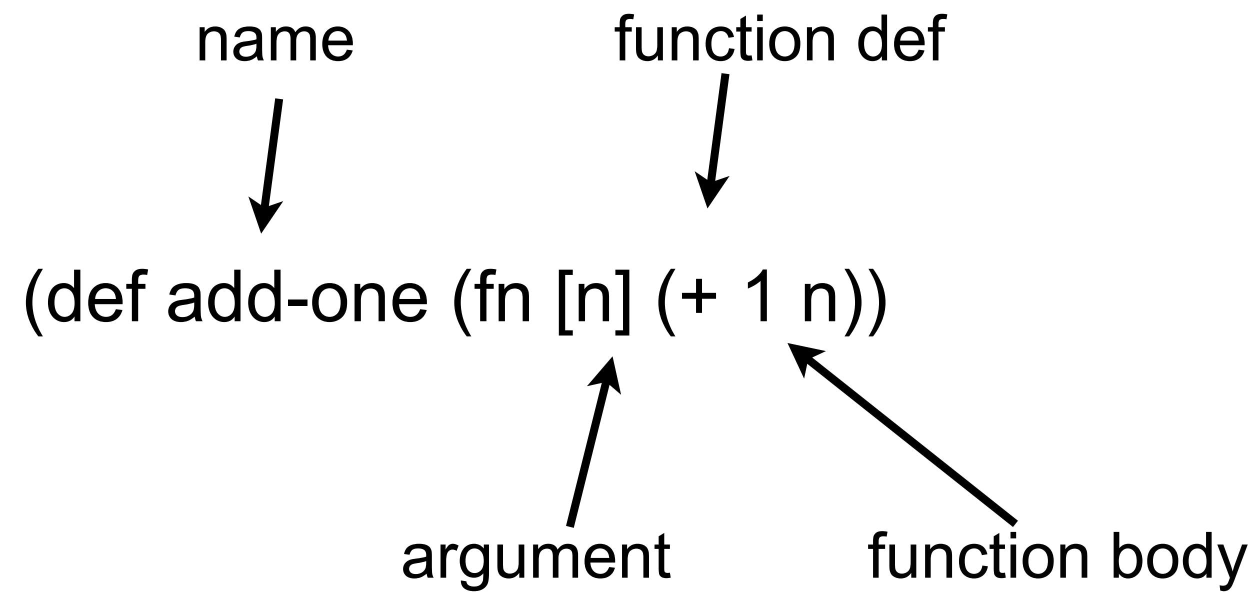
Clojure programs are represented by Clojure data structures

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))
```

defn Format

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

Doc Strings

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

Prints doc string in REPL

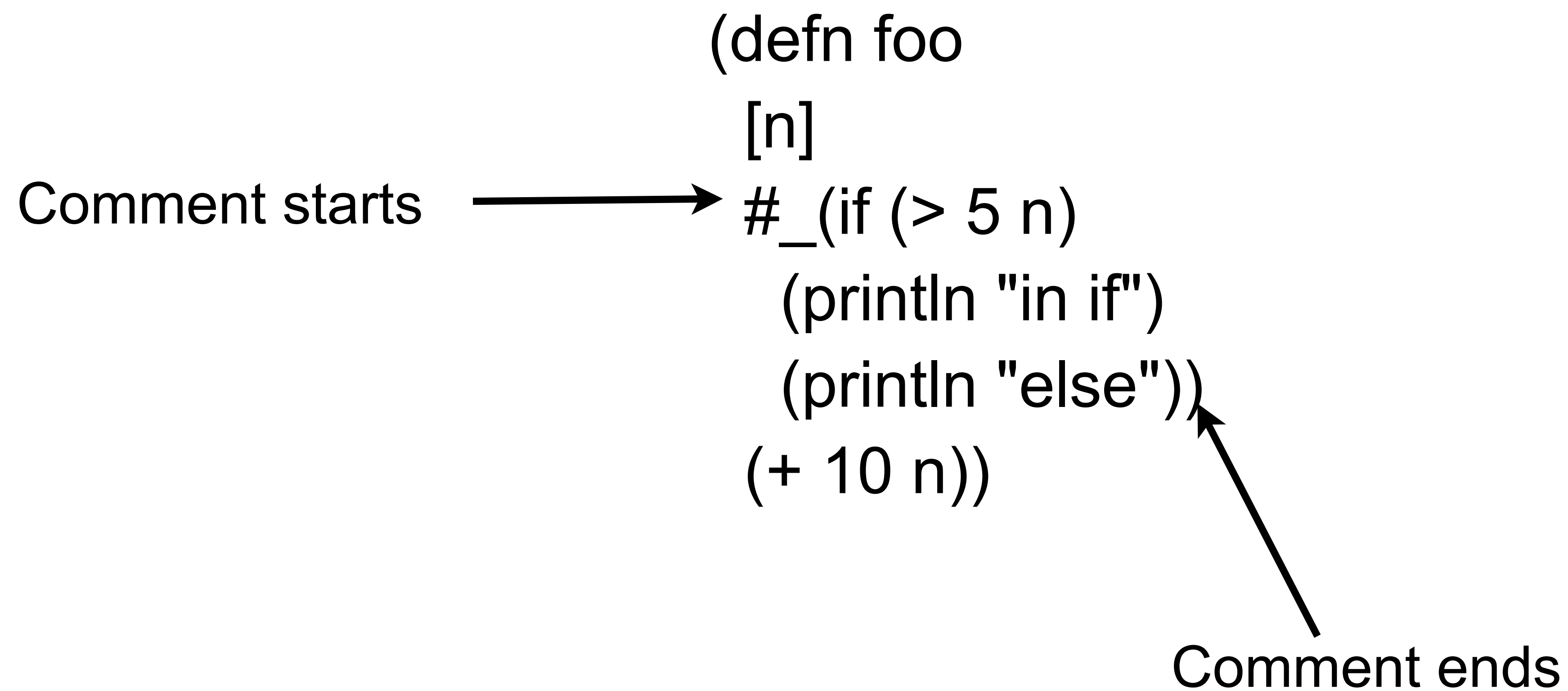
(find-doc "pop")
(clojure.repl/find-doc "pop"

Finds functions related to "pop"

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))
```

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

Battleship Example

The Problem

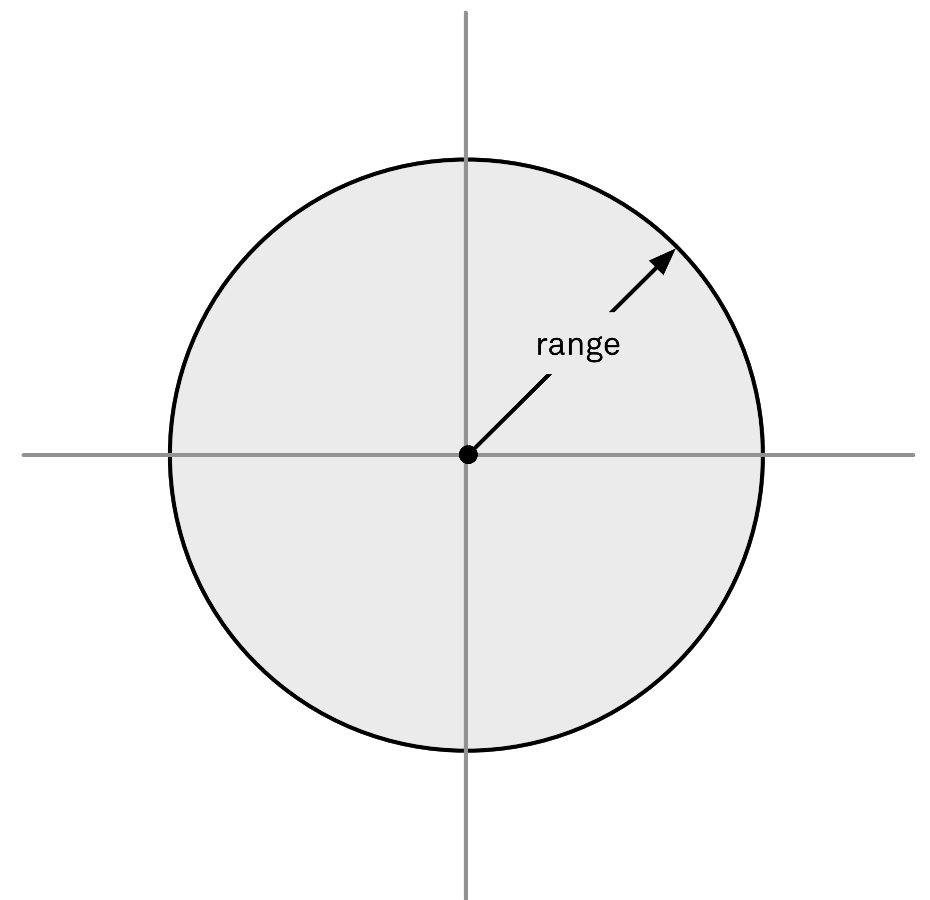
Context - Writing a battleship game

Need a function that determines

- Is an enemy ship within range of our ships weapon
- But weapon has a blast area so cannot use weapon if
  - Enemy ship is to close to us or other friendly ships

First Pass

Assume we are at origin   Point - [x y]
Given a point & range
Is point within range




```
(defn in-range-1
  [position range]
  (let [pos-x (first position)
        pos-y (last position)
        target-distance (Math/sqrt (+ (* pos-x pos-x) (* pos-y pos-y)))]
    (< target-distance range)))
```

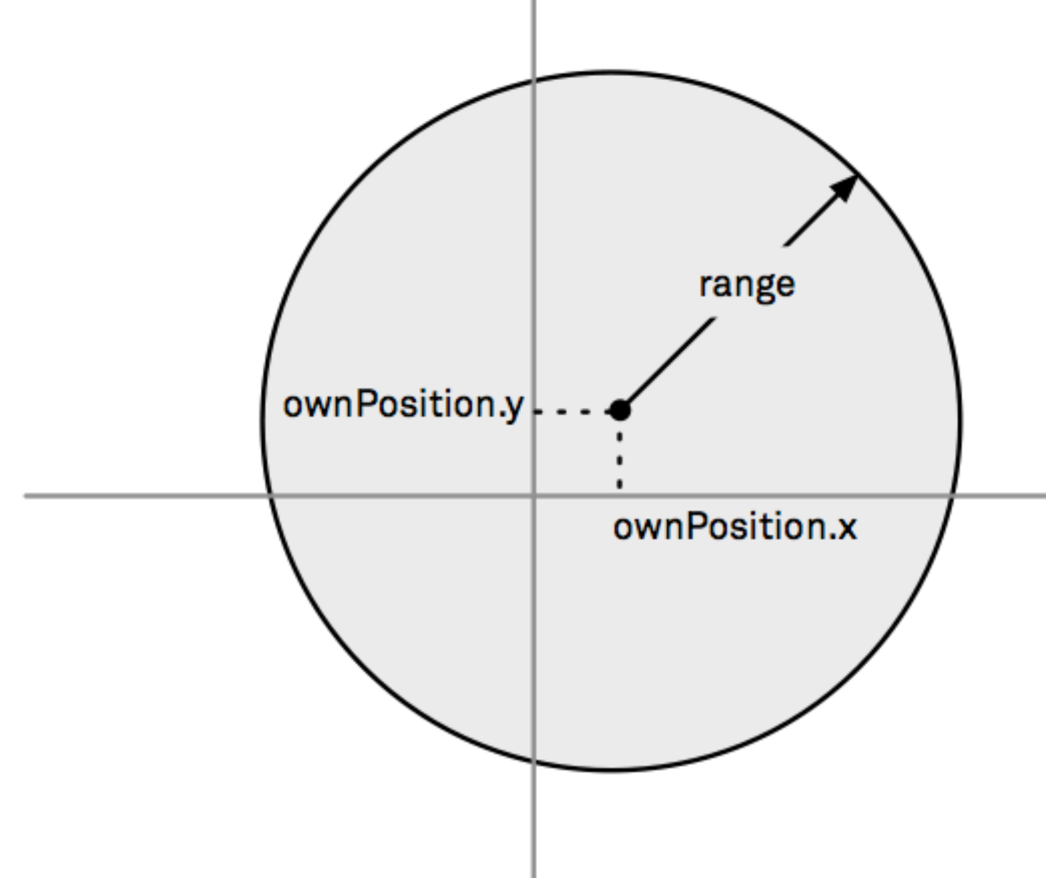
```
(in-range-1 [1 1] 1)        false

(in-range-1 [1 1] 2)        true
```

Second Pass

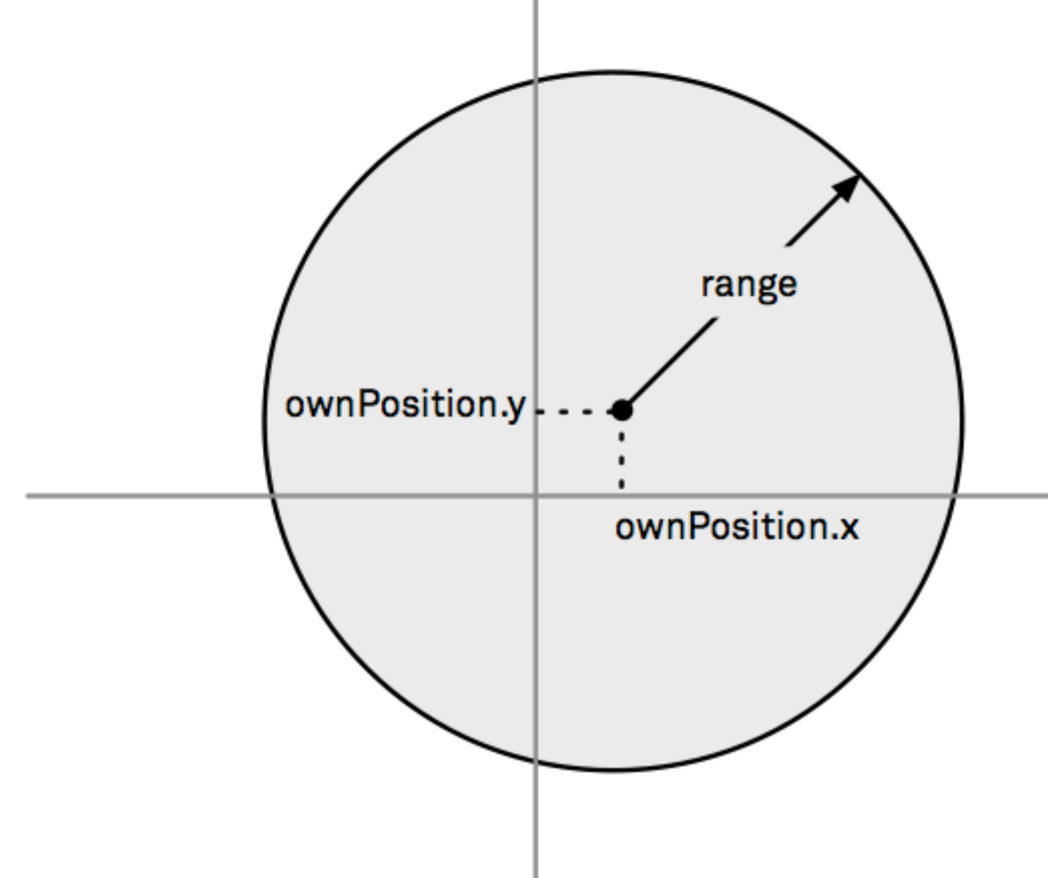
Let our position be any location

```
(defn in-range-2
  [position own-position range]
  (let [pos-x (first position)
        pos-y (last position)
        own-x (first own-position)
        own-y (last own-position)
        dx (- pos-x own-x)
        dy (- pos-y own-y)
        target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
   (< target-distance range)))
```



This is a Java program using Clojure syntax

Second Pass - a

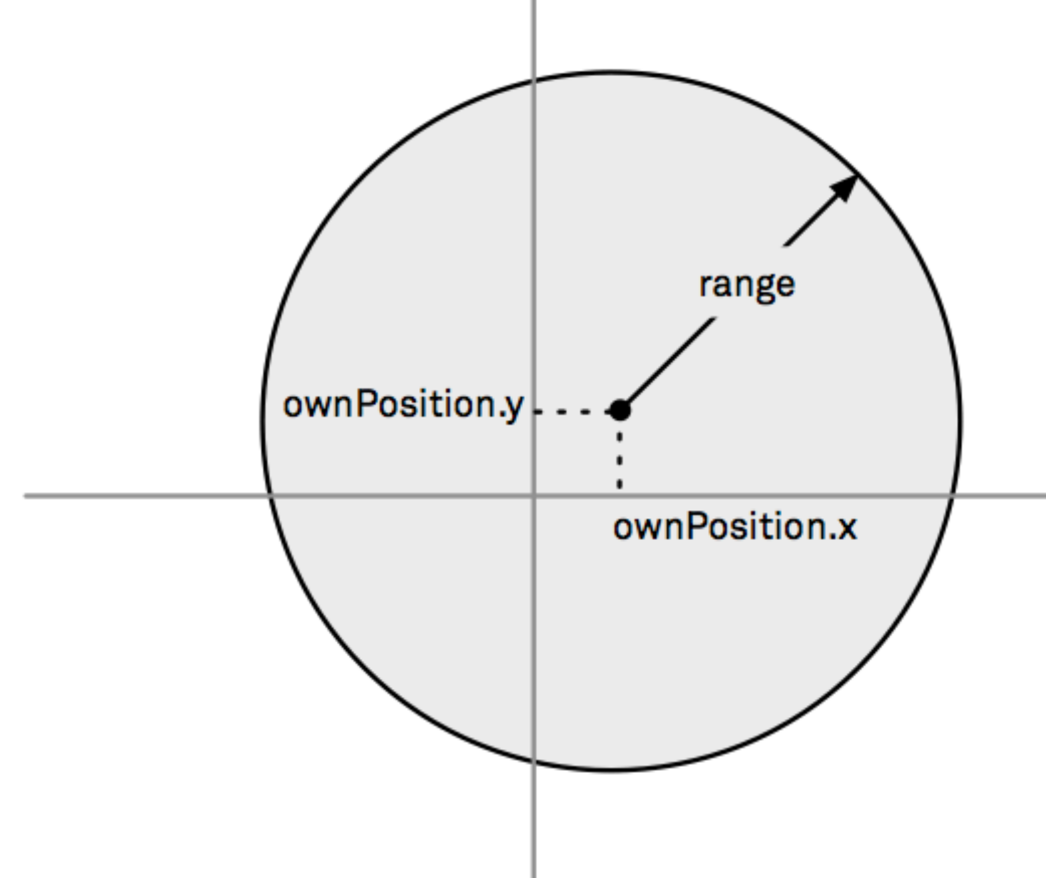


Using destructuring

```
(defn in-range-2a
  [[pos-x pos-y] [own-pos-x own-pos-y] range]
  (let [dx (- own-pos-x pos-x)
        dy (- own-pos-y pos-y)
        target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
    (< target-distance range)))
```

What do we gain? lose?

Second Pass - b

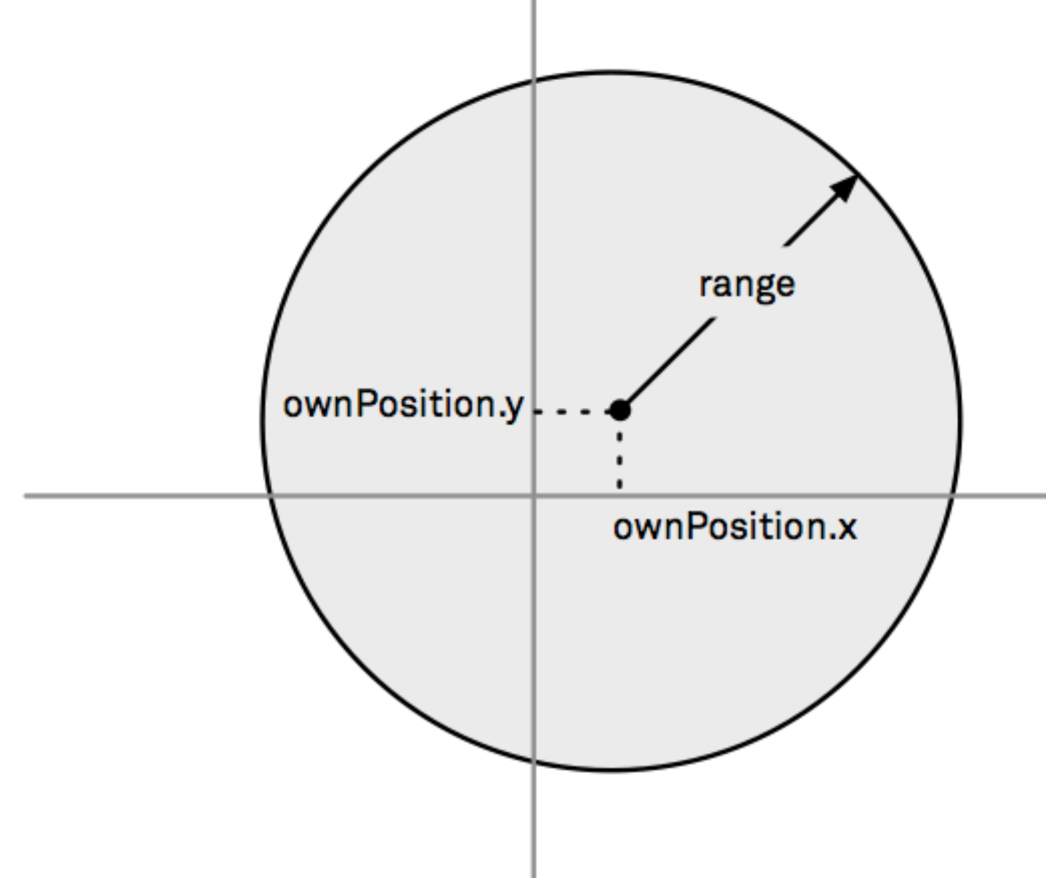


With map

```
(defn in-range-2b
  [position own-position range]
  (let [[dx dy] (map - position own-position)
        target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
    (< target-distance range)))
```

What do we gain? lose?

Second Pass - c

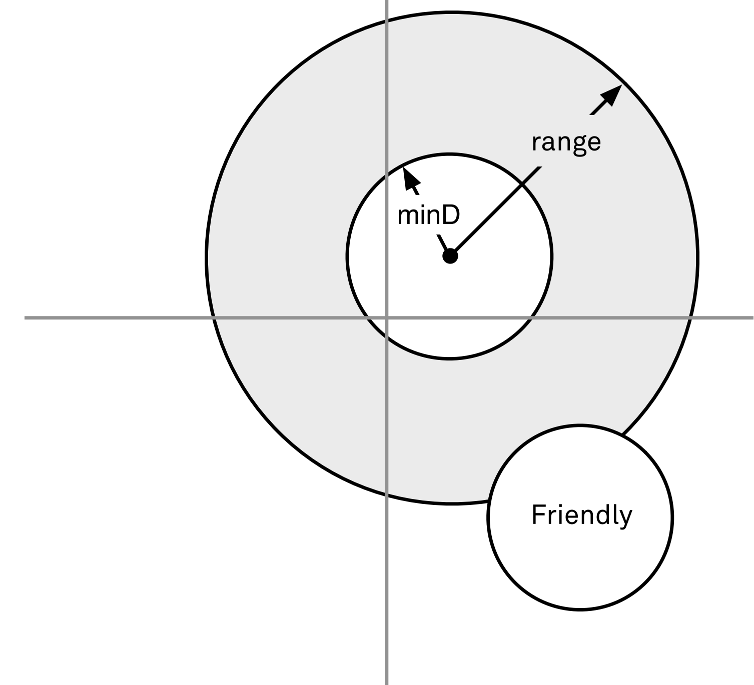


Using map & reduce

```
(defn in-range-2c
  [position own-position range]
  (let [delta (map - position own-position)
        target-distance (Math/sqrt (reduce + (map * delta delta)))]
    (< target-distance range)))
```

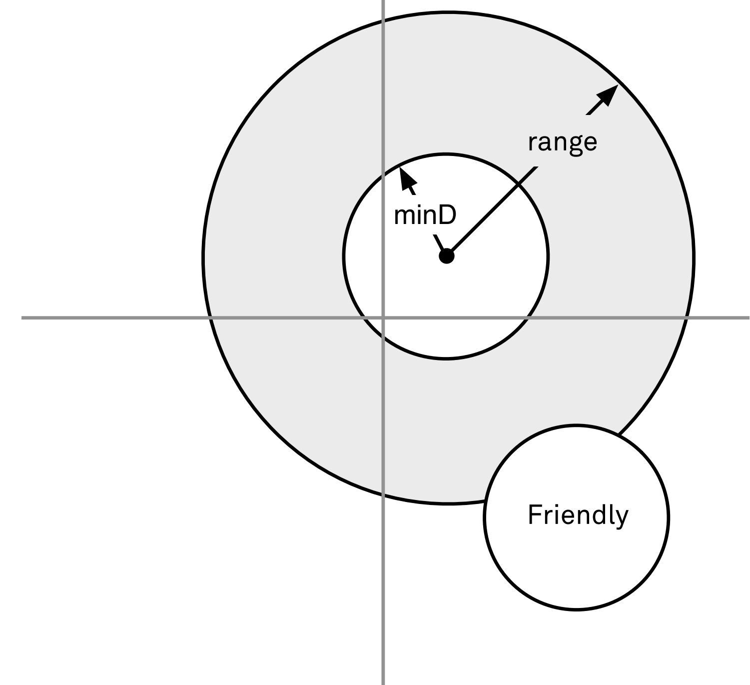
What do we gain? lose?

Third Pass




```
(defn in-range-3
  [safe-distance range own-position position friend-position]
  (let [delta (map - position own-position)
        target-distance  (Math/sqrt (reduce + (map * delta delta)))
        friend-delta (map - position friend-position)
        target->friend  (Math/sqrt (reduce + (map * friend-delta friend-delta)))]
    (and
     (< safe-distance target->friend)
     (< safe-distance target-distance  range))))
```

Third Pass




```
(defn distance-between
  [a b]
  (let [delta (map - a b)]
    (Math/sqrt (reduce + (map * delta delta)))))

(defn in-range-3a
  [safe-distance range self target friend]
  (and
     (< safe-distance (distance-between friend target))
     (< safe-distance (distance-between self target)  range)))
```

What is the Abstraction?

What are we doing?

Dealing with circles shapes

Union
Intersection
Complement

Is a point in a shape

circle - returns a function

```
(defn circle
  ([radius]
   (circle [0 0] radius))
  ([center radius]
  (fn
    [point]
    (<= (distance-between center point) radius))))


(def small-circle (circle 1))

(small-circle [0.5 0])              true
(small-circle [1 2])                false
```

outside

```
(defn outside
  [shape]
  (complement shape))
```

```
(def small-circle (circle 1))

((outside small-circle) [0.5 0])          false
((outside small-circle) [1 2])            true
```

union

```
(defn union
  ([shape]
   shape)
  ([shape-a shape-b]
  (fn [point]
    (or (shape-a point) (shape-b point))))
  ([shape-a shape-b & shapes]
   (fn [point]
     (let [all-shapes (conj shapes shape-a shape-b)]
       (reduce #(or %1 (%2 point)) false all-shapes)))))
```

Higher Level in range

```
(defn in-range-4
  [safe-distance range self target friend]
  (let [self-safe-zone (outside (circle self safe-distance))
        friend-safe-zone (outside (circle friend safe-distance))
        weapon-area (circle self range)
        target-zone (intersection weapon-area friend-safe-zone self-safe-zone)]
    (target-zone target)))
```

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)
    (/ 2)
    (- 1))

(+ c 3)

(/ **8** 2)

(- **4** 1)

-> Example

(def c 5)

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

```
(-> 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"} )
```

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

```
(def foo "hi")
```

foo & bar are symbols

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

They are bound to values

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