

Solving the Concurrency Problem with Clojure

Github: <https://github.com/gigasquid/oscon-solve-concurrency>

@gigasquid

Downloads: <http://10.10.32.101>

Setup

Github: <https://github.com/gigasquid/oscon-solve-concurrency>

```
cd clojure-intro
```

```
lein repl
```

Introduction

Carin Meier

aka @gigasquid

author of **Living Clojure**

works at **Cognitect**

Structure

Just enough Clojure

Structure

Just enough Clojure

Clojure State & Concurrency

Structure

Just enough Clojure

Clojure State & Concurrency

ClojureScript & core.async

Structure

Just enough Clojure

Clojure State & Concurrency

ClojureScript & core.async

Clojure Overview



Clojure Overview



Dynamic
Functional
Java Interop
Concurrency

ClojureScript Overview



Dynamic
Functional
JavaScript Interop
Concurrency



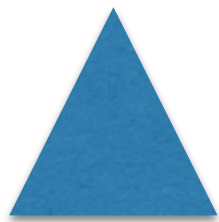
Let's Dive In



Clojure Intro

42

;; -> 42

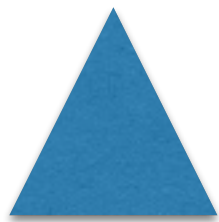


Clojure Intro

```
42
```

```
;; -> 42
```

Type in 42
expression evaluates 42
42 prints out



Clojure Intro

42

;; -> 42

REPL

Read Eval Print Loop

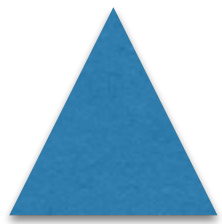


Simple Values

42

i *i* -> 42

Integers

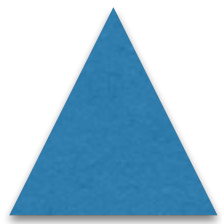


Simple Values

42.11

;; -> 42.11

Decimal

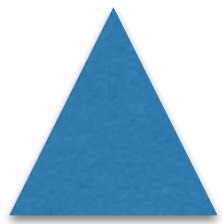


Simple Values

$1/3$

$:: \rightarrow 1/3$

Ratio

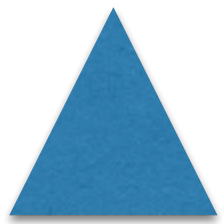


Simple Values

"cake"

;; -> "cake"

Strings



Simple Values

```
:cake
```

```
;; -> :cake
```

Keywords



Simple Values

`\c`

`;; -> \c`

Characters

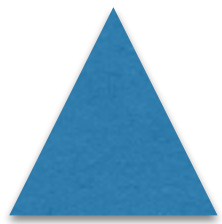


Simple Values

```
true
```

```
;; -> true
```

Booleans



Simple Values

```
false
```

```
;; -> false
```

Booleans

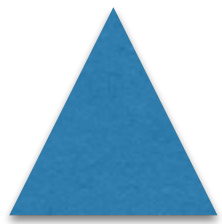


Simple Values

```
nil
```

```
;; -> nil
```

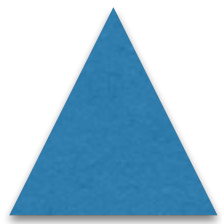
No value – nil



Simple Expressions

(+ 1 1)
; ; -> 2

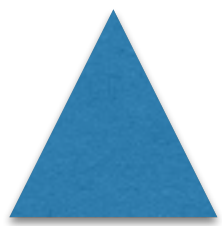
Operator goes first



Simple Expressions

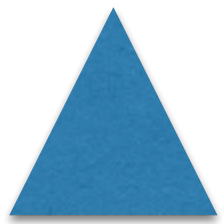
(+ 1 1)
; ; -> 2

Parens



Hitchhiker's Guide to Clojure

DON'T WORRY
ABOUT
THE
PARENS



Simple Expressions

```
( + 1 ( + 8 3 ) )  
; ; -> 12
```

Nesting



Ready for an adventure?

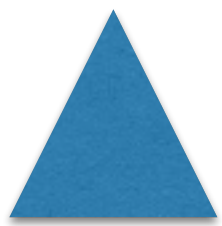




Collections

List

```
' (1 2 "jam" :marmalade-jar)  
;; -> (1 2 "jam" :marmalade-jar)
```

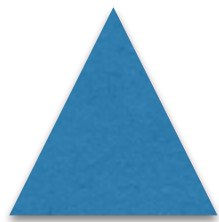


Collections

List

```
' ( 1 2 "jam" :marmalade-jar )  
;; -> ( 1 2 "jam" :marmalade-jar )
```

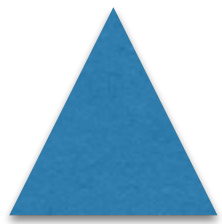
No commas needed!



Collections

List Manipulation

```
(first '(:rabbit :watch :marmalade :door))  
;; -> :rabbit  
(rest '(:rabbit :watch :marmalade :door))  
;; -> (:watch :marmalade :door)
```

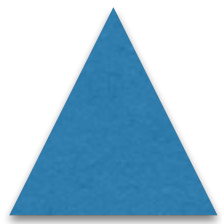


Collections

List Manipulation

```
(first (rest '(:rabbit :watch :marmalade :door)))  
;; -> :watch
```

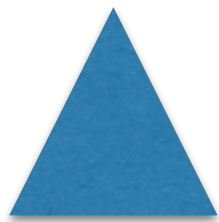
More nesting!



Collections

Vectors

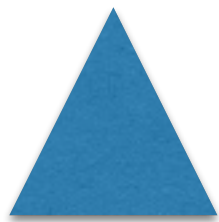
```
[ :jar1 1 2 3 :jar2 ]  
;; -> [ :jar1 1 2 3 :jar2 ]
```



Collections

Vector Manipulation

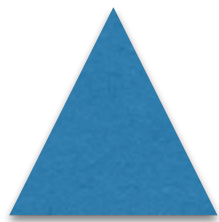
```
(first [:jar1 1 2 3 :jar2])  
;; -> :jar1
```



Collections

Vector Manipulation

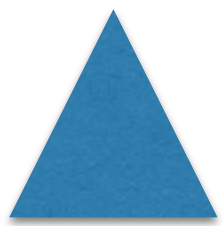
```
(last [:jar1 1 2 3 :jar2])  
;; -> :jar2
```



Collections

Vector Manipulation

```
(rest [ :jar1 1 2 3 :jar2 ])  
;; -> (1 2 3 :jar2)
```

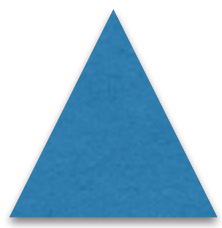


Collections

Vector Manipulation

```
(nth [ :jar1 1 2 3 :jar2 ] 0)  
;; -> :jar1  
(nth [ :jar1 1 2 3 :jar2 ] 2)  
;; -> 2
```

Index access!

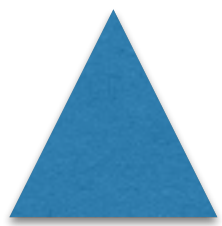


Collections

What do they have in common?

Immutable

value of the collection does not change



Collections

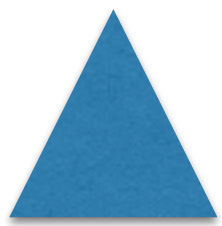
What do they have in common?

Immutable

value of the collection does not change

Persistent

create “smart” new versions of themselves with
structural sharing



Collections

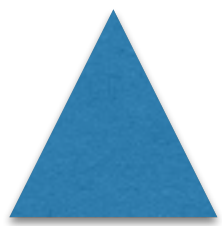
What do they have in common?

Common functions

`first, rest, last, count`

```
(count [1 2 3 4])
```

```
;; -> 4
```

Collections

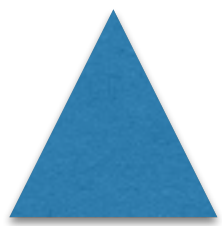
What do they have in common?

Common functions

`conj with vectors`

```
(conj [:toast :butter] :jam)  
;; -> [:toast :butter :jam]
```

```
(conj [:toast :butter] :jam :honey)  
;; -> [:toast :butter :jam :honey]
```



Collections

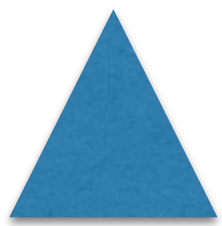
What do they have in common?

Common functions

`conj with lists`

```
(conj '(:toast :butter) :jam)  
;; -> (:jam :toast :butter)
```

```
(conj '(:toast :butter) :jam :honey)  
;; -> (:honey :jam :toast :butter)
```

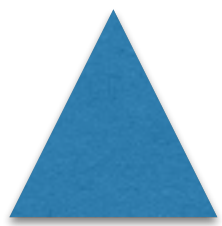


Collections

Maps

Key value pairs

```
{ :jam1 "strawberry"  
  :jam2 "blackberry" }  
;; -> { :jam2 "blackberry",  
        :jam1 "strawberry" }
```

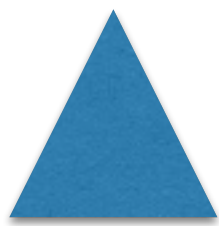


Collections

Maps

Getting data out – explicit get

```
(get { :jam1 "strawberry"  
      :jam2 "blackberry" } :jam2 )  
;; -> "blackberry"
```

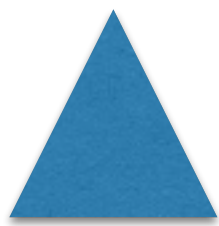


Collections

Maps

Getting data out – using keyword

```
( :jam2 { :jam1 "strawberry"  
          :jam2 "blackberry"  
          :jam3 "marmalade" } )  
;; -> "blackberry"
```

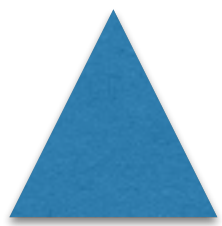


Collections

Maps

Manipulation – assoc

```
(assoc { :jam1 "red" :jam2 "black" }  
       :jam1 "orange")  
;; -> { :jam2 "black", :jam1 "orange" }
```

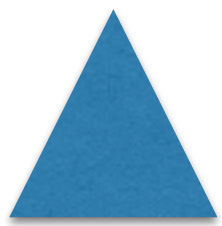


Collections

Maps

Manipulation – dissoc

```
(dissoc { :jam1 "strawberry"  
          :jam2 "blackberry" } :jam1)  
;; -> { :jam2 "blackberry" }
```

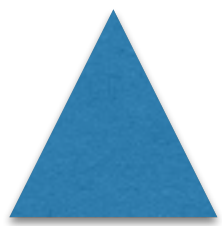


Collections

Maps

Manipulation – merge

```
(merge { :jam1 "red" :jam2 "black" }  
      { :jam1 "orange" :jam3 "red" }  
      { :jam4 "blue" } )  
;; -> { :jam4 "blue", :jam3 "red",  
       :jam2 "black", :jam1 "orange" }
```

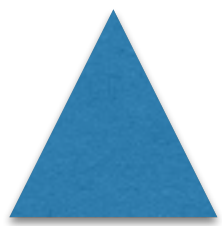



Collections

Sets

Elements with no dups

```
#{:red :blue :white :pink}  
;; -> #{:white :red :blue :pink}
```

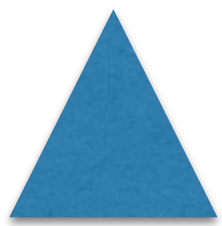


Collections

Sets

Handy Set Functions – difference

```
(clojure.set/difference #{:r :b :w}  
#{:w :p :y})  
;; -> #{:r :b}
```

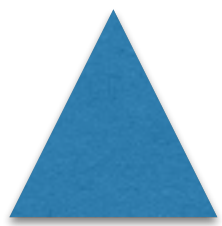


Collections

Sets

Handy Set Functions – intersection

```
(clojure.set/intersection #{:r :b :w}  
#{:w :p :y})  
;; -> #{:w}
```



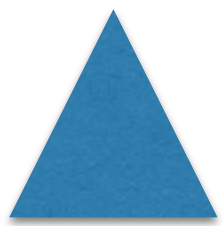
Collections

Sets

getting element

```
(get #{:rabbit :door :watch} :rabbit)  
;; -> :rabbit
```

```
(:rabbit #{:rabbit :door :watch})  
;; -> :rabbit
```

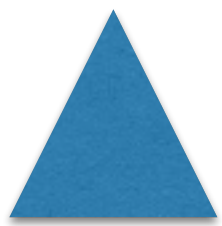


Collections

Sets

contains?

```
(contains?  
#{:rabbit :door :watch} :rabbit)  
;; -> true  
(contains? #{:rabbit :door :watch} :jam)  
;; -> false
```



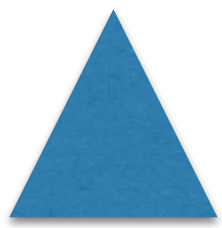
Collections

Sets

conj/disj

```
(conj #{:rabbit :door} :jam)  
;; -> #{:door :rabbit :jam}
```

```
(disj #{:rabbit :door} :door)  
;; -> #{:rabbit}
```



Collections

Summary

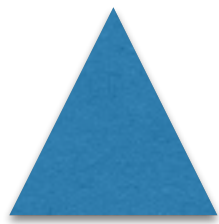
- Strings
- Integers
- Ratios
- Decimals
- Keywords
- Characters
- Booleans

Exercise Setup

Editor or Light Table

Leiningen

Git

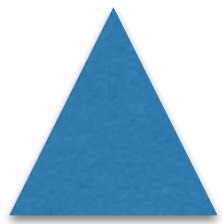


Exercises!

- `cd oscon-solve-concurrency`
- `cd clojure-intro`
- `lein test-refresh`

If you need it
`git checkout solutions`

to get back
`git checkout master`

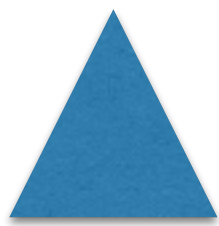


Symbols

def

```
(def developer "Alice")  
;; -> #'user/developer
```

```
developer  
;; -> "Alice"
```

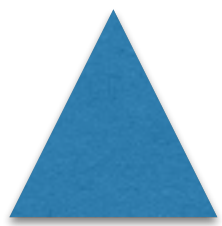


Symbols

def – (uses a global var)
values do not change

```
(def developer "Alice")  
;; -> #'user/developer
```

```
developer  
;; -> "Alice"
```



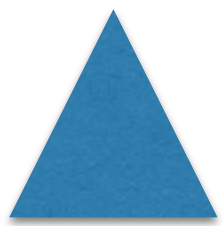
Symbols

let – (temporary var)
within the context of let

```
(def developer "Alice")  
;; -> #'user/developer
```

```
(let [developer "Alice in Wonderland"]  
  developer)  
;; -> "Alice in Wonderland"
```

```
developer  
;; -> "Alice"
```

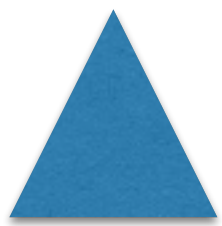


Symbols

defn – functions

```
(defn follow-the-rabbit [] "Off we go!")  
;; -> #'user/follow-the-rabbit
```

```
(follow-the-rabbit)  
;; -> "Off we go!"
```

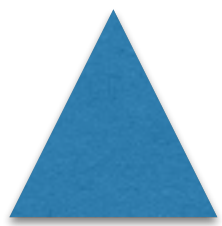


Functions

anonymous with fn

```
;;returns back a function  
(fn [] (str "Off we go" "!!"))  
;; -> #<user$eval1790$fn__791 user>
```

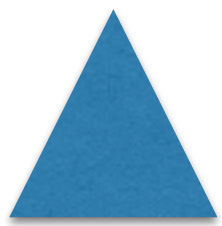
```
;;invoke with parens  
((fn [] (str "Off we go" "!!")))   
;; -> "Off we go!"
```



Functions

shorthand with #()

```
(#(str "Off we go" "!" ))  
;; -> "Off we go!"
```



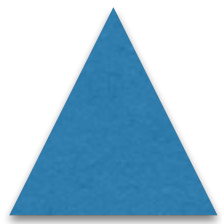
Flow Control

if

```
(if true "it is true" "it is false")  
;; -> "it is true"
```

```
(if false "it is true" "it is false")  
;; -> "it is false"
```

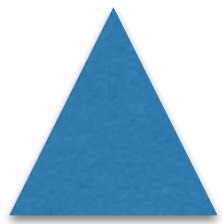
```
(if nil "it is true" "it is false")  
;; -> "it is false"
```

Flow Control

if

```
(if (= :drinkme :drinkme)  
  "Try it"  
  "Don't try it")  
;; -> "Try it"
```

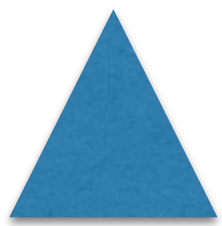


Flow Control

when

```
(when true "hi")  
;; -> "hi"
```

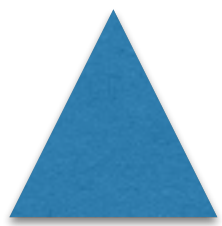
```
(when false "hi")  
;; -> nil
```



Flow Control

cond

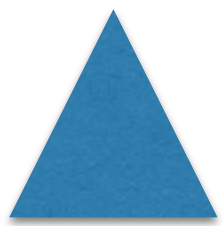
```
(let [bottle "drinkme"]  
  (cond  
    (= bottle "poison") "don't touch"  
    (= bottle "drinkme") "grow smaller"  
    (= bottle "empty") "all gone"))  
;; -> "grow smaller"
```



Flow Control

cond with else

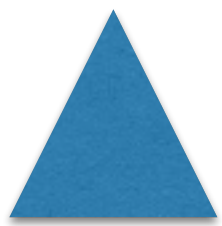
```
(let [bottle "mystery"]  
  (cond  
    (= bottle "poison") "don't touch"  
    (= bottle "drinkme") "grow smaller"  
    (= bottle "empty") "all gone"  
    :else "unknown"))  
;; -> "unknown"
```



Flow Control

case

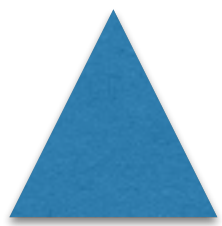
```
(let [bottle "drinkme"]  
  (case bottle  
    "poison" "don't touch"  
    "drinkme" "grow smaller"  
    "empty" "all gone" ) )
```



Flow Control

case with default

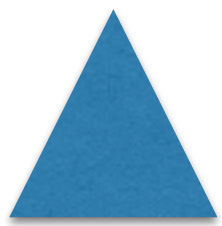
```
(let [bottle "mystery"]  
  (case bottle  
    "poison" "don't touch"  
    "drinkme" "grow smaller"  
    "empty" "all gone"  
    "unknown"))  
;; -> "unknown"
```



Functional Transformation

map the ultimate

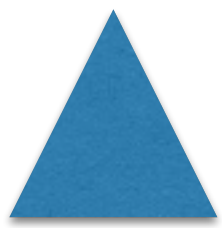
```
(map str [1 2 3 4 5])  
;; -> ("1" "2" "3" "4" "5")
```



Functional Transformation

map the ultimate

```
(map (fn [x] (str x "!")) [1 2 3 4 5])  
;; -> ("1!" "2!" "3!" "4!" "5!")
```

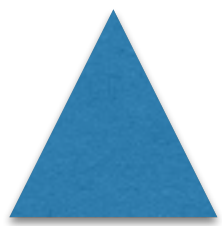



Functional Transformation

reduce the ultimate

```
(reduce + [1 2 3 4 5])
```

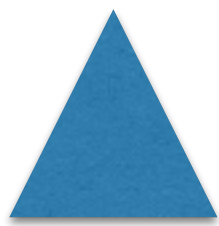
```
;; -> 15
```



Functional Transformation

reduce the ultimate

```
(reduce (fn [r x]  
          (str r (* 2 x))) [1 2 3 4 5])  
;; -> "146810"
```



Exercises!

Summary

- `cd oscon-solve-concurrency`
- `cd clojure-intro`
- `lein test-refresh`
- `open intro2_test.clj`
- `uncomment tests to get started`