CLOJURESCRIPT FROM SCRATCH

KRIS JENKINS

@KRISAJENKINS

CLOJURESCRIPT FROM SCRATCH

- We'll Cover
  - Reading Clojure code
  - Start playing with ClojureScript apps
  - Get a taste of event-based UI coding.

BEFORE WE START

What experience do we have in the room?

SYNTAX

FUNCTION CALLS

- Here's a Familiar Function Call:

```
plus(
  5,
  multiply(2, 3)
)
```

- Commas are Whitespace

```
plus(
  5
  multiply(2 3)
)
```

- The Function Name is the 1st Item

```
(plus
  5
  (multiply 2 3)
)
```

- Tidy it Up

```
(plus 5
      (multiply 2 3))
```

LISTS

- Clojure is a Lisp. Lisps like lists.
- The `first` thing is the function.
- The `rest` of the list are arguments.

VECTORS

```
[1 "two" 3.0]
```

- Similar to lists.
- The first element isn't special.
- Different performance characteristics.

NAMED FUNCTIONS

- Let's Make a New Function

```
(defn add-five
  "This adds five."
  [n]
  (+ n 5))
```

- In Use:

```
(add-five 3)
;=> 8
```

NAMED FUNCTIONS

- Let's Make Another

```
(defn triple
  "Triple the argument."
  [n]
  (* 3 n))
```

- In Use:

```
(triple (add-five 3))
;=> 24
```

COMBINING FUNCTIONS

- Let's make this into a Function

```
(triple (add-five 3))
;=> 24
```

- Abstract out the Argument

```
(defn foo
  [n]
  (triple (add-five n)))
```

- In Use:

```
(foo 3)
;=> 24
```

- Protip
  Call it "Syntactic Abstraction" for hipster points.

SYNTACTIC ABSTRACTION

- This isn't new.

```clojure
(defn foo
  [n]
  (triple (add-five n)))
```

- But it is lightweight.
- And reuses the core syntax.

```clojure
(+ 3 4)
;=> 7

(foo 3)
;=> 24
```

FUNCTION ARGUMENTS

- Functions Can Receive Multiple Arguments

```
(defn bar
  [a b c]
  (+ a (* b c)))
```

- Functions Can Take Multiple Arguments

```
(defn pow4
  [n]
  (* n n n n))
```

- Once You're Used To It, This:

```
(* a b c d)
```

- ...is Nicer Than This:

```
a * b * c * d
```

1ST PRACTICAL

TASK

Write a function that computes `5(a^2 + 3b)`

Hint: Start by writing a `square` function.

- LightTable

```
C-space instarepl
```

- Syntax Reminder

```
(defn bar
  [a b c]
  (+ a (* b c)))

(defn pow4
  [n]
  (* n n n n))
```

HOW DID WE DO?

- Something Like This?

```clojure
(defn square
  [n]
  (* n n))

(defn triple
  [n]
  (* 3 n))

(defn formula
  "Compute 5(a^2 + 3b)"
  [a b]
  (* 5
     (+ (square a)
        (triple b))))
```

- Operator Precedence?

GOTCHAS

CALLING FUNCTIONS 1

```
(defn square
  [x]
  (x * x))
```

The function is always the *first* thing in the list.

```
(defn square
  [x]
  (* x x))
```

CALLING FUNCTIONS 2

```
(defn square
  [x]
  * (x x))
```

The function is always the first thing *in* the list.

```
(defn square
  [x]
  (* x x))
```

CALLING FUNCTIONS 3

```
(defn square
  [x]
  * x x)
```

The function is always the first thing in the *list*.

```
(defn square
  [x]
  (* x x))
```

SUMMARY

- The function is always the *first* thing in the list.
- The function is always the first thing *in* the list.
- The function is always the first thing in the *list*.

MORE SYNTAX

STRINGS

```
["hello" "world"]
```

- Double Quotes, Not Single Quotes

KEYWORDS

```
[:foo :bar :foobarbara]
```

MAPS

```
{:name "Steve"
 :age 34}
```

- Properties:
  - Unordered collection of key-value pairs.
  - Remember commas are whitespace.

MAPS & KEYWORDS ARE FUNCTIONS

```clojure
(def person
  "Hello, my name is Steve."
  {:name "Steve"
   :age 34})
```

- Can We Print It?

  ```clojure
  (str person)
  ;=> {:age 34, :name "Steve"}
  ```

- A Map is a Function of its Keys

  ```clojure
  (person :age)
  ;=> 34
  ```

- A Keyword is a Function that takes a Map

  ```clojure
  (:age person)
  ;=> 34
  ```

ANOTHER EXAMPLE

```
{0 "zero"
 1 "one"
 "???" :john}
```

- Arbitrary keys, arbitrary values.

ON TO CLOJURESCRIPT

LET'S DIVE IN

…by exploring:

```
git clone https://github.com/krisajenkins/CLJS-Demo
```

```
lein new chestnut hackernews -- --less --om-tools
lein repl
```

```
(run)
(browser-repl)
```