understanding clojure through data

James Reeves
@weavejester

boolean·knot
disclaimer

I'll be telling a few lies
disclaimer

I'll be telling a few *simplifications*
a functional programming language

a lisp (more on that later)
officially targets **Java (Clojure)**

**.NET (ClojureCLR)**

**javascript (ClojureScript)**
first released in September 2007
by Rich Hickey
let's start with edn
extensible data notation
edn ⊆ clojure

json ⊆ javascript
<table>
<thead>
<tr>
<th>json</th>
<th>edn</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159</td>
<td>3.14159</td>
</tr>
<tr>
<td>&quot;hello world&quot;</td>
<td>&quot;hello world&quot;</td>
</tr>
<tr>
<td>[1, 2, 3, 4]</td>
<td>[1 2 3 4]</td>
</tr>
<tr>
<td>{&quot;name&quot;: &quot;alice&quot;}</td>
<td>{&quot;name&quot; &quot;alice&quot;}</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>null</td>
<td>nil</td>
</tr>
</tbody>
</table>
collections

json

array  [1 2 3]  ordered, random access
collections

edn

vector [1 2 3] ordered, random access
list (1 2 3) ordered
set #{1 2 3} unordered, distinct
Identifiers

JSON

string "name" text data and identifier
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>&quot;name&quot;</td>
<td>text data</td>
</tr>
<tr>
<td>keyword</td>
<td>:name</td>
<td>references itself</td>
</tr>
<tr>
<td>symbol</td>
<td>name</td>
<td>references something else</td>
</tr>
</tbody>
</table>
example

{:name "Alice"
 :sex :female
 :job cryptographer}
namespaces

:status/ready

mammal.canine/dog
tags

#inst "1985-04-12T23:20:50.52Z"

#uuid "f81d4fae-7dec-11d0-a765-00a0c91e6bf6"
tags

#color/rgb  "e8a433"

#color/rgb  [232 164 51]
what's the connection to clojure?
clojure = edn + eval
evaluation

most data evaluates to itself

eval

"foo" → "foo"
123 → 123
[1 2 3] → [1 2 3]
{:x 1} → {:x 1}
evaluation

symbols evaluate to a bound value

eval

pi ➞ 3.14159
message ➞ "Hello World"
lists evaluate based on their first element

(eval

(+ 1 1)  ➡️  2
(and true false)  ➡️  false

)
evaluation

functions evaluate their arguments

\((+ (* 3 3) (* 4 4))\)

\(\Rightarrow (+ 9 16)\)

\(\Rightarrow 25\)
evaluation

macros evaluate their return value

\[ \text{postfix} \ (9 \ 16 \ +) \]

\[ \Rightarrow \ (+ \ 9 \ 16) \]

\[ \Rightarrow \ 25 \]
homoiconic
homo · iconic

the same representation
homo · iconic
writing code with data
clojure  ♥  data
why such a close relationship?
macros allow us to add new syntax through libraries

core.async   async programming
core.logic   logic programming
core.typed   static typing
but is that the only reason?
“Simple Made Easy”
simple

complex

why?
complexity \neq \text{cardinality}
complexity = interlacing connections coupling
how do we usually deal with complexity?
mutable state
mutable state
object encapsulates state
object
methods
encapsulation isolates complexity
defensive strategy
what would an offensive strategy look like?
can something have zero complexity?
immutable values
mutable state needed for

1. performance
2. communication across threads
object
do we need encapsulation?
walls are expensive
can we do that?
in distributed environments we often work with immutable values
are there any immediate benefits?
free API

getters  transformations  diffing
setters  transversal  serialisation
equality  merging  deserialisation
lensing  auditing  concurrency
end

questions?