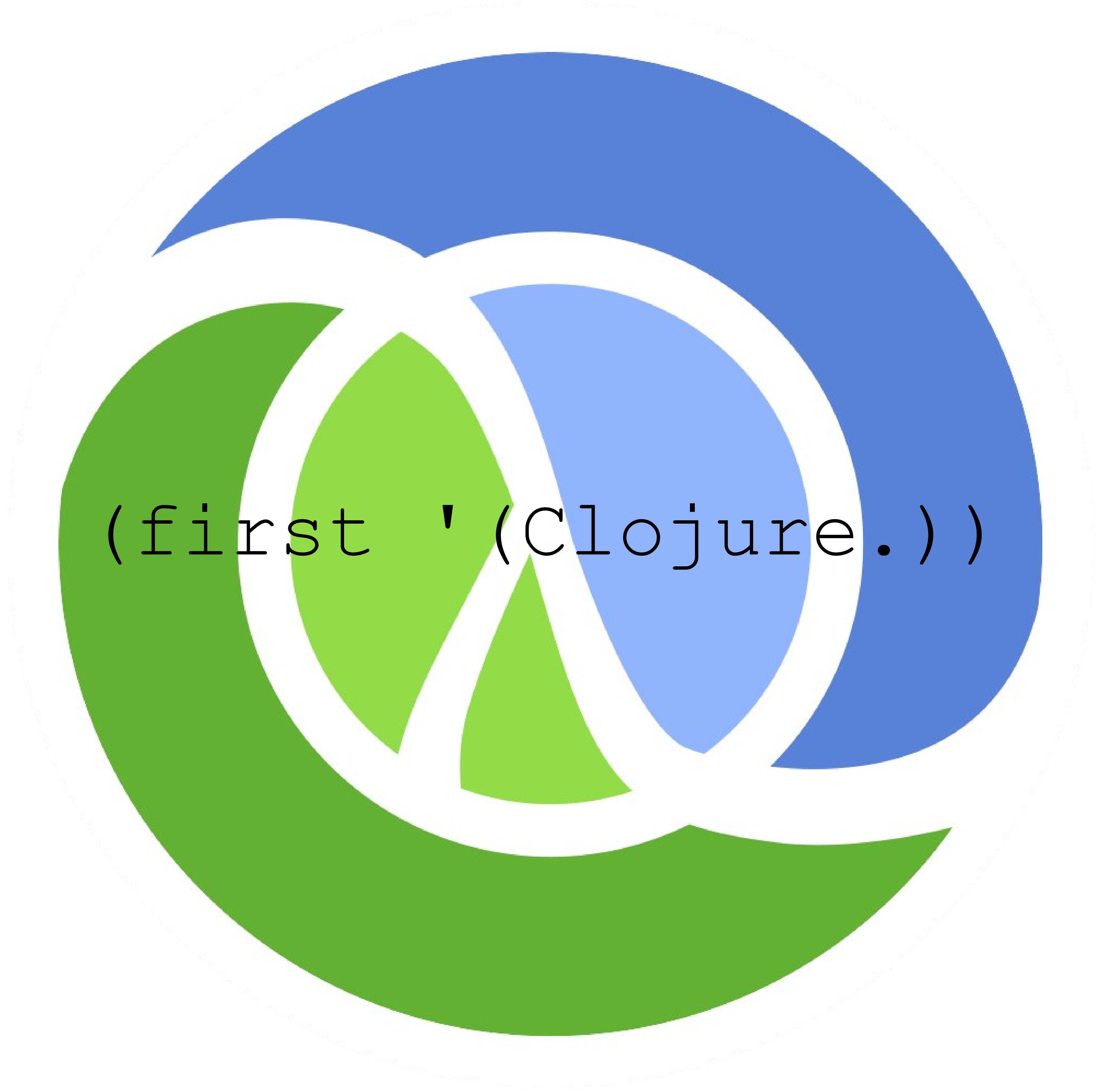

(first '(Clojure.))

Clojure is

A dynamic,

LISP-based

programming language

running on the JVM

Origin

2007, Rich Hickey

.. 1958, John McCarthy

Features

Functional

Homoiconic

Immutability (persistent data structures)

Powerful (composition, macros)

Plus

Concurrency (separates identity and state)

Pragmatic (compact syntax, high-level collections)

Java Interop (symbiotic)

Applicability

Designed as a general purpose language where functional style is idiomatic

.. Use anywhere Java (the JVM) is used

The JVM?

Fun!

Homoiconic

```
(+ 1 1)
```

- Prefix notation
- List Processing *(LISP)*
- Code is Data
- Math Substitution Principle

=> Predictable!

Syntax

- Scalars
- Collections
- Special Forms

Scalars

```
\c          ; characters
"Hello"     ; strings

42          ; numbers
3/2         ; ratios

true        ; booleans

nil
```

Names

```
; symbols
i
println
+
empty?
next-state
.toUpperCase

; keywords
:name
:use
```

Collections

Lists

```
(println "Hello")
;=> Hello

'(println "Hello")
;=> (println "Hello")

(first '(println "Hello"))
;=> println
```

Vectors

```
["bag" "of" 4 "chips"]
```

Hash-maps

```
{:name "Intro"
 :date "2012-05-29"}
```

Sets

```
#{"unique" "snowflakes"}
```

Commas

```
; commas: just whitespace!

["bag", "of", 4, "chips"]

{:name "Intro",
 :date "2012-05-29"}
```

Special Forms

Define

```
(def i 1)
```

If

```
(if true
  (/ 1 1)
  (/ 1 0))
```

Functions

```
(fn [name] (str "Hello " name "!"))
```

Named Functions

```
(def greet (fn [name]
               (str "Hello " name)))
```

The `defn` macro

```
(defn greet [name]
  (str "Hello " name))
```

Let the local names in

```
(defn greet [name]
  (let [greeting "Hello"
        out (str greeting " " name)]
    out))
```

Sugared Lambdas

```
#(str %1 %2)

; same as
(fn [a1 a2] (str a1 a2))
```

Some More Sugar

```
#"\w+" ; regexps

@ref ; (deref ref)
```

The rest is just eval/apply

Basic Usage

=> represents results

Vectors

```
(def items [1 2 3])

(first items)
=> 1

(rest items)
=> (2 3)

(items 0)
=> 1
```

Adding to collections

```
(cons items [3 4])
=> [1 2 3 4]

(conj items 3)
=> [1 2 3]
```

Mapping

```
(map inc [1 2 3])
=> (2 3 4)

(map #(+ % 1) [1 2 3])
=> (2 3 4)
```

Reducing

```
(reduce #(- %1 %2) [1 2 3])
=> -4
```

For Comprehension

```
(for [i (range 1 7) :when (even? i)]
  (str "item-" i)))

=> ("item-2" "item-4" "item-6")
```

Laziness

```
(take 3 (repeat 1))
=> (1 1 1)

(take 3 (filter even?
                (iterate inc 1)))
=> (2 4 6)

(read-lines "/var/huge-data.log")
```

Composing Data

Constructs

```
(def persons [{:id 1
               :name "Some Body"}
              {:id 2
               :name "Some Other"}])

(let [person (persons 0)]
  (person :name))

=> "Some Body"
```

The key is getting the value

```
(:name (first persons))
=> "Some Body"

(map :name persons)
=> ("Some Body" "Some Other")
```

Records

```
(defrecord Person [id name])

(let [person (Person. 1 "Some Body")]
  (:name person))

=> "Some Body"
```

Destructuring

```
(let [[one other] [1 2]]
  (+ one other))

=> 3
```

The Relevant Parts

```
(let [person {:id 1 :name "Anyone"}
      {id :id name :name} person]
  [id name])

=> [1 "Anyone"]
```

Immutability

```
(let [unknown {}]
  (assoc unknown :name "Neo")
  (:name unknown))
=> nil
```

One State at a Time

```
(defn reborn [person name]
  (assoc person :name name))

(println (:name (reborn {} "Neo")))

=> "Neo"
```

Namespaces

```
(ns some.cli
  (:require [some.core :as core]))

(defn -main [& args]
  (doseq [item args]
    (println (core/process item))))
```

Compose Data, Compose Functions

Data first

Collections

Flow

Abstractions

How To Think In Clojure

.. leave imperative programming

pulling levers

flipping switches

adding and removing in place

Functional programming

Composes data

Composes functions

Transforms data

A succession of states over time

Concurrency

States over Time

Value as State

Reference to Identity

Identity over Time

Time Isolation

STM

Atoms

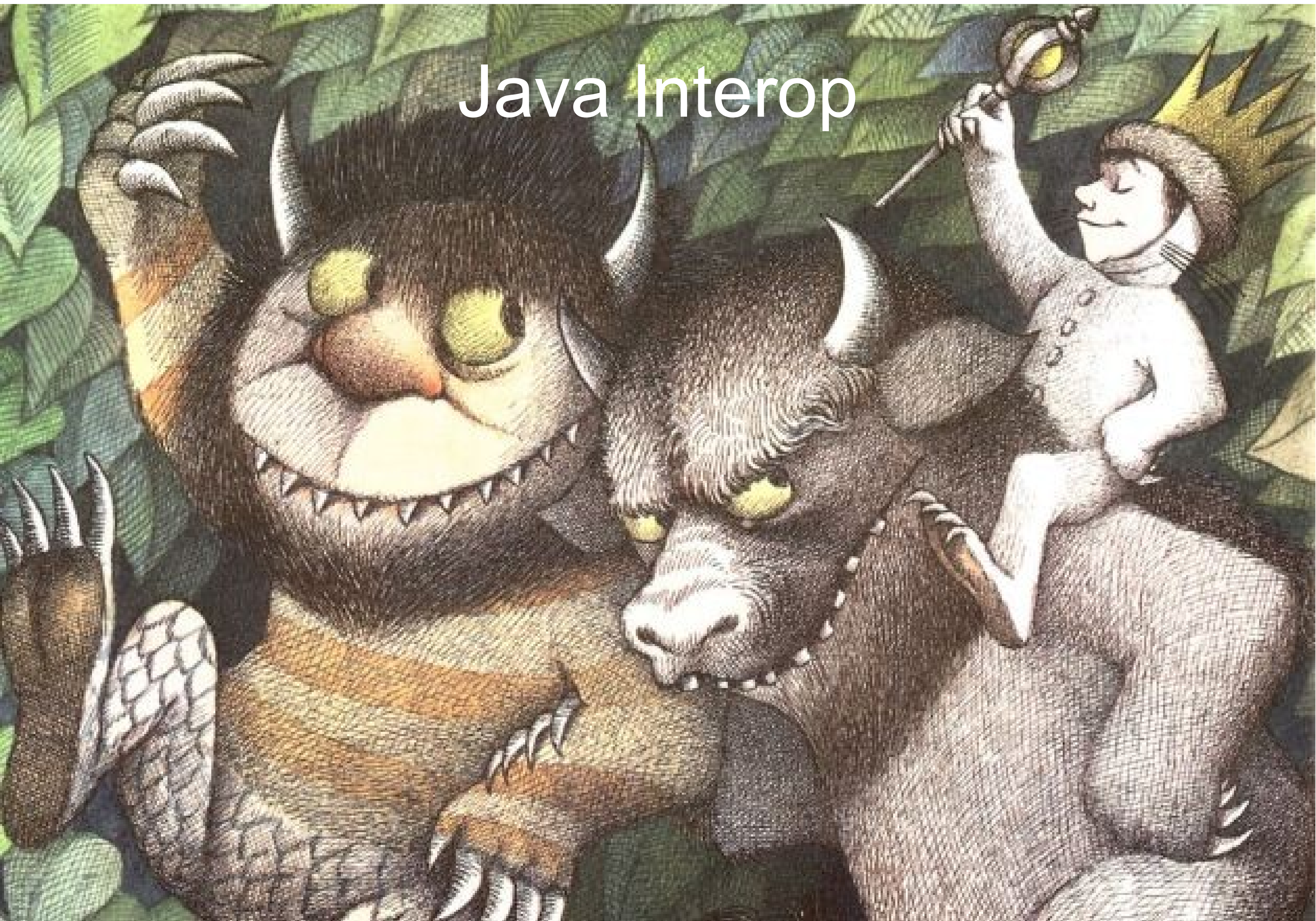
```
(def item-counter (atom 0))

(defn next-item []
  (swap! item-counter inc)
  (str "item-" @item-counter))

(next-item)
=> "item-1"
(next-item)
=> "item-2"
```

Refs

```
(def item1 (ref {:a "foo"}))
(def item2 (ref {:a "bar"}))

(let [v1 (:a @item1) v2 (:a @item2)]
  (dosync
    (alter item1 assoc :a v2)
    (alter item2 assoc :a v1)))
```

Agents

```
(def info (agent {}))

(send info assoc :status :open)

; eventually,
; (assoc info :status :open)
; is called
```


Java Interop

Monsters In The Cellar

Can be shackled by parentheses,

even fed immutability,

but their nature is wild

.method

```
(.toString
  (.resolve (java.net.URI. "http://example.org/a")
            "/b"))

=> "http://example.org/b"
```

Parallel HTTP Fetches

```
(ns parallel-fetch
  (:import (java.io InputStream InputStreamReader BufferedReader)
           (java.net URL HttpURLConnection)))
```

```
(defn get-url [url]
  (let [conn (doto (.openConnection (URL. url))
               (.setRequestMethod "GET")
               (.connect))]
    (with-open [stream (BufferedReader.
                         (InputStreamReader.
                           (.getInputStream conn)))]
      (.toString (reduce #(.append %1 %2)
                         (StringBuffer.) (line-seq stream))))))
```

```clojure
(defn get-urls [urls]
  (let [agents (doall (map #(agent %) urls))]
    (doseq [agent agents] (send-off agent get-url))
    (apply await-for 5000 agents)
    (doall (map #(deref %) agents))))

(prn (get-urls ["http://erlang.org" "http://clojure.org/"]))
```

Interface On Your Own Terms

Idiom

```
(get-name root)

(get-attr link "href")

(get-child-elements div)
```

Protocol

```
(defprotocol DomAccess
  (get-name [this])
  (get-attr [this attr-name])
  (get-child-elements [this]))
```

Realization

```
(extend-type org.w3c.dom.Node
  DomAccess

   (get-name [this] (.getNodeName this))

   (get-attr [this attr-name]
     (if (.hasAttribute this attr-name)
       (.getAttribute this attr-name)))

   (get-child-elements [this]
     (filter #(= (.getNodeType %1) Node/ELEMENT_NODE)
             (node-list (.getChildNodes this)))))
```

Using Clojure For Real

Leiningen

```
$ lein deps

$ lein compile

$ lein repl

$ lein uberjar

$ lein ring server-headless
```

Editors

Vim: VimClojure

Emacs: Swank

IDEA: La Clojure

More Info

<http://clojure.org/>

<http://clojure.org/cheatsheet>

<http://clojuredocs.org/>

<http://dev.clojure.org/display/community/Clojure+Success+Stories>

<http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey>

<http://www.infoq.com/presentations/Simple-Made-Easy>

Have Fun!

(thanks!)

@niklasl

@valtechSweden

Attribution

- Clojure Logo © Rich Hickey
- Darth Vader
- "Where the Wild Things Are" © Maurice Sendak * Parallel HTTP Fetches by Will Larson