Scala & Clojure Playing Nice

David Pollak

QCon Beijing April, 2015

About @dpp

- Wrote some Spreadsheets
- Founded Lift/Wrote Beginning Scala
- Coding Clojure 3 Years
- Crazy Passionate Lawyer-trained Tech Dude

Preso Structure

- Background on Scala & Clojure
- Live Coding
- Thoughts & Questions

Scala

- Hybrid Functional/OO Language… All things to all people
- Gnarly (特别危险的冲浪条件) Type System
- Java-like syntax
- Excellent Java Interopt

Clojure

- Modern Lisp/Functional
- Optional Type Systems
- Opinionated re: Immutability
- Super-Excellent Java Interopt

Both Compile to JVM ByteCode

… Can Subclass Java Classes

… And Implement Java Interfaces

Similarities

- Immutable Data & Collections
- Super easy to pass "functions" (really anonymous inner classes)
- Great for reducing complexity & concurrent systems
- Both address "Expression Problem"

Expression Problem

"The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code."

表达问题

“我们的目标是根据使用情况定义一种数据类型，同时可以向这个数据类型上添加新的使用情况和新函数，而无需重新编译代码”。

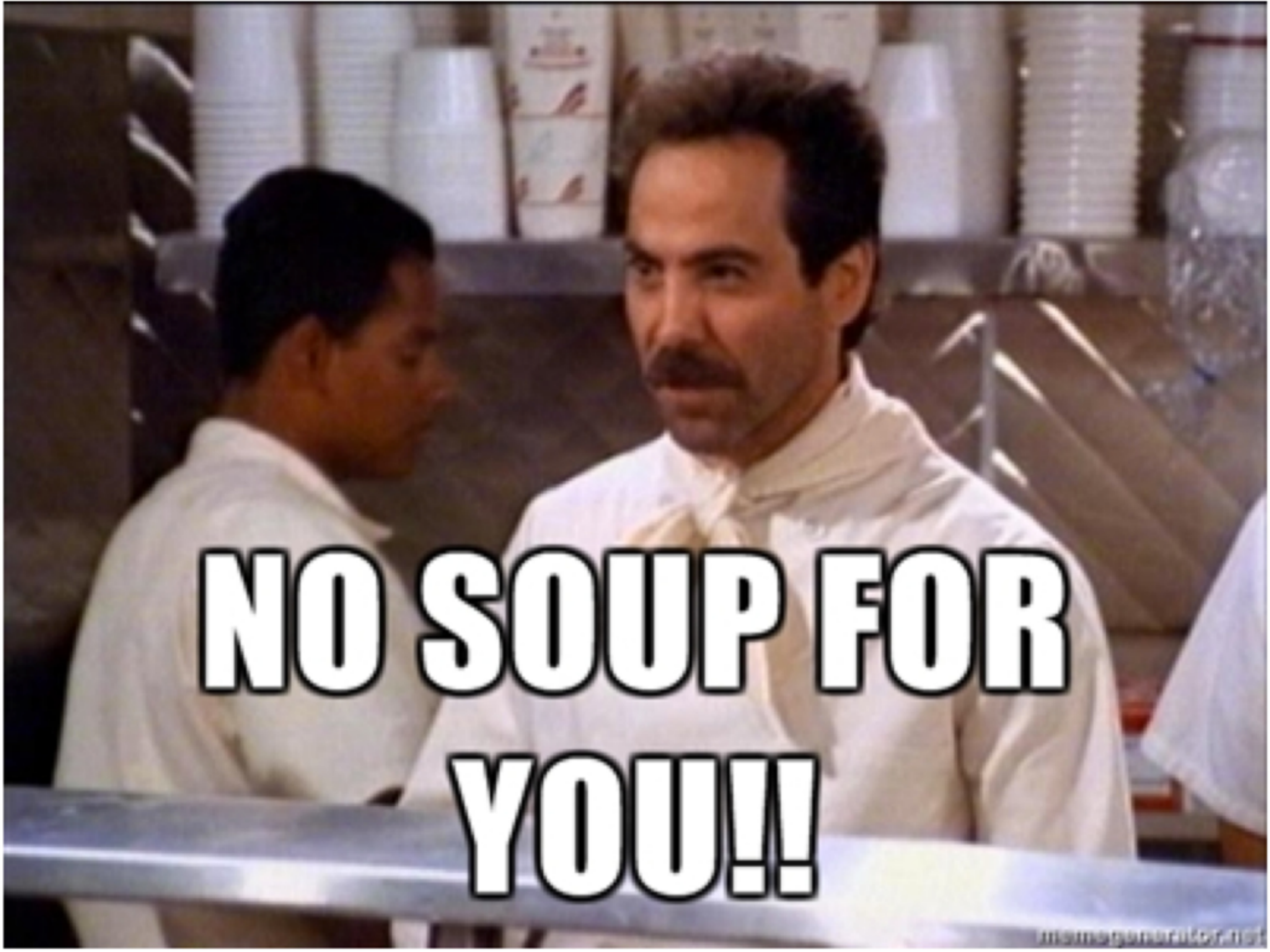
All I want to do is add a method…

to a library class!

…in Java

- Subclassing – can add new data
- But cannot add functions to existing data/classes

MT CYCLOPTIC COLLEAGUE TELLS ME
I CAN'T HAVE
"SHARKS WITH FRICKEN LASER BEAMS ATTACHED TO
THEIR HEADS
memegenerator.net


NO SOUP FOR
YOU!!
memegenerator.net

...in Scala

- Subclassing: Add new data
- implicits: "Scoped" adding new functionality to existing data

Scala Sample

```
“foo”.toWombat()

class MyWombat(s: String) {
  def toWombat() …
}

implicit def asAWombat(s: String):
  MyWombat = new MyWombat(s)
```

…in Clojure

- Subclassing & Maps
- Protocols add functions to data

Clojure Sample

```
(defprotocol FromScala
 (to-c [x] "Scala -> Clojure”))

(extend Iterator FromScala
  {:to-c
   (fn [it](letfn [(build []
                (if (.hasNext it)
                  (cons (to-c (.next it))
                        (lazy-seq (build)))
                  nil))]
       (build)))})

(defn seq-to [^Seq seq]
  (-> seq .iterator to-c))

(extend Seq FromScala
  {:to-c seq-to})
```

That Distributed & Concurrent Thing...

Distributed & Concurrent

- Easily Serializable
- Immutable
- Like REST: data in, answer out

Live Coding

HTML

```
<div data-lift="Actorize"></div>
<p>
    <span id="chats">
        <ul>
            <li>Chat 1</li>
            <li>Chat 2</li>
        </ul>
    </span>
<hr>

<input id="in">
<button id="send">Chat</button>
</p>
```

Browser Receive

```
(defn receive
      "receive from server"
      [x]
      (let
        [msg (t/read t-reader x)]
        (cond
          (seq? msg)
          (swap! app-state assoc-in [:chats]
                                    (vec msg))

          (string? msg)
          (swap! app-state update-in [:chats]
                           conj msg)

          :else nil)))
```

Browser Render

```
(om/root
  (fn [data owner]
      (reify om/IRender
             (render [x]
    (apply
       dom/ul
       nil
       (map #(dom/li nil %)
            (:chats data))))))
  app-state
  {:target (by-id "chats")})
```

Browser Send

```
(defn send
      "send data to the server"
      [data]
      (js/sendToServer (t/write t-writer data)))

(defn send-chat
      []
      (let [box (by-id "in")]
           (send (.-value box))
           (set! (.-value box) "")
           ))

(set! (.-onclick (by-id "send")) send-chat )
```

Lift - Set up xport

```
val clientProxy =
   session.serverActorForClient("omish.core.receive",
     shutdownFunc = Full(actor =>
                postMsg.invoke('remove -> actor)),
     dataFilter = transitWrite(_))

 postMsg.invoke('add -> clientProxy)

 val serverActor = new LiftActor {
   override protected def messageHandler =
   {case JString(str) =>
        postMsg.invoke(ClojureInterop.transitRead(str))}}

 Script(JsRaw("var sendToServer = " +
        session.clientActorFor(serverActor).toJsCmd).cmd)
```

Clojure Chat Server

```
(async/go-loop [chats [] listeners []]
    (match (<! chat-server)
      [:add f]
      (do
        (send! f (take-last 40 chats))
        (recur chats (conj listeners f)))

      [:remove f]
      (recur chats (remove #(identical? f %) listeners))

      (msg :guard string?)
      (do
        (doseq [f listeners] (send! f msg))
        (recur (conj chats msg) listeners))

      :else
      (recur chats listeners)
      ))
```

Wrap-up

- Easy to convert between Scala & Clojure types
- Clojure & Scala do well for distributed apps
- JVM makes it easy to play together

Scala & Clojure
Play well together

Thanks!
Questions?