Clojure

A brief introduction into the modern Lisp dialect

Piotr Ptaszyński

Poznań, 2016

- Created by Rich Hickey
- Initial public release in October 2007
- First stable release in May 2009
- But *Lisp* family dates from 1958 (second-oldest high-level widespread programming languages)
- Language name comes from *closures* – significant functional programming mechanism

- Created by Rich Hickey
- Initial public release in October 2007
- First stable release in May 2009
- But *Lisp* family dates from 1958 (second-oldest high-level widespread programming languages)
- Language name comes from *closures* – significant functional programming mechanism

- Created by Rich Hickey
- Initial public release in October 2007
- First stable release in May 2009
- But *Lisp* family dates from 1958 (second-oldest high-level widespread programming languages)
- Language name comes from *closures* – significant functional programming mechanism

- Created by Rich Hickey
- Initial public release in October 2007
- First stable release in May 2009
- But *Lisp* family dates from 1958 (second-oldest high-level widespread programming languages)
- Language name comes from *closures* – significant functional programming mechanism

- Created by Rich Hickey
- Initial public release in October 2007
- First stable release in May 2009
- But *Lisp* family dates from 1958 (second-oldest high-level widespread programming languages)
- Language name comes from *closures* – significant functional programming mechanism

- Typical for *Lisp* functional programming paradigm
- Available for multiplatform engines i.e. (*JVM*, *CLR*, *JavaScript*)
- Emphasis on concurrency and immutable data structures
- Code as data approach
- Community-driven development process
- REPL – *read-eval-print loop*

- Typical for *Lisp* functional programming paradigm
- Available for multiplatform engines i.e. (*JVM*, *CLR*, *JavaScript*)
- Emphasis on concurrency and immutable data structures
- Code as data approach
- Community-driven development process
- REPL – *read-eval-print loop*

- Typical for *Lisp* functional programming paradigm
- Available for multiplatform engines i.e. (*JVM*, *CLR*, *JavaScript*)
- Emphasis on concurrency and immutable data structures
- Code as data approach
- Community-driven development process
- REPL – *read-eval-print loop*

- Typical for *Lisp* functional programming paradigm
- Available for multiplatform engines i.e. (*JVM*, *CLR*, *JavaScript*)
- Emphasis on concurrency and immutable data structures
- Code as data approach
- Community-driven development process
- REPL – *read-eval-print loop*

- Typical for *Lisp* functional programming paradigm
- Available for multiplatform engines i.e. (*JVM*, *CLR*, *JavaScript*)
- Emphasis on concurrency and immutable data structures
- Code as data approach
- Community-driven development process
- REPL – *read-eval-print loop*

- Typical for *Lisp* functional programming paradigm
- Available for multiplatform engines i.e. (*JVM*, *CLR*, *JavaScript*)
- Emphasis on concurrency and immutable data structures
- Code as data approach
- Community-driven development process
- REPL – *read-eval-print loop*

Figure: Companies which use *Clojure* according to ClojureDocs

```clojure
(ns hello-world.core
    (:gen-class))

(defn -main
    [& args]
    (println "Hello, World!"))

; => Hello, World!
```

- Literal representation of data structures:

```
3
"an example of string"
'("a" "list" "of" "strings")
```

- Operations:

```
(operator operand1 operand2)
(* 2 3 4)
; => 24
(str "Hello, " "Piotr")
; => "Hello, Piotr"
```

Functions calls are not the only expressions which are operations:

- macro calls,
- special forms i.e. function definition.

What differs them from function calls:

- operand evaluation,
- cannot pass them as arguments.

Functions calls are not the only expressions which are operations:

- macro calls,
- special forms i.e. function definition.

What differs them from function calls:

- operand evaluation,
- cannot pass them as arguments.

Arity

```
(defn greet
    "Optional docstring"
    ([greeting name]
        (str greeting name))
    ([name]
        (greet "Hello, " name)))

(greet "Piotr")
; => "Hello, Piotr"
(greet "Welcome, " "Piotr")
; => "Welcome, Piotr"
(greet "Welcome, " "Piotr" "Agata")
; => ArityException Wrong number of args (3)...
```

Destructing assignment

```
(defn greet-only-2-first
    [[first second & rest]]
    (println (greet first))
    (println (greet second))
    (println (str "Others we don't greet: "
                  (clojure.string/join ", " rest))))

(greet-only-2-first ["Piotr", "Agata", "Joanna", "Pawel"])
; => Hello, Piotr
; => Hello, Agata
; => Others we don't greet: Joanna, Pawel
```

- Even more useful for hash maps!

Anonymous functions

```
(defn greetings-replacer-maker
    [replacee-regexp replacer]
    #(clojure.string/replace % replacee-regexp replacer))
    ; or (fn [args] ...)
(def greetings '("Hi, Piotr" "Hi, Agata"))
(def hi-replacer
     (greetings-replacer-maker #"^Hi" "Hello"))
(map hi-replacer greetings)
; => ("Hello, Piotr" "Hello, Agata")
```

- Functions can be returned just like data, such functions are called *closures*!

Function equality

"All functions are equal, but some functions are more equal than others."

How *Clojure* treats built-in functions?

```
(defn +
    [first & rest]
    (reduce - first (map #(* -1 %) rest)))
; => WARNING: + already refers to: #'clojure.core/+
; => in namespace...
```

If statement

```
(if true
    "Maybe I'll be shown"
    (println "Or maybe I will be"))
; => "Maybe I'll be shown"

(if false
    "Maybe I'll be shown"
    (println "Or maybe I will be"))
; => "Or maybe I will be"

(if false
    "Maybe I'll be shown")
; => nil
```

When statement

- What if multiple operations should be done?

```
(if cond?
    (do (do-sth)
        do-sth-more))
; Which is same as
(when cond?
    (do-sth)
    (do-sth-more))
```

- As in any functional language `if`, `when` should not be overused

Loop statement

```
(defn my-sum
    [seq]
    (loop [remaining-seq seq
           seq-sum 0]
        (if (empty? remaining-seq)
            seq-sum
            (let [[head-seq & tail-seq] remaining-seq]
                (recur tail-seq (+ seq-sum head-seq))))))
```

- Many loops in *Clojure* could be replaced with clear recursion, but loop is more optimized and usually more concise
- What is the effect of `(recur)`?

Loop statement

```
(defn my-sum
    [seq]
    (loop [remaining-seq seq
           seq-sum 0]
        (if (empty? remaining-seq)
            seq-sum
            (let [[head-seq & tail-seq] remaining-seq]
                (recur tail-seq (+ seq-sum head-seq))))))
```

- Many loops in *Clojure* could be replaced with clear recursion, but loop is more optimized and usually more concise
- What is the effect of `(recur)`?

Numbers, strings and vectors

- Numbers: `93 1.2 5/4`
- Strings: `"Some string" "\"Quotation\""`
- Vectors:

```
[0 1 2]
(vector 0 1 2)
; => [0 1 2]

(first [0 1 2])
(get [0 1 2] 0)
; => 0

(into [1 2 3] [4 5 6])
; => [1 2 3 4 5 6]
```

Lists

```
'(0 1 2)
(list 1 2 3)
; => (0 1 2)

(get '(0 1 2) 0)
; => nil
(nth '(0 1 2) 0)
; => 0

(into '(1 2 3) '(4 5 6))
; => (6 5 4 1 2 3)
```

Hash sets

```
#{:a :b 5}
(hash-set :a :b 5)
; => #{:a :b 5}

(get #{:a :b 5} :a)
(:a #{:a :b 5})
; => :a

(contains? #{:a :b 5} 6)
; => false

(into #{:a :b 5} #{:a :c})
; => #{:c :b 5 :a}
```

Hash maps

```
{:a 3 :b 5}
(hash-map :a 3 :b 5)
; => {:a 3 :b 5}

(get {:a 3 :b 5} :a)
(:a {:a 3 :b 5})
; => 3
(:d {:a 3 :b 5} "Not found")
; => Not found

(into {:a 3 :b 5} {:c 7})
; => {:a 3, :b 5, :c 7}
```

- Lots of functions apply to **all** *seqs* like `into` does

Programming epigram

"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."
by Alan Perlis

- Bear in mind that *Clojure* works on lists:

```
(map #(first %) {:a 1 :b 2})
; => (:a :b)

(take 3 [1 2 3 4 5 6 7 8 9 10])
; => (1 2 3)
```

Map and reduce

```
(defn pow
    [base exp]
    (reduce * (repeat exp base)))
(pow 2 3)
; => 8

(into {}
    (map
        (fn [[key val]] [key (inc val)])
        {:max 30 :min 10}))
(reduce (fn [new-map [key val]]
         (assoc new-map key (inc val)))
        {}
        {:max 30 :min 10})
; => {:max 31, :min 11}
```

Map and reduce

```
(into {}
      (map vector [:key1 :key2] ["val1" "val2"]))
; => {:key1 "val1", :key2 "val2"}
```

- Some believe that `apply` is the same as `reduce`:

```
(defn my-into
      [target additions]
      (apply conj target additions))
      ; or reduce
(my-into [] '(1 2))
; => [1 2]

(apply hash-map [:a 5 :b 6])
; => {:a 5, :b 6}
(reduce hash-map [:a 5 :b 6])
; => {{{:a 5} :b} 6}
```

Lazy seqs

```
(def database [{:name "Piotr" :other-field? true}
               {:name "Agata" :other-field? true}
               {:name "Marcin" :other-field? false}
               {:name "Maciej" :other-field? true}
               ; 100 entries altogether
               {:name "Joanna" :other-field? true}])

(defn details
    [index]
    (Thread/sleep 1000)
    (get database index))

(defn find-special-one
    [indices]
    (first (filter (complement :other-field?)
                   (map details indices))))
```

Lazy seqs

```
(time (find-special-one (range 0 99)))
; => {:name "Marcin", :other-field? false}
```

- But *Clojure* will also tell us about execution time. How long will it be?
- *Clojure* doesn't apply function passed to map until it is necessary. But when it is necessary, it does it in whole chunks!

Lazy seqs

```
(time (find-special-one (range 0 99)))
; => {:name "Marcin", :other-field? false}
```

- But *Clojure* will also tell us about execution time. How long will it be?
  ```
  ; => "Elapsed time: 32007.177964 msecs"
  ```
- *Clojure* doesn't apply function passed to map until it is necessary. But when it is necessary, it does it in whole chunks!

Lazy seqs

```
(time (find-special-one (range 0 99)))
; => {:name "Marcin", :other-field? false}
```

- But *Clojure* will also tell us about execution time. How long will it be?
  ```
  ; => "Elapsed time: 32007.177964 msecs"
  ```
- *Clojure* doesn't apply function passed to map until it is necessary. But when it is necessary, it does it in whole chunks!

- Too vast topic to cover it during this lecture
- Usage of JVM threads
- Simple tools: *futures*, *delays*, *promises*
- And sophisticated ones: *atoms*, *pmap*

- Too vast topic to cover it during this lecture
- Usage of JVM threads
- Simple tools: *futures*, *delays*, *promises*
- And sophisticated ones: *atoms*, *pmap*

- Too vast topic to cover it during this lecture
- Usage of JVM threads
- Simple tools: *futures*, *delays*, *promises*
- And sophisticated ones: *atoms*, *pmap*

- Too vast topic to cover it during this lecture
- Usage of JVM threads
- Simple tools: *futures*, *delays*, *promises*
- And sophisticated ones: *atoms*, *pmap*

- Leiningen – all-in-one: REPL, dependency management, testing, deployment etc.
- Emacs – environment closely bound with *Lisps*
- Fireplace.vim – plugin for Vim
- Counterclockwise – plugin for Eclipse
- Cursive or La Clojure – plugins for IntelliJ
- Enhanced Clojure + Sublime REPL – plugins for Sublime Text

- Leiningen – all-in-one: REPL, dependency management, testing, deployment etc.
- Emacs – environment closely bound with *Lisps*
- Fireplace.vim – plugin for Vim
- Counterclockwise – plugin for Eclipse
- Cursive or La Clojure – plugins for IntelliJ
- Enhanced Clojure + Sublime REPL – plugins for Sublime Text

- Leiningen – all-in-one: REPL, dependency management, testing, deployment etc.
- Emacs – environment closely bound with *Lisps*
- Fireplace.vim – plugin for Vim
- Counterclockwise – plugin for Eclipse
- Cursive or La Clojure – plugins for IntelliJ
- Enhanced Clojure + Sublime REPL – plugins for Sublime Text

- Leiningen – all-in-one: REPL, dependency management, testing, deployment etc.
- Emacs – environment closely bound with *Lisps*
- Fireplace.vim – plugin for Vim
- Counterclockwise – plugin for Eclipse
- Cursive or La Clojure – plugins for IntelliJ
- Enhanced Clojure + Sublime REPL – plugins for Sublime Text

- Leiningen – all-in-one: REPL, dependency management, testing, deployment etc.
- Emacs – environment closely bound with *Lisps*
- Fireplace.vim – plugin for Vim
- Counterclockwise – plugin for Eclipse
- Cursive or La Clojure – plugins for IntelliJ
- Enhanced Clojure + Sublime REPL – plugins for Sublime Text

- Leiningen – all-in-one: REPL, dependency management, testing, deployment etc.
- Emacs – environment closely bound with *Lisps*
- Fireplace.vim – plugin for Vim
- Counterclockwise – plugin for Eclipse
- Cursive or La Clojure – plugins for IntelliJ
- Enhanced Clojure + Sublime REPL – plugins for Sublime Text

- Ring + Compojure + Noir (lib-noir)
  - an example of toolset for web development
- Seesaw
  - "Seesaw turns the Horror of *Swing* into a friendly, well-documented, *Clojure* library"
- Incanter
  - data analysis package
- clojure.test

- Ring + Compojure + Noir (lib-noir)
  - an example of toolset for web development
- Seesaw
  - "Seesaw turns the Horror of *Swing* into a friendly, well-documented, *Clojure* library"
- Incanter
  - data analysis package
- clojure.test

- Ring + Compojure + Noir (lib-noir)
  - an example of toolset for web development
- Seesaw
  - "Seesaw turns the Horror of *Swing* into a friendly, well-documented, *Clojure* library"
- Incanter
  - data analysis package
- clojure.test

- Ring + Compojure + Noir (lib-noir)
  - an example of toolset for web development
- Seesaw
  - "Seesaw turns the Horror of *Swing* into a friendly, well-documented, *Clojure* library"
- Incanter
  - data analysis package
- clojure.test

- Daniel Higginbotham, *Clojure for the Brave and True*, available at `http://www.braveclojure.com`

- Rich Hickey's profiles
  - GitHub: `https://github.com/richhickey`
  - Twitter: `https://twitter.com/richhickey`
- *Clojure* home site: `http://clojure.org`
- *Clojure* Google Group: `https://groups.google.com/forum/#!forum/clojure`
- ClojureDocs – community-powered documentation: `http://clojuredocs.org`
- *Clojure* Github repository: `https://github.com/clojure/clojure`
- 4clojure – problem listing: `http://www.4clojure.com/problems`
- ClojureTV on Youtube: `https://www.youtube.com/user/ClojureTV`

- Rich Hickey's profiles
  - GitHub: `https://github.com/richhickey`
  - Twitter: `https://twitter.com/richhickey`
- *Clojure* home site: `http://clojure.org`
- *Clojure* Google Group:
  `https://groups.google.com/forum/#!forum/clojure`
- ClojureDocs – community-powered documentation:
  `http://clojuredocs.org`
- *Clojure* Github repository:
  `https://github.com/clojure/clojure`
- 4clojure – problem listing:
  `http://www.4clojure.com/problems`
- ClojureTV on Youtube:
  `https://www.youtube.com/user/ClojureTV`

Thank you for your attention and cooperation.

Any questions?