# Everyone set up?

http://bit.ly/clojure-ws

# Intro to Clojure

Alf Kristian Støyle
Anders Furseth
Kristian Frøhlich Hansen

KODEMAKER

# Clojure workshop

- Basic syntax
- Data structures
- Functions
- Reading code
- More functions
- Side effects
- Destructuring
- Structuring programs

# Clojure

- General purpose
- Lisp (List Processing)
- Functional
- Compiled
- Dynamically typed

# Literals

```
(class "String")    ;=> java.lang.String
(class #"regex")    ;=> java.util.regex.Pattern
(class 123)         ;=> java.lang.Integer
(class 2147483648)  ;=> java.lang.Long
(class 123M)        ;=> java.math.BigDecimal
(class 123N)        ;=> clojure.lang.BigInt
(class true)        ;=> java.lang.Boolean
(class false)       ;=> java.lang.Boolean
(class 42/43)       ;=> clojure.lang.Ratio
(class \c)          ;=> java.lang.Character
(class 'foo)        ;=> clojure.lang.Symbol
(class :bar)        ;=> clojure.lang.Keyword
(class nil)         ;=> nil
```

# Collection literals

```
; List
'(3 2 1) -> (list 3 2 1)

; Vector
[1 2 3] -> (vector 1 2 3)

; Set
#{1 2 3} -> (hash-set 1 2 3)

; Map
{:one "one", :two "two", :three "three"} ->
(hash-map 1 "one", 2 "two", 3 "three")
```

This is a list

```
'(+ 1 2)
;=> (+ 1 2)
```

This is an expression (form)

```
(+ 1 2)
;=> 3
```

# Prefix notation

```
(+ 1 2)
;=> 3
```

This is an expression (form)

```
(apply + '(1 2))
;=> 3
```

# Functions

```
(fn [n] (* 2 n))
```

```
;=> #<core$eval376$fn__377 user$eval376$fn__377@5c76458f>
```

# Functions

```
((fn [n] (* 2 n)) 4)
;=> 8
```

# Functions

```
(fn [n] (* 2 n))
#(* 2 %)
#(* 2 %1)
```

# Functions & Vars

```
(def times-two
  (fn [n] (* 2 n)))
;=> #'user/times-two

(times-two 4)

;=> 8
```

# Functions & Vars

```clojure
(def times-two
  (fn [n] (* 2 n)))
```

```clojure
(defn times-two
  [n] (* 2 n))
```

# Arity

```
(defn times [a b] (* a b))
```

```
(times 1 2)
;=> 2
```

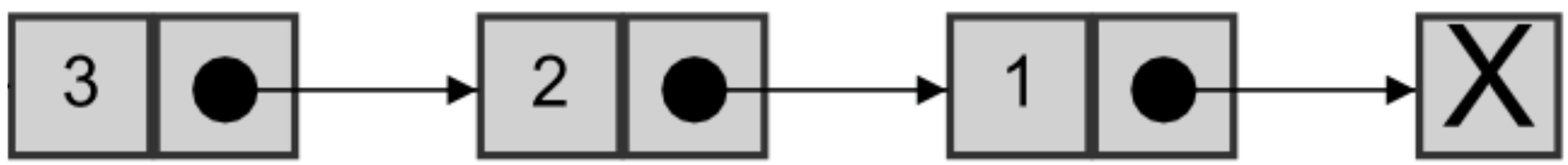
```
(times 1 2 3)
;=> ArityException Wrong number of args (3) passed to: user/times ...
```

```
(defn times [& s] (apply * s))
```

```
(times 1)
;=> 1
(times 1 2)
;=> 2
(times 1 2 3)
;=> 6
```

# Immutable and persistent data structures

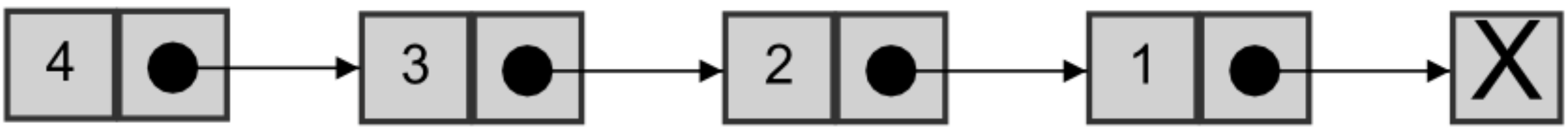
```
(def my-list '(3 2 1))
; => (3 2 1)
```



# Immutable and persistent data structures

```
(def my-list '(3 2 1))
; => (3 2 1)
```

```
(def my-other-list (cons 4 my-list))
; => (4 3 2 1)
```



# Immutable collections

```
; List
'(3 2 1)

; Vector
[1 2 3]

; Set
#{1 2 3}

; Map
{1 "one", 2 "two", 3 "three"}
```

One dash: _ => 0

Two dashes: __ => A function

Three dashes: ___ => A value (list)

```clojure
(deftest very-basic-types
  ; Which number is the same as 1? :)
  (is (= 1 _)))
```

```clojure
(deftest very-basic-types
  ; Which number is the same as 1? :)
  (is (= 1 1)))
```

```
(deftest basic-types-with-are
  (are [x y] (= x y)
  (+ 0 1)  ___
  (= 1 1)  ___))
```

```clojure
(deftest basic-types-with-are
  (are [x y] (= x y)
    (+ 0 1) 1
    (= 1 1) true))
```

# Light table

- Ctrl-space - Commands
- Tab - autocomplete
- Ctrl-d - Shows doc-string of current var
- Ctrl/Cmd-enter - Evaluate current form
- Ctrl/Cmd-shift-space - Evaluate entire file
- Instarepl
- lein midje :autotest

# Exercise 1

- lein midje :autotest
- test/clojure_workshop_flatmap/ex_1.clj
- Remove #_ from each test
- Fix tests
- Lots of hints in the comments

http://bit.ly/flatmap-clojure-ws

http://clojure.github.io/clojure/
**http://bit.ly/clj-cheatsheet**
**http://clojuredocs.org/clojure_core**

(Solutions in the solution branch)

# Reading code

# Nesting

```
(- 10 (+ 1 2 (+ 3 4)))
```

# Reads inside out

```
(- 10 (+ 1 2 (+ 3 4)))
```

# let

```
(- 10 (+ 1 2 (+ 3 4)))
```

```
(let [inner  (+ 3 4)
      middle (+ 1 2 inner)
      ten    10
      result (- ten middle)]
  result)
```

# Reads inside out

```
(- 10 (+ 1 2 (+ 3 4)))
```

```
(if true
  (println "true")
  (println "false"))
```

# if

```
(if true
  (println "true")
  (println "false"))
```

```
"true"
;=> nil
```

# if

```
(if thruthy
  (println "true")
  (println "false"))
```

# Everything is an expression

```
(if true
  "true"
  "false")
;=> "true"
```

# More functions

# More functions

```
(count
  (filter ifn?
          (map #(var-get (second %))
               (ns-publics 'clojure.core))))
;=> 567
```

# first

```
(first '(1 2 3))
;=> 1

(first [1 2 3])
;=> 1

(first #{1 2 3})
;=> 1

(first {:one "one" :two  "two"})
;=> [:one "one"]
```

# rest

```
(rest '(1 2 3))
;=> (2 3)

(rest [1 2 3])
;=> (2 3)

(rest #{1 2 3})
;=> (2 3)

(rest {:one "one" :two  "two"})
;=> ([:two "two"])
```

# rest

```
(rest '())
;=> ()

(rest [])
;=> ()

(rest #{})
;=> ()

(rest {})
;=> ()
```

# next

```
(next '())
;=> nil

(next [])
;=> nil

(next #{})
;=> nil

(next {})
;=> nil
```

# collections are functions

```
('(1 2 3) 0)
```

```
([1 2 3] 0)
;=> 1

(#{3 2 1} 1)
;=> 1

({:one 1 :two 2 :three 3} :one)
;=> 1
```

# some collections are functions

```
('(1 2 3) 0)
;=> java.lang.ClassCastException: clojure.lang.PersistentList cannot be cast to clojure.lang.IFn

([1 2 3] 0)
;=> 1

(#{3 2 1} 1)
;=> 1

({:one 1 :two 2 :three 3} :one)
;=> 1
```

# conj

```
(conj '(2 1) 3)
;=> (3 2 1)

(conj [1 2] 3)
;=> [1 2 3]

(conj #{1 2} 3)
;=> #{1 2 3}

(conj {:k1 1} [:k2 2])
;=> {:k2 2, :k1 1}
```

# Filter

```
(defn filter
  "Returns a lazy sequence of the items in coll for which
  (pred item) returns true. pred must be free of side-effects."
  ([pred coll]
    ...))
```

# Filter

```
(filter #(= 0 (mod % 2)) [0 1 2 3 4 5])
;=>  (0 2 4)
```

# Filter

```
(filter even? [0 1 2 3 4 5])
;=>  (0 2 4)
```

# Map

```
(defn map
  "Returns a lazy sequence consisting of the result of applying f to the
  set of first items of each coll, followed by applying f to the set
  of second items in each coll, until any one of the colls is
  exhausted.  Any remaining items in other colls are ignored. Function
  f should accept number-of-colls arguments."
  {:added "1.0"
   :static true}
  ([f coll]
    ...))
```

# Map

```
(map #(+ 1 %) [1 2 3 4 5])
```

```
;=> (2 3 4 5 6)
```

# Reduce (Fold)

```
(defn reduce
  "f should be a function of 2 arguments. If val is not supplied,
  returns the result of applying f to the first 2 items in coll, then
  applying f to that result and the 3rd item, etc. If coll contains no
  items, f must accept no arguments as well, and reduce returns the
  result of calling f with no arguments.  If coll has only 1 item, it
  is returned and f is not called.  If val is supplied, returns the
  result of applying f to val and the first item in coll, then
  applying f to that result and the 2nd item, etc. If coll contains no
  items, returns val and f is not called."
  {:added "1.0"}
  ([f coll]
     ...)
  ([f val coll]
     ...))
```

# Reduce (Fold)

```
(reduce #(+ %1 %2) [1 2 3 4 5])
;=> 15
```

# Reduce (Fold)

```
(reduce + [1 2 3 4 5])
```

```
;=> 15
```

# Reduce (Fold)

```
(reduce conj () [1 2 3 4 5])
```

```
;=> (5 4 3 2 1)
```

Side-effects

IO

Mutability

# IO (side-effects)

```clojure
(defn foo []
  (println "Doing IO, and returning 1")
  1)

(do
(println "Doing IO, and returning 1")
 1)
```

```clojure
(if true
  (do
    (println "in true part")
    1)
  (do
    (println "in else parte")
    2))
```

# IO (side-effects)

```clojure
(defn foo []
  (println "Doing IO, and returning 1")
  1)

(do
  (println "Doing IO, and returning 1")
  1)

(dotimes [x 10]
  (println x))

(doseq [x (range 10)]
  (println x))
```

# Mutability (side-effects)

```
(atom 0)
;=> #<Atom@4b8baa34: 0>

(def an-atom (atom 0))
;=> #'user/an-atom

(swap! an-atom inc)
;=> 1

(deref an-atom)
;=> 1
@an-atom
;=> 1
```

# Midje syntax

```
(fact "A couple of things about midje"
  1 => 1
  (throw (Exception.)) => (throws Exception))
```

# Exercise 2

- test/clojure_workshop_flatmap/ex_2.clj
- Remove #_ from each test
- Fix tests
- Lots of hints in the comments
- lein midje :autotest

**http://bit.ly/clj-cheatsheet**

# Destructuring

# Destructuring

```clojure
(defn foo [s]
  (let [one (first s)
        two (fnext s)]
    (println one two)))

(defn foo [s]
  (let [[one two] s]
    (println one two)))
```

# Destructuring

```clojure
(defn bar [m]
  (let [one (:one m)
        two (:two m)]
    (println one two)))

(defn bar [m]
  (let [{one :one two :two} m]
    (println one two)))
```

# Destructuring

```clojure
(defn bar [m]
  (let [{one :one two :two} m]
    (println one two)))
```

```clojure
(defn bar [{one :one two :two}]
  (println one two))
```

```clojure
(defn foo [s]
  (let [[one two] s]
    (println one two)))
```

```clojure
(defn foo [[one two]]
  (println one two))
```

# Putting an app together

- Organising code
- Namespaces
- Main

# Namespaces

```
(ns clojure-workshop-flatmap.cat
 (:use clojure.core)
 (:require
  [clojure.string :as string]))
```

# Namespaces

```clojure
(ns clojure-workshop-flatmap.cat
 (:use clojure.core)
 (:require
  [clojure.string :as string]))
```

```clojure
(clojure.core/map inc [1 2 3 4 5])
```

# Namespaces

```clojure
(ns clojure-workshop-flatmap.cat
 (:use clojure.core)
 (:require
  [clojure.string :as string]))
```

```clojure
(clojure.core/map inc [1 2 3 4 5])
```

```clojure
(map inc [1 2 3 4 5])
```

# Namespaces

```clojure
(ns clojure-workshop-flatmap.cat
 (:use clojure.core)
 (:require
  [clojure.string :as string]))
```

```clojure
(clojure.core/map inc [1 2 3 4 5])
```

```clojure
(map inc [1 2 3 4 5])
```

```clojure
(clojure.string/join \newline coll)
```

# Namespaces

```clojure
(ns clojure-workshop-flatmap.cat
 (:use clojure.core)
 (:require
  [clojure.string :as string]))
```

```clojure
(clojure.core/map inc [1 2 3 4 5])
(map inc [1 2 3 4 5])
(clojure.string/join \newline coll)
(string/join \newline coll)
```

# Main

```clojure
(ns clojure-workshop-flatmap.cat
 (:gen-class :main)
 (:require
  [clojure.string :as string]))

(defn- say-hello []
  (println "Hello world!"))

(defn -main [& args]
  (say-hello))
```

# Exercise 3

- Implement the cat command. Tests are in test/clojure_workshop_flatmap/cat.clj
- Shell of program is there, you should provide logic
- Remove #_ from each test
- ‘lein uberjar’ build executable.
- ‘java -jar target/cat.jar -h’ for usage.

**http://bit.ly/clj-cheatsheet**

# Where to go next?

- Recursion
- Laziness
- Reference types (atoms, agents, refs, vars)
- Multi methods
- Protocols
- Macros
- Java interop
- Libraries

# Where to go next?

http://kodemaker.no/clojure

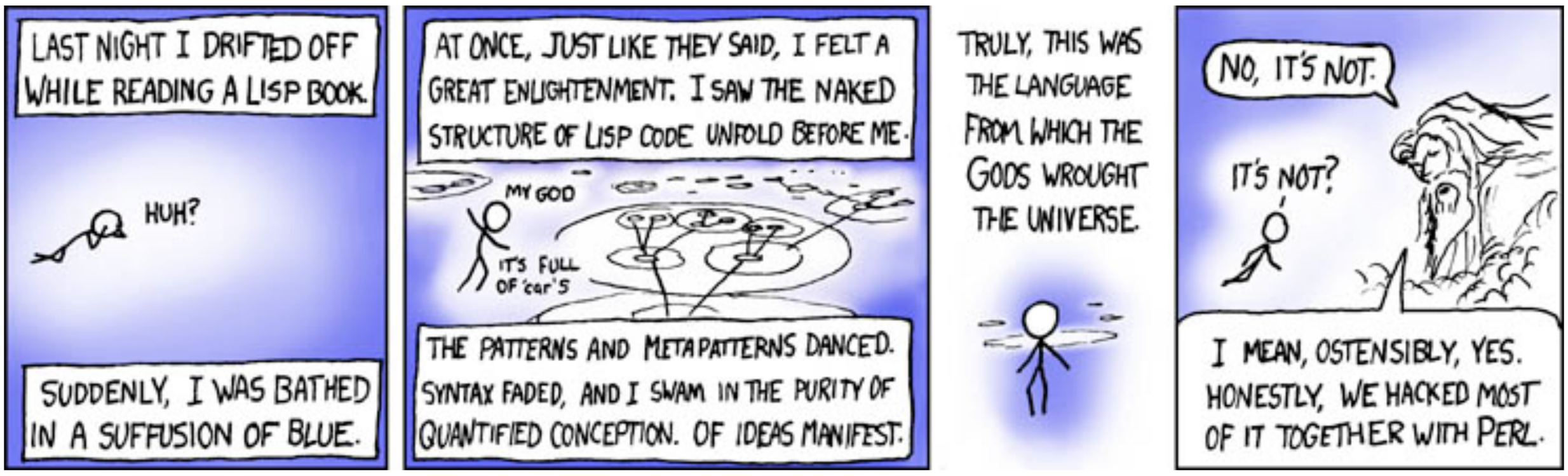
http://clojure.org/books

https://groups.google.com/forum/#!forum/clojure

http://www.4clojure.com/

Thank you!
clojure.org

alf.kristian@kodemaker.no
anders@kodemaker.no
kristian@kodemaker.no




http://xkcd.com/224/