Clojure Functional Lisp on the JVM

START HERE

`clj`
Start a REPL

`(map inc [1 2 3])`
Transform a sequence

`(def x 42)`
Bind a global name

`(->> xs (filter even?) (map inc))`
Thread a pipeline

`(defn f [x] (* x x))`
Define a function

`{:a 1 :b 2}`
Map literal

`(let [a 1 b 2] (+ a b))`
Local bindings

`(doc map)`
Show docs in the REPL

Source form (+ 1 2) → **Data structure** list of symbols → **JVM bytecode** on the fly → **Value** 3

CONCEPTS

Code is data: everything is (function arg ...). Collections are immutable, so assoc and conj return new values.

Only nil and false are falsey: 0, "" and () are all true.

Sequence functions are lazy; force them with doall, doseq or a REPL print.

COMMAND LINE

`clj`
Start a REPL with line editing (rlwrap)

`clojure`
Run without rlwrap, for scripts and pipes

`clojure -M -m <ns>`
Run the -main function of a namespace

`clojure -M <file.clj>`
Run a script file

`clojure -M:<alias>`
Run with a deps.edn alias

`clojure -X <ns>/<fn> :k v`
Call a function with a map of args

`clojure -Spath`
Print the computed classpath

`clojure -Sdeps '<edn>'`
Add deps.edn data from the command line

REPL

`(doc <name>)`
Show docs for a name

`(source <name>)`
Print the source of a function

`*1 / *2 / *e`
Last results / last exception

`(require '[<ns> :as <a>] :reload)`
Load a namespace, re-reading its file

BASICS

`(def x 1)`
Define a var in the namespace

`(defn f "doc" [a & more] ...)`
Function with docstring and rest args

`(fn [x] ...) / #(* % 2)`
Anonymous function, short form

`(let [a 1 b (+ a 1)] ...)`
Local bindings, sequential

BASICS (cont.)

`(if <t> <a> <b>) / (when <t> ...)`
Conditional / one-armed with body

`(cond <t1> <a> :else <b>)`
Multi-way conditional

`(loop [i 0] (recur (inc i)))`
Loop with explicit recur

`(doseq [x xs] ...)`
Loop for side effects

`(for [x xs :when (odd? x)] ...)`
Lazy list comprehension

`(str "a" 1 :b)`
Concatenate as strings

`(println ...) / (prn ...)`
Print for humans / as readable data

`(try ... (catch Exception e ...) (finally ...))`
Handle errors

`(ex-info "msg" {:k v})`
Exception carrying data

`(nil? x) / (some? x)`
Nil tests

`(= a b) / (not= a b)`
Value equality

COLLECTIONS

`[1 2 3] / '(1 2 3)`
Vector / list literal

`{:a 1 :b 2}`
Map literal

`#{1 2 3}`
Set literal

`(conj coll x)`
Add: end of vector, front of list

`(assoc m :k v)`
Add or replace a key (or vector index)

`(dissoc m :k)`
Remove a key

`(get m :k <default>) / (:k m)`
Look up a key

`(get-in m [:a :b])`
Look up a nested path

`(assoc-in m [:a :b] v)`
Set a nested path

`(update m :k inc)`
Apply a function to a value

`(update-in m [:a :b] f)`
Apply a function at a nested path

`(merge m1 m2)`
Combine maps, later wins

`(keys m) / (vals m)`
Keys / values of a map

COLLECTIONS (cont.)

`(count c) / (empty? c)`
Size / emptiness

`(into [] xs)`
Pour a sequence into a collection

`(group-by f xs)`
Map from f result to items

`(frequencies xs)`
Count occurrences

SEQUENCES

`(map f xs)`
Apply f to each item, lazily

`(filter pred xs) / (remove pred xs)`
Keep / drop matching items

`(reduce f init xs)`
Fold into one value

`(first xs) / (rest xs)`
Head / tail

`(take n xs) / (drop n xs)`
First n / all but first n

`(range 5) / (range 1 10 2)`
Number sequence

`(apply f args)`
Call f with a sequence as arguments

`(sort-by f xs)`
Sort by a key function

`(some pred xs) / (every? pred xs)`
Any match / all match

FUNCTIONS & MACROS

`(-> x (f a) (g b))`
Thread x as first argument

`(->> xs (map f) (filter p))`
Thread as last argument

`(comp f g)`
Compose, applied right to left

`(partial f a)`
Fix leading arguments

`(fn [{:keys [a b]}] ...)`
Destructure a map argument

`(let [[a b & r] xs] ...)`
Destructure a sequence

`(let [{:keys [a] :or {a 0}} m] ...)`
Destructure with defaults

STATE & CONCURRENCY

`(def a (atom 0))`
Create an atom

`@a / (deref a)`
Read the current value

STATE & CONCURRENCY (cont.)

`(swap! a f args)`
Atomically update with f

`(reset! a v)`
Set a new value

NAMESPACES & JAVA

`(ns <name> (:require [<lib> :as <a>]))`
Declare a namespace and its deps

`(:import (java.util Date))`
Import Java classes in ns

`(require '[clojure.string :as str])`
Load a library with an alias

`(str/join ", " xs)`
Join strings

`(str/split s #",")`
Split by regex

`(Math/abs -1)`
Call a static method

`(.method obj args)`
Call an instance method

`(slurp "<file>") / (spit "<file>" s)`
Read / write a whole file

DEPS.EDN

deps.edn

```
{:paths ["src"]
 :deps {org.clojure/clojure
{:mvn/version "1.11.1"}
        metosin/malli {:mvn/
version "<ver>"}}
 :aliases
 {:test {:extra-paths ["test"]}
  :dev  {:extra-deps {<lib>
{:git/url "<url>" :git/sha
"<sha>"}}}}}
```

Keys: :paths, :deps, :aliases. A dep is a lib symbol mapped to :mvn/version, :git/sha or :local/root.

RECIPES

New project, run main

```
mkdir app && cd app && mkdir
src
echo '{:paths ["src"]}' >
deps.edn
# src/app.clj: (ns app) (defn -
main [& args] (println args))
clojure -M -m app hello
```

Pipeline over data

```
(->> (range 100)
     (filter odd?)
     (map #(* % %))
     (take 5)
     (reduce +))
```

Word frequencies

```
(->> (slurp "book.txt")
     (re-seq #"\w+")
     frequencies
     (sort-by val >) (take 10))
```