

Clojure Cheatsheet

Tooling & REPL CLI (deps.edn)

- Start REPL: clj
- Run main: clj -M -m my.ns
- Add alias: clj -M:dev (defined in deps.edn)

Leiningen (older/common)

- lein repl, lein test, lein run

```
Namespace      skeleton      (ns my.ns      (:require [clojure.string :as str]))
```

Syntax & Evaluation Forms

- Everything is a form; lists evaluate as calls: (f a b)
- '(1 2 3) quote (do not eval)
- '(a x) syntax-quote + unquote

Literals

- Numbers: 42, 3.14, ratios 1/3
- Strings: "hi" Chars: \a
- Keywords: :name Symbols: foo
- Regex: #"\d+"

Comments

- Line: ; ...
- Form: #_(+ 1 2) (skips next form)

Bindings & Definitions Global defs

- (def x 10)
- (defn add [a b] (+ a b))
- Docstring: (defn f "doc" [x] ...)

Local bindings

- (let [x 1 y 2] (+ x y))
- Destructuring supported in let/args

Anonymous functions

- (fn [x] (+ x 1))
- Shorthand: #(+ % 1) multi-args: #(+ %1 %2)

Core Data Structures (Immutable) List, Vector, Map, Set

- List: '(1 2 3) (linked list; great for seq ops)
- Vector: [1 2 3] (indexed)
- Map: {:a 1 :b 2}
- Set: #{1 2 3}

Common ops

- Get: (get m :a); keywords are fns: (:a m)
- Assoc/dissoc: (assoc m :c 3), (dissoc m :b)
- Update: (update m :a inc)
- Conj: (conj [1 2] 3); (conj '(1 2) 0) (diff seman-

- tics)
- Contains? (keys for maps): (contains? m :a)

Nested updates

- (assoc-in m [:a :b] 1)
- (update-in m [:a :b] inc)
- (get-in m [:a :b])

Seqs & Collection Processing Seq model

- Many collections produce a sequence view via seq
- Seq ops are lazy (often); realize with doall if needed

Bread & butter

- map, filter, remove
- reduce, into
- take, drop, partition, group-by
- some, every?, not-any?

Threading macros (pipeline)

- Thread-first: (-> x f (g 1) h)
- Thread-last: (-> xs (map f) (filter p) (take 10))

Comprehension (for [x (range 5) :when (odd? x)] (* x x))

Conditionals & Control If / when

- (if cond then else)
- (when cond ...) (no else; returns nil if false)

Cond / case

- (cond p1 e1 p2 e2 :else e)
- (case x 1 "one" 2 "two" "other") (constants only)

Short-circuit

- (and a b c) (or a b c)

Loops

- Recursion: recur (tail-call to loop/fn)
- ```
(loop [i 0 acc 0] (if (= i 10) acc (recur (inc i) (+ acc i))))
```

## Destructuring (Very Useful) Vector destructuring

- (let [[a b & rest] [1 2 3 4]] ...)
- (let [[x \_ y] [1 2 3]] ...) (ignore)

## Map destructuring

- (let [{:keys [a b]} {:a 1 :b 2}] ...)
- Rename: (let [{:keys [a] :as m} ...] ...)
- With defaults: (let [{:keys [a] :or {a 0}} ...] ...)

## Functions & Higher-Order Patterns Arity & variadic

- Multi-arity:
- ```
(defn f ([x] (f x 0)) ([x y] (+ x y)))
```
- Variadic: (defn sum [& xs] (reduce + xs))
 - Apply: (apply + [1 2 3])
 - Partial: (partial + 10)
 - Complement: (complement pred)

Memoization

- (def fast-f (memoize slow-f))

State & Concurrency Primitives Atoms (sync, independent state)

- (def a (atom 0))
- Read: @a Set: (reset! a 1)
- Update: (swap! a inc)

Refs + STM (coordinated state)

- (def r (ref 0))
- In transaction: (dosync (alter r inc))

Agents (async updates)

- (def ag (agent 0))
- (send ag inc); await: (await ag)

Futures / Promises / Delays

- (future (do-work)) then @f
- (promise) + (deliver p v) then @p
- (delay expr) then (force d)

core.async (library)

- CSP-style channels + go blocks; great for pipelines

Interop & Exceptions Java interop

- Construct: (java.util.Date.)
- Call method: (.toString d)
- Static: (Math/sqrt 9)
- Field: (.-x obj)
- Doto:

```
(doto (java.util.ArrayList.) (.add 1) (.add 2))
```

Exceptions

- Throw: (throw (ex-info "bad" {:x 1}))
- Try/catch/finally:

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

Common pattern

- Prefer ex-info for structured data; use ex-data to read

it

Macros (Basics) What macros are

- Code that transforms code (compile-time-ish)
- Use functions first; macros when you need control over evaluation

```
Define macro (defmacro unless [pred & body] '(if (not pred) (do @body) nil))
```

Inspect expansion

- (macroexpand-1 '(-> x f))

Spec, Tests, Common Libs Spec (clojure.spec.alpha)

- (s/def ::id int?)
- Validate: (s/valid? ::id 1)
- Explain: (s/explain ::id "x")

Testing

- clojure.test: (deftest ... (is (= 2 (+ 1 1))))
- Run: via clj -X:test (depends on setup) or Lein

Ecosystem staples

- clojure.string, clojure.set
- EDN: clojure.edn
- Data transforms: transducers
- Web: Ring/Compojure/Reitit
- Data: next.jdbc, honeysql

Common Idioms (Quick Recipes) Nil-punning (be explicit)

- nil means “no value” and is falsey
- Prefer some? if you mean “not nil”

Use maps as configs

- Functions often take an options map: (f x {:timeout 100})

Transducer sketch

- (into [] (comp (filter p) (map f)) xs)
- Avoid intermediate collections

Prefer pure core + small boundary

- Keep business logic pure; push IO/DB to edges

Mini Reference

- Truthiness: only nil and false are falsey
- Keywords as fns: (:k m) and maps as fns: (m :k)
- Equality: = value equality; identical? reference
- Common print: pr-str (readable), println (human)