

Clojure Cheat Sheet (Clojure 1.3, sheet v1.4)

Documentation

clojure.repl/ doc find-doc apropos source pst javadoc
(foo.bar/ is namespace for later syms)

Primitives

Numbers

Literals	Long: 7 BigInt: 7N Ratio: -22/7 Double: 2.78 BigDecimal: 4.2M
Arithmetic	+ - * / quot rem mod inc dec max min
Compare	= == not= < > <= >= compare
Bitwise	bit-{and, or, xor, not, re-flip, set, shift-right, shift-left, and-not, clear, test}
Cast	byte short int long float double bigdec bigint num rationalize biginteger
Test	nil? identical? zero? pos? neg? even? odd?
Random	rand rand-int
BigInt	with-precision
Unchecked	unchecked-{add, dec, divide, inc, multiply, negate, remainder, subtract}-int

Strings

Create	str format See also IO/to string
Use	count get subs compare (clojure.string/) join escape split split-lines replace replace-first reverse (String) .indexOf .lastIndexOf
Regex	#"pattern" re-find re-seq re-matches re-pattern re-matcher re-groups (clojure.string/) replace replace-first
Letters	(clojure.string/) capitalize lower-case upper-case
Trim	(clojure.string/) trim trim-newline triml trimr
Test	char char? string? (clojure.string/) blank?

Other

Characters	char char-name-string char-escape-string
Keywords	keyword keyword? find-keyword
Symbols	symbol symbol? gensym

Collections

Collections

Generic ops	count empty not-empty into conj
Content tests	distinct? empty? every? not-every? some not-any?
Capabilities	sequential? associative? sorted? counted? reversible?
Type tests	coll? list? vector? set? map? seq?

Lists

Create	'() list list*
Examine	first nth peek .indexOf .lastIndexOf
'Change'	cons conj rest pop

Vectors

Create	[] vector vec vector-of
Examine	(my-vec idx) → (nth my-vec idx) get peek .indexOf .lastIndexOf
'Change'	assoc pop subvec replace conj rseq

Sets

Create	#{} set hash-set sorted-set sorted-set-by
Examine	(my-set item) → (get my-set item) contains?
'Change'	conj disj
Rel algebra	(clojure.set/) join select project union difference intersection
Get map	(clojure.set/) index rename-keys rename map-invert
Test	(clojure.set/) subset? superset?

Maps

Create	{ } hash-map array-map zipmap sorted-map sorted-map-by bean frequencies group-by
Examine	(:key my-map) → (get my-map :key) get-in contains? find keys vals
'Change'	assoc assoc-in dissoc merge merge-with select-keys update-in
Entry	key val
Sorted maps	rseq subseq rsubseq

Transients (clojure.org/transients)

Create	transient persistent!
Change	conj! pop! assoc! dissoc! disj! Note: always use re- turn value for later changes, never original!

Misc

Compare	= == identical? not= not compare clojure.data/diff
Test	true? false? nil? instance?

Sequences

Creating a Lazy Seq

From collection	seq vals keys rseq subseq rsubseq
From producer fn	lazy-seq repeatedly iterate
From constant	repeat range
From other	file-seq line-seq resultset-seq re-seq tree-seq xml-seq iterator-seq enumeration-seq
From seq	keep keep-indexed

Seq in, Seq out

Get shorter	distinct filter remove for
Get longer	cons conj concat lazy-cat mapcat cycle interleave interpose
Tail-items	rest nthrest fnext nnext drop drop-while take-last for
Head-items	take take-nth take-while butlast drop-last for
'Change'	conj concat distinct flatten group-by partition partition-all partition-by split-at split-with filter remove replace shuffle
Rearrange	reverse sort sort-by compare
Process items	map pmap map-indexed mapcat for replace seque

Using a Seq

Extract item	first second last rest next ffirst nfirst fnext nnext nth nthnext rand-nth when-first max-key min-key
Construct coll	zipmap into reduce reductions set vec into-array to-array-2d
Pass to fn	apply
Search	some filter
Force evaluation	doseq dorun doall
Check for forced	realized?

Zippers (clojure.zip/)

Create	zipper seq-zip vector-zip xml-zip
Get loc	up down left right leftmost rightmost
Get seq	lefts rights path children
'Change'	make-node replace edit insert-child insert-left insert-right append-child remove
Move	next prev
Misc	root node branch? end?

IO

to/from ...	spit slurp (to writer/from reader, Socket, string with file name, URI, etc.)
to *out*	pr prn print printf println newline (clo- jure.pprint/) print-table
to writer	(clojure.pprint/) pprint cl-format also: (binding [*out* writer] ...)
to string	format with-out-str pr-str prn-str print-str println-str
from *in*	read-line read
from reader	line-seq read also: (binding [*in* reader] ...) java.io.Reader
from string	read-string with-in-str
Open	with-open (clojure.java.io/) text: reader writer binary: input-stream output-stream
Binary	(.write ostream byte-arr) (.read istream byte-arr) java.io.OutputStream java.io.InputStream GitHub: gloss byte-spec
Misc	flush (.close s) file-seq *in* *out* *err* (clojure.java.io/) file copy GitHub: fs

Functions

Create	<code>fn defn defn- definline identity constantly memfn comp complement partial juxt memoize fn! every-pred some-fn</code>
Call	<code>-> ->> apply</code>
Test	<code>fn? ifn?</code>

Abstractions

Protocols (clojure.org/protocols)

Define	<code>(defprotocol Slicey (slice [at]))</code>
Extend	<code>(extend-type String Slicey (slice [at] ...))</code>
Extend null	<code>(extend-type nil Slicey (slice [_] nil))</code>
Reify	<code>(reify Slicey (slice [at] ...))</code>

Records (clojure.org/datypes)

Define	<code>(defrecord Pair [h t])</code>
Access	<code>(:h (Pair. 1 2)) → 1</code>
Create	<code>Pair. ->Pair map->Pair</code>

Types (clojure.org/datypes)

Define	<code>(deftype Pair [h t])</code>
Access	<code>(.h (Pair. 1 2)) → 1</code>
Create	<code>Pair. ->Pair</code>
	<code>(deftype Pair [h t]</code>
With methods	<code>Object</code>
	<code>(toString [this] (str "<" h ", " t ">")))</code>

Multimethods (clojure.org/multimethods)

Define	<code>(defmulti my-mm dispatch-fn)</code>
Method define	<code>(defmethod my-mm :dispatch-value [args] ...)</code>
Dispatch	<code>get-method methods</code>
Remove	<code>remove-method remove-all-methods</code>
Prefer	<code>prefer-method prefers</code>
Relation	<code>derive isa? parents ancestors descendants make-hierarchy</code>

Macros

Create	<code>defmacro definline macroexpand-1 macroexpand</code>
Branch	<code>and or when when-not when-let when-first if-not if-let cond condp case</code>
Loop	<code>for doseq dotimes while</code>
Arrange	<code>.. doto -></code>
Scope	<code>binding locking time with-{in-str, local-vars, open, out-str, precision, redefs, redefs-fn}</code>
Lazy	<code>lazy-cat lazy-seq delay</code>
Doc.	<code>assert comment doc</code>

Reader Macros

<code>'</code>	Quote 'form → (quote form)
<code>\</code>	Character literal
<code>;</code>	Single line comment
<code>^</code>	Metadata (see Metadata section)
<code>@</code>	Deref @form → (deref form)
<code>'</code>	Syntax-quote
<code>~</code>	Unquote
<code>~@</code>	Unquote-splicing
<code>#"p"</code>	Regex Pattern p
<code>#'</code>	Var quote #'x → (var x)
<code>#()</code>	#(...) → (fn [args] (...))
<code>#_</code>	Ignore next form

Metadata (clojure.org/special_forms)

General	<code>^{:key1 val1 :key2 val2 ...}</code>
Abbrevs	<code>^Type → ^{:tag Type}, ^:key → ^{:key true}</code>
Common	<code>^:dynamic ^:private ^:static ^:const</code>
Examples	<code>(defn ^:private ^:static ^String my-fn ...)</code> <code>(def ^:dynamic *dyn-var* val)</code>
On Vars	<code>meta with-meta vary-meta alter-meta! reset-meta! doc find-doc test</code>

Special Forms (clojure.org/special_forms)

<code>def if do let quote var fn loop recur throw try monitor-enter monitor-exit</code>	
Binding Forms / Destructuring	(examples) <code>let fn defn defmacro loop for doseq if-let when-let</code>

Vars and global environment (clojure.org/vars)

Def variants	<code>def defn defn- definline defmacro defmethod defmulti defonce defrecord</code>
Interned vars	<code>declare intern binding find-var var</code>
Var objects	<code>with-local-vars var-get var-set alter-var-root var?</code>
Var validators	<code>set-validator! get-validator</code>

Namespace

Current	<code>*ns*</code>
Create/Switch	<code>(tutorial) ns in-ns create-ns</code>
Add	<code>alias def import intern refer</code>
Find	<code>all-ns find-ns</code>
Examine	<code>ns-{name, aliases, map, interns, publics, refers, imports}</code>
From symbol	<code>resolve ns-resolve namespace</code>
Remove	<code>ns-unalias ns-unmap remove-ns</code>

Loading

Load libs	<code>(tutorial) require use import refer</code>
List loaded	<code>loaded-libs</code>
Load misc	<code>load load-file load-reader load-string</code>

Concurrency

Atoms	<code>atom swap! reset! compare-and-set!</code>
Futures	<code>future future-{call, done?, cancel, cancelled?} future?</code>
Threads	<code>bound-fn bound-fn* {get, push, pop}-thread-bindings thread-bound?</code>
Misc	<code>locking pcalls pvalues pmap seque promise deliver</code>

Refs and Transactions (clojure.org/refs)

Create	<code>ref</code>
Examine	<code>deref @ (@form → (deref form))</code>
Transaction	<code>sync dosync io!</code>
In transaction	<code>ensure ref-set alter commute</code>
Validators	<code>set-validator! get-validator</code>
History	<code>ref-history-count ref-{min, max}-history</code>

Agents and Asynchronous Actions (clojure.org/agents)

Create	<code>agent</code>
Examine	<code>agent-error</code>
Change state	<code>send send-off restart-agent</code>
Block waiting	<code>await await-for</code>
Ref validators	<code>set-validator! get-validator</code>
Watchers	<code>add-watch remove-watch</code>
Thread handling	<code>shutdown-agents</code>
Error	<code>error-handler set-error-handler! error-mode set-error-mode!</code>
Misc	<code>*agent* release-pending-sends</code>

Java Interoperation (clojure.org/java_interop)

General	<code>.. doto Classname/ Classname. new bean</code> <code>comparator enumeration-seq import iterator-seq</code> <code>memfn set!</code>
Cast	<code>boolean byte short char int long float double</code> <code>bigdec bigint num cast biginteger</code>
Exceptions	<code>throw try catch finally pst</code>

Arrays

Create	<code>make-array {object, boolean, byte, short, char,</code> <code>int, long, float, double}-array aclone to-array</code> <code>to-array-2d into-array</code>
Use	<code>aget aset aset-{boolean, byte, short, char, int,</code> <code>long, float, double} alength amap areduce</code>
Cast	<code>booleans bytes shorts chars ints longs floats</code> <code>doubles</code>

Proxy

Create	<code>proxy get-proxy-class {construct, init}-proxy</code>
Misc	<code>proxy-mappings proxy-super update-proxy</code>

Other

XML	<code>clojure.xml/parse xml-seq</code>
REPL	<code>*1 *2 *3 *e *print-dup* *print-length*</code> <code>*print-level* *print-meta* *print-readably*</code>
Code	<code>*compile-files* *compile-path* *file*</code> <code>*warn-on-reflection* compile gen-class</code> <code>gen-interface loaded-libs test</code>
Misc	<code>eval force hash name *clojure-version*</code> <code>clojure-version *command-line-args*</code>