

Clojure Cheat Sheet (Clojure 1.6 - 1.9, sheet v43)

Documentation

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

Primitives

Numbers

Literals

Long: 7, hex 0xff, oct 017, base 2 2r1011, base 36 36rCRAZY BigInt: 7N Ratio: -22/7 Double: 2.78 -1.2e-5 BigDecimal: 4.2M

Arithmetic

Compare

Bitwise

Cast

Test

Random

BigDecimal

Unchecked

Strings

Other

Collections

Collections

Lists (conj, pop, & peek at beginning)

Vectors (conj, pop, & peek at end)

Sets

Maps

Queues (conj at end, peek & pop from beginning)

Relations (set of maps, each with same keys, aka rels)

Transients (clojure.org/reference/transients)

Misc

Sequences

Creating a Lazy Seq

Seq in, Seq out

Using a Seq

Transducers (clojure.org/reference/transducers)

Spec (rationale, guide)

Predicates with test.check generators

IO

Functions

Abstractions (Clojure type selection flowchart)

Protocols (clojure.org/reference/protocols)

Define	(defprotocol Slicey (slice [at]))
Extend	(extend-type String Slicey (slice [at] ...))
Extend null	(extend-type nil Slicey (slice [] nil))
Reify	(reify Slicey (slice [at] ...))
Test	satisfies? extends?
Other	extend extend-protocol extenders

Records (clojure.org/reference/datypes)

Define	(defrecord Pair [h t])
Access	(.h (Pair. 1 2)) → 1
Create	Pair. ->Pair map->Pair
Test	record?

Types (clojure.org/reference/datypes)

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

Multimethods (clojure.org/reference/multimethods)

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

Macros

Create	defmacro definline
Debug	macroexpand-1 macroexpand (clojure.walk/) macroexpand-all
Branch	and or when when-not when-let when-first if-not if-let cond condp case when-some if-some
Loop	for doseq dotimes while
Arrange	.. doto -> ->> as-> cond-> cond->> some-> some->>
Scope	binding locking time with-in-str with-local-vars with-open with-out-str with-precision with-redefs with-redefs-fn
Lazy	lazy-cat lazy-seq delay
Doc.	assert comment doc

Special Characters (clojure.org/reference/reader, guide)

,	Comma reads as white space. Often used between map key/value pairs for readability.
'	quote: 'form → (quote form)
/	Namespace separator (see Primitives/Other section)
\	Character literal (see Primitives/Other section)
:	Keyword (see Primitives/Other section)
;	Single line comment
~	Metadata (see Metadata section)
foo	'earmuffs' - convention to indicate dynamic vars, compiler warns if not dynamic
@	Deref: @form → (deref form)
'	Syntax-quote
foo#	'auto-gensym', consistently replaced with same auto-generated symbol everywhere inside same '(...)
~	Unquote
~@	Unquote-splicing
->	'thread first' macro ->
->>	'thread last' macro ->>
>!! <!! >! <!	core.async channel macros >!! <!! >! <!
(List literal (see Collections/Lists section)
[Vector literal (see Collections/Vectors section)
{	Map literal (see Collections/Maps section)
#'	Var-quote #'x → (var x)
#"	#"p" reads as regex pattern p (see Strings/Regex section)
{#	Set literal (see Collections/Sets section)
#(Anonymous function literal: #(...) → (fn [args] (...))
%	Anonymous function argument: %N is value of anonymous function arg N. % short for %1. %& for rest args.
##?	(1.7) Reader conditional: ##?(:clj x :cljs y) reads as x on JVM, y in ClojureScript, nothing elsewhere. Other keys: :cljr :default
##?	(1.7) Splicing reader conditional: [1 ##?(:clj [x y] :cljs [w z]) 3] reads as [1 x y 3] on JVM, [1 w z 3] in ClojureScript, [1 3] elsewhere.
#foo	tagged literal e.g. #inst #uuid
\$	JavaContainerClass\$InnerClass
foo?	conventional ending for a predicate, e.g.: zero? vector? instance? (unenforced)
foo!	conventional ending for an unsafe operation, e.g.: set! swap! alter-meta! (unenforced)
-	conventional name for an unused value (unenforced)
#_	Ignore next form

Metadata (clojure.org/reference/reader, special_forms)

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

Special Forms (clojure.org/reference/special_forms)

def if do let letfn quote var fn loop recur set! throw try monitor-enter monitor-exit	
Binding Forms / Destructuring	(examples) let fn defn defmacro loop for doseq if-let when-let if-some when-some

Vars and global environment (clojure.org/reference/vars)

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

Namespace

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

Loading

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

Concurrency

Atoms	atom swap! reset! compare-and-set! (1.9) swap-vals! reset-vals!
Futures	future future-call future-done? future-cancel future-cancelled? future?
Threads	bound-fn bound-fn* get-thread-bindings push-thread-bindings pop-thread-bindings thread-bound?
Volatiles	(1.7) volatile! vreset! vswap! volatile?
Misc	locking pcalls pvalues pmap seque promise deliver

Refs and Transactions (clojure.org/reference/refs)

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

Agents and Asynchronous Actions (clojure.org/reference/agents)

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

Java Interoperation (clojure.org/reference/java_interop)

General	.. doto Classname/ Classname. new bean comparator enumeration-seq import iterator-seq memfn set! class class? bases supers type gen-class gen-interface definterface
Cast	boolean byte short char int long float double bigdec bigint num cast biginteger
Exceptions	throw try catch finally pst ex-info ex-data (1.9) StackTraceElement->vec

Arrays

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

Proxy (Clojure type selection flowchart)

Create	proxy get-proxy-class construct-proxy init-proxy
Misc	proxy-mappings proxy-super update-proxy

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?

Other

XML	clojure.xml/parse xml-seq
REPL	*1 *2 *3 *e *print-dup* *print-length* *print-level* *print-meta* *print-readably*
Code	*compile-files* *compile-path* *file* *warn-on-reflection* compile loaded-libs test
Misc	eval force hash name *clojure-version* clojure-version *command-line-args*
Browser / Shell	(clojure.java.browse/) browse-url (clojure.java.shell/) sh with-sh-dir with-sh-env