Cheatsheet

Clojure 1.8 Cheat Sheet (v35)

Download PDF version / Source repo

Many thanks to Steve Tayon for creating it and Andy Fingerhut for ongoing maintenance.

Documentation

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

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

Create	`clojure.lang.Persis` `(no literal syntax o` `fn)`
Examine	`peek`
'Change'	`conj` `pop`

Relations (set of maps, each

	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	+ - * / quot rem mod inc dec max min +' -' *' inc' dec'
Compare	== < > <= >= compare
Bitwise	bit-and bit-or bit-xor bit-not bit-flip bit-set bit-shift-right bit-shift-left bit-and-not bit-

with same keys, aka rels)

Rel algebra	(clojure.set/) join select project union difference intersection index rename

Transients (clojure.org/reference/transient

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

Misc

Compare	= identical? not= not compare clojure.data/diff
Test	true? false? instance? nil? (1.6) some?

	`clear` `bit-test` (1.6) `unsigned-bit-shift-right` (see `BigInteger` for integers larger than Long)
Cast	`byte` `short` `int` `long` `float` `double` `bigdec` `bigint` `num` `rationalize` `biginteger`
Test	`zero?` `pos?` `neg?` `even?` `odd?` `number?` `rational?` `integer?` `ratio?` `decimal?` `float?`
Random	`rand` `rand-int`
BigDecimal	`with-precision`
Unchecked	`*unchecked-math*` `unchecked-add` `unchecked-dec` `unchecked-inc` `unchecked-multiply` `unchecked-negate` `unchecked-subtract`

Sequences

Creating a Lazy Seq

From collection	`seq` `vals` `keys` `rseq` `subseq` `rsubseq` `sequence`
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` `take-nth` `for`
Get longer	`cons` `conj` `concat` `lazy-cat` `mapcat` `cycle` `interleave` `interpose`
Tail-items	`rest` `nthrest` `next` `fnext`

Strings

Create	str format "a string" "escapes \b\f\n\t\r\" octal \377 hex \ucafe" See also section IO/to string
Use	count get subs compare (clojure.string/) join escape split split-lines replace replace-first reverse (1.8) index-of last-index-of
Regex	#"*pattern*" re-find re-seq re-matches re-pattern re-matcher re-groups (clojure.string/) replace replace-first re-quote-replacement Note: \ in #"" is not escape char. (re-pattern "\\s*\\d+") can be written #"\s*\d+"
Letters	(clojure.string/) capitalize lower-case upper-case

	nnext drop drop-while take-last for
Head-items	take 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

Trim	(clojure.string/) trim trim-newline triml trimr
Test	string? (clojure.string/) blank? (1.8) starts-with? ends-with? includes?

Other

Characters	char char? char-name-string char-escape-string literals: \a \newline (more at link)
Keywords	keyword keyword? find-keyword literals: :kw :my.ns/kw ::in-cur-ns
Symbols	symbol symbol? gensym literals: my-sym my.ns/foo
Misc	literals: true false nil

	reductions set vec into-array to-array-2d mapv filterv
Pass to fn	apply
Search	some filter
Force evaluation	doseq dorun doall (1.7) run!
Check for forced	realized?

Transducers

(clojure.org/reference/trans

Off the shelf	map mapcat filter remove take take-while take-nth dr drop-while replac partition-by partition-all kee keep-indexed map-indexed distinct interpose (1.7) c dedupe random-samp
Create your own	(1.7) completing ensure-reduced unreduced See also section

Collections

Collections

Generic ops	count empty not-empty into conj (clojure.walk/) walk prewalk prewalk-demo prewalk-replace postwalk postwalk-demo postwalk-replace
Content tests	distinct? empty? every? not-every? some not-any?
Capabilities	sequential? associative? sorted? counted? reversible?
Type tests	coll? list? vector? set? map? seq? (1.6) record? (1.8) map-entry?

Lists (conj, pop, & peek at beginning)

	Concurrency/Volat
Use	into sequence (1. transduce eductio
Early termination	reduced reduced? deref

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

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

Vectors (conj, pop, & peek at end)

Create	[] vector vec vector-of mapv filterv (clojure.core.rrb-vector/) vector vec vector-of
Examine	(my-vec idx) → (nth my-vec idx) get peek .indexOf .lastIndexOf
'Change'	assoc assoc-in pop subvec replace conj rseq update-in (1.7) update
Ops	reduce-kv

Sets

Create unsorted	#{} set hash-set
Create sorted	sorted-set sorted-se by (clojure.data.avl sorted-set sorted-se

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

	by (flatland.ordered.se ordered-set (clojure.data.int-ma int-set dense-int-se
Examine	(my-set item) → (get my-set item) contains?
'Change'	conj disj
Set ops	(clojure.set/) union difference intersection select See also section Relations
Test	(clojure.set/) subset? superset?
Sorted sets	rseq subseq rsubseq

Maps

Create unsorted	{} hash-map array-map zipmap bean frequencies group-by (clojure.set/) index
Create sorted	sorted-map sorted-map-by (clojure.data.avl/) sorted-map sorted-map-by (flatland.ordered.map/) ordered-map (clojure.data.priority-map/) priority-map (flatland.useful.map/) ordering-map

	(clojure.java.io/) f copy delete-file reso as-file as-url as-rel path GitHub: fs
Data readers	*data-readers* defau data-readers *default reader-fn*

	(clojure.data.int-map/) int-map
Examine	(my-map k) → (get my-map k) also (: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 (1.7) update (clojure.set/) rename-keys map-invert GitHub: Medley
Ops	reduce-kv
Entry	key val
Sorted maps	rseq subseq rsubseq

Functions

Create	fn defn defn- definline identity constantly memfn comp complement partial juxt memoize fnil every-pred some-fn
Call	apply -> ->>

Special Forms

(clojure.org/reference/spec

def if do let letfn quote var fn loop recur set! throw try monitor-enter monitor-exit

Binding	(examples)

	trampoline as-> cond-> cond->> some-> some->>
Test	fn? ifn?

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/datatypes

Forms / Destructuring	let fn defn defmacro loop for doseq if-let when-let (1.6) 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

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

Types (clojure.org/reference/datatypes

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

Multimethods (clojure.org/reference/multimeth

Define	`( defmulti my-mm dispatch-fn)`
Method define	`( defmethod my-mm :dispatch-value [args] ...)`
Dispatch	`get-method methods`

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	`(tutorial) require`

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` `(1.6)` `when-some` `if-some`
Loop	`for` `doseq` `dotimes` `while`
Arrange	`..` `doto` `->` `->>` `as->` `cond->` `cond->>` `some->` `some->>`

libs	`use` `import` `refer`
List loaded	`loaded-libs`
Load misc	`load` `load-file` `load-reader` `load-string`

Concurrency

Atoms	`atom` `swap!` `reset!` `compare-and-set!`
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`

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/reade tutorial)

,	Comma reads as white spac used between map key/valu readability.
'	quote: 'form → (qu
/	Namespace separator (see Primitives/Other section)
\	Character literal (see Primit section)
:	Keyword (see Primitives/Ot
;	Single line comment
^	Metadata (see Metadata se

	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	await await-for

foo	'earmuffs' - conventi indicate dynamic vars warns if not dynamic
@	Deref: @*form* → (de
`	Syntax-quote
foo#	'auto-gensym', consis replaced with same au generated symbol every inside same `(...)
~	Unquote
~@	Unquote-splicing
->	'thread first' macro
->>	'thread last' macro -
(	List literal (see Collections/L
[	Vector literal (see Collection section)
{	Map literal (see Collections/Maps section)
#'	Var-quote: #'*x* → (
#"	#"*p*" reads as regex patte Strings/Regex section)
#{	Set literal (see Collections/S
#(	Anonymous function li (...) → (fn [args] (
%	Anonymous function ar %N is value of anony function arg N . % s %1 . %& for rest arg
#?	(1.7) Reader conditic (:clj x :cljs y) reac on JVM, y in Clojure nothing elsewhere. Ot :cljr :default
#?@	(1.7) Splicing reader

waiting	
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_

General	.. doto Classname/ Classname. new bean comparator enumeration-seq import iterator-seq memfn set! class class? bases supers

	conditional: [1 #?(:c :cljs [w z]) 3] reads y 3] on JVM, [1 w z ClojureScript, [1 3] elsewhere.
#foo	tagged literal e.g. #uuid
$	JavaContainerClass$I
foo?	conventional ending f predicate, e.g.: zero instance? (unenforced
foo!	conventional ending f unsafe operation, e.g swap! alter-meta! (un
_	conventional name for unused value (unenfor
#_	Ignore next form

Metadata (clojure.org/reference/reade special_forms)

General	^{:key1 val1 :key2 val2 ...}
Abbrevs	^Type → ^{:tag Type} ^:key → ^{:key true}

	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

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

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

	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

Other

XML	clojure.xml/parse xm seq
REPL	*1 *2 *3 *e *print-dup* *print-length* *print-level* *print meta* *print-readabl
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