Clojure Cheat Sheet (Clojure 1.6 - 1.9, sheet v44)

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	`+ - * / 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-clear bit-test 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? (1.9) double? int? nat-int? neg-int? pos-int?`
Random	`rand rand-int`
BigDecimal	`with-precision`
Unchecked	`*unchecked-math* unchecked-add unchecked-dec unchecked-inc unchecked-multiply unchecked-negate unchecked-subtract`

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`
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.name.space/kw ::in-cur-namespace ::namespace-alias/kw`
Symbols	`symbol symbol? gensym literals: my-sym my.ns/foo`
Misc	`literals: true false nil`

Collections

Collections

Generic ops	`count empty not-empty into conj (clojure.walk/) walk prewalk prewalk-demo prewalk-replace postwalk postwalk-demo postwalk-replace (1.9) bounded-count`
Content tests	`distinct? empty? every? not-every? some not-any?`
Capabilities	`sequential? associative? sorted? counted? reversible?`
Type tests	`coll? list? vector? set? map? seq? record? (1.8) map-entry?`

Lists (conj, pop, & peek at beginning)

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-set-by (clojure.data.avl/) sorted-set sorted-set-by (flatland.ordered.set/) ordered-set (clojure.data.int-map/) int-set dense-int-set`
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.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`

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

Create	`clojure.lang.PersistentQueue/EMPTY (no literal syntax or constructor fn)`
Examine	`peek`
'Change'	`conj pop`

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

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

Transients (clojure.org/reference/transients)

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? some?`

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 (1.7) dedupe random-sample`
Get longer	`cons conj concat lazy-cat mapcat cycle interleave interpose`
Tail-items	`rest nthrest next fnext 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 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/transducers)

Off the shelf	`map mapcat filter remove take take-while take-nth drop drop-while replace partition-by partition-all keep keep-indexed map-indexed distinct interpose (1.7) cat dedupe random-sample (1.9) halt-when`
Create your own	`(1.7) completing ensure-reduced unreduced` See also section Concurrency/Volatiles
Use	`into sequence (1.7) transduce eduction`
Early termination	`reduced reduced? deref`

Spec (rationale, guide)

Operations	`valid? conform unform explain explain-data explain-str explain-out form describe assert check-asserts check-asserts?`
Generator ops	`gen exercise exercise-fn`
Defn. & registry	`def fdef registry get-spec spec? spec with-gen`
Logical	`and or`
Collection	`coll-of map-of every every-kv keys merge`
Regex	`cat alt * + ? & keys*`
Range	`int-in inst-in double-in int-in-range? inst-in-range?`
Other	`nilable multi-spec fspec conformer`
Custom explain	`explain-printer *explain-out*`

Predicates with test.check generators

Numbers	`number? rational? integer? ratio? decimal? float? zero? (1.9) double? int? nat-int? neg-int? pos-int?`
Symbols, keywords	`keyword? symbol? (1.9) ident? qualified-ident? qualified-keyword? qualified-symbol? simple-ident? simple-keyword? simple-symbol?`
Other scalars	`string? true? false? nil? some? (1.9) boolean? bytes? inst? uri? uuid?`
Collections	`list? map? set? vector? associative? coll? sequential? seq? empty? (1.9) indexed? seqable?`
Other	`(1.9) any?`

IO

to/from ...	`spit slurp` (to writer/from reader, Socket, string with file name, URI, etc.)
to *out*	`pr prn print printf println newline` (clojure.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` (clojure.tools.reader.edn/) `read`
from reader	`line-seq` (clojure.tools.reader.edn/) `read` also: `(binding [*in* reader] ...) java.io.Reader`
from string	`with-in-str` (clojure.tools.reader.edn/) `read-string`
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 delete-file resource as-file as-url as-relative-path` GitHub: `fs`
Data readers	`*data-readers* default-data-readers *default-data-reader-fn*`

Functions

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

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`
	`( 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`