Clojure Cheat Sheet (Clojure 1.2)

Documentation

	doc find-doc
clojure.repl	apropos source

Primitives

Numbers

Arithmetic	+ - * / quot rem mod inc dec max min
Compare	= == not= < > <= >=
Bitwise	bit-{and, or, xor, not, flip, set, shift-right, shift-left, and-not, clear, test}
Cast	byte short int long float double bigdec bigint num rationalize
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}

Strings

Create	str print-str println-str pr-str prn-str with-out-str
Use	count get subs format
Cast/Test	char char? string?

Strings (clojure.string)

Test	blank?
Letters	capitalize lower-case upper-case
Use	join escape split split-lines replace replace-first reverse
Trim	trim trim-newline triml trimr

Other

Characters	char char-name-string char-escape-string
Keywords	keyword 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? seq? vector? list? map? set?

Lists

Create	'() list list*
Stack	peek pop
Examine	first rest peek list?
'Change'	cons conj

Vectors

Create	[] vector vec vector-of
Examine	get nth peek rseq vector?
'Change'	assoc pop subvec replace conj

Sets

Create	#{} hash-set sorted-set sorted-set-by set conj disj
Examine	get

Sets (clojure.set)

Rel. algebra	join select project union difference intersection
Get map	index rename-keys rename map-invert
Test	subset? superset?

Maps

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

StructMaps

Create	defstruct create-struct accessor
Individual	struct-map struct
Use	get assoc

Transients

Create	transient persistent!
Change	conj! pop! assoc! dissoc! disj! Remember to bind result to a symbol!

Misc

Compare	= == identical? not= not
Test	true? false? nil?

Sequences

Creating a Lazy Seq

From collection	seq vals keys rseq subseq rsubseq
From producer fn	lazy-seq repeatedly iterate
From constant	repeat replicate 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 fnext nnext drop drop-while for
Head-items	take take-nth take-while take-last butlast drop-last for
'Change'	conj concat distinct group-by partition partition-all partition-by split-at split-with filter remove replace shuffle
Rearrange	reverse sort sort-by
Process each item	map pmap map-indexed mapcat for replace seque
Un-lazy Seq	sequence

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

Zippers (clojure.zip)

Create	zipper
Get zipper	seq-zip vector-zip xml-zip
Get location	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?

Printing

Print to *out*	pr prn print printf println newline
Print to string	pr-str prn-str print-str println-str with-out-str

Functions

Create	fn defn defn- definline identity constantly memfn comp complement partial juxt memoize
Call	-> -» apply
Test	fn? ifn?

Multimethods

Create	`defmulti defmethod`
Dispatch	`get-method methods`
Remove	`remove-method remove-all-methods`
Prefer	`prefer-method prefers`
Relation	`derive isa? parents ancestors descendants make-hierarchy`

Macros

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

Reader Macros

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

Vars and global environment

Def variants	`defn defn- definline defmacro defmethod defmulti defonce defstruct`
Interned vars	`declare intern binding find-var var`
Var objects	`with-local-vars var-get var-set alter-var-root var?`
Var validators	`set-validator! get-validator`
Var metadata	`doc find-doc test`

Namespace

Current	`*ns*`
Create/Switch	`in-ns 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`
Remove	`ns-unalias ns-unmap remove-ns`

Loading

Loading libs	`require use import refer`
Listing loaded libs	`loaded-libs`
Loading misc	`load load-file load-reader load-string`

Special Forms

```
def if do let quote var fn loop recur throw try
monitor-enter monitor-exit
```

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?`
Misc	`locking pcalls pvalues pmap seque promise deliver`

Refs and Transactions

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

Agents and Asynchronous Actions

Create	`agent`
Examine	`agent-error`
Change state	`send send-off restart-agent`
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

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

Arrays

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

Proxy

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

Other

Regex	`#"pattern" re-pattern re-matcher re-find re-matches re-groups re-seq`
XML	`clojure.xml/parse xml-seq`
REPL	`*1 *2 *3 *e *print-dup* *print-length* *print-level* *print-meta* *print-readably*`
IO	`*in* *out* *err* flush read-line read read-string slurp spit with-in-str with-out-str with-open`
Code	`*compile-files* *compile-path* *file* *warn-on-reflection* compile gen-class gen-interface loaded-libs test`
Misc	`eval force hash name *clojure-version* clojure-version *command-line-args*`