The Clojure Programming Language: Quick Reference Sheet

Command Line	
clojure file.clj	Run file in clojure
clojure -r file.clj	Run file with REPL
lein new	Create new project
lein repl	Run repl in project
lein jar	Create jar of project

General Concepts	
Iteration	Closure uses loop and recur for tail recursion.
Threading	Threading is done with the external core.async library.
Reducers	Reducing functions are in the core.reducers library.
Dependencies	External libraries are used through the leiningen project manager.
Laziness	Lazy evaluation can be invoked with `cycle` and `take` or `lazy-seq`.

Control Flow	
`(if <boolexp> <trueval> <elseval>)`	If Statements
`(cond (<boolexp>) <result> :else <result>)`	Cond block
`(when <boolexp> <thenval>)`	If without else
`(do <exp1> [exp...])`	Do these statements in order

List Operations	
`(list arg1 arg2 ...)`	Create a list
`(first <list>)`	Get first element of a list
`(last <list>)`	Get last element of a list
`(rest <list>)`	Get cdr of list
`(cons <item> <list>)`	Join car and cdr of a list
`(nth <list> <index>)`	Get the element at index

Functions	
`(<name> [arg1 ...])`	Call a function
`(defn <name> ([args] <body>) [([args] <body>)] ...)`	Define a function
`(fn [args] <body>)`	Anonymous function
`#(<body>)`	Shorter Anonymous function
`(defn <name> [[<args>]] <body> )`	Parameter destructuring
`(def <name> (partial <fname> [arg1 arg2 ...]))`	Partial function application

Assignment	
`(let [<name> <value> <name> <value> ...] body)`	Temporary assignment
`(letfn [(<name> <func>)] <body>)`	A let for recursion

Types	
`(class <value>)`	Returns the class of value
`(defprotocol <name> (<fname> [sig]))`	Defines and interface with signature
`(defrecord <name> [values] <impl> (<methodimpl>))`	Defines an object with implementation
`(<class>. args)`	Creates object with given values
`(.<method> <object>)`	Call a method on an object

Complex Data Structures	
`#{1 2 3}`	Creates a set
`(sort #{1 3 2})`	Sorts a given set
`(sorted-set 2 3 1)`	Creates a sorted set
`{:key "value", :a "b"}`	Creates a map
`(:key myMap)`	Gets a value from map with given key
`(sorted-map 1 :one, 3 :three, 2 :two)`	Creates a sorted map

Thread Macros	
`->`	Thread first macro
`-»`	Thread last macro
`as->`	Thread as macro
`some->` and `some-»`	Same as -> and -» but short circuit if value is ever nil
`cond->` and `cond-»`	Same as -> and -» but only does action if condition is true