Clojure Cheatsheet

A concise reference for Clojure syntax, data structures, functions, and macros, designed to help you quickly recall key elements of the language.

Core Data Structures

Basic Data Types

`nil`	Represents null or the absence of a value.
`boolean`	`true` or `false`
`number`	Integers, floats, ratios. Example: `1`, `1.0`, `1/2`
`string`	Immutable sequence of characters. Example: `"Hello, Clojure!"`
`keyword`	Interned strings, used as keys in maps. Example: `:name`
`symbol`	Represents variables or function names. Example: `my-variable`

Collections

`list`	Ordered collection. Created with `'(1 2 3)`. Implemented as a singly linked list.
`vector`	Indexed collection. Created with `[1 2 3]`. Supports efficient random access.
`map`	Key-value pairs. Created with `{ :a 1, :b 2 }`. Keys and values can be any type.
`set`	Collection of unique values. Created with `#{ 1 2 3 }`.
`queue`	A sequence supporting FIFO semantics. Created with `clojure.lang.PersistentQueue/EMPTY` and `conj` and `pop`.

Atoms

Atoms provide a mutable reference to an immutable value.

```
(def my-atom (atom 0))
(swap! my-atom inc) ; Increment the
value
@my-atom ; Dereference to get the
current value
```

Functions and Macros

Function Definition

Functions are defined using `defn`.

```
(defn my-function [arg1 arg2]
  (+ arg1 arg2))
```

Anonymous functions can be created with `fn` or the reader macro `#()`.

```
(fn [x] (* x x))
#(* % %)
```

Basic Functions

`(+ x y)`	Addition
`(- x y)`	Subtraction
`(* x y)`	Multiplication
`(quot x y)`	Integer division
`(rem x y)`	Remainder
`(inc x)`	Increment
`(dec x)`	Decrement

Macros

Macros are code transformations performed at compile time. Defined with `defmacro`.

```
(defmacro my-macro [arg]
  `(println ~arg))

(my-macro "Hello") ; expands to (println
"Hello")
```

Control Flow

Conditionals

`if`	`(if condition then else)`
`when`	`(when condition & body)` - executes body if condition is true.
`when-not`	`(when-not condition & body)` - executes body if condition is false.
`cond`	`(cond condition1 expr1 condition2 expr2 ...)` - multi-branch conditional.
`case`	`(case expr clause1 expr1 clause2 expr2 ...)` - conditional based on the value of an expression.

Looping and Iteration

`loop`	`(loop [bindings...] & body)` - defines a recursive loop with initial bindings.
`recur`	`(recur exprs...)` - jumps back to the beginning of the innermost loop with updated bindings.
`doseq`	`(doseq [seq-exprs...] & body)` - iterates over a sequence, executing the body for each element (side effects only).
`dotimes`	`(dotimes [i n] & body)` - executes the body `n` times, with `i` bound to the current iteration number.
`for`	`(for [seq-exprs...] & body)` - list comprehension, returns a lazy sequence of the results of evaluating body for each element.

Exception Handling

`try` / `catch` / `finally`

```
(try
  (/ 1 0)
  (catch ArithmeticException e
    (println "Caught exception:",
(.getMessage e)))
  (finally
    (println "Finally block executed")))
```

Sequences and Collections

Sequence Operations

`map`	`(map f coll)` - Applies function `f` to each element in `coll`, returning a new sequence.
`filter`	`(filter pred coll)` - Returns a new sequence containing only the elements of `coll` for which `(pred element)` is true.
`reduce`	`(reduce f val coll)` - Reduces `coll` using function `f`, starting with initial value `val`.
`take`	`(take n coll)` - Returns a new sequence containing the first `n` elements of `coll`.
`drop`	`(drop n coll)` - Returns a new sequence without the first `n` elements of `coll`.
`first`	`(first coll)` - Returns the first element of `coll`.
`rest`	`(rest coll)` - Returns a sequence without the first element of `coll`.
`cons`	`(cons x coll)` - Adds `x` to the beginning of `coll`.

Collection Specific Functions

`get`	`(get map key)` - Returns the value associated with `key` in `map`.
`assoc`	`(assoc map key val)` - Returns a new map with `key` associated with `val`.
`dissoc`	`(dissoc map key)` - Returns a new map without `key`.
`conj`	`(conj coll val)` - Adds `val` to the collection. Behavior depends on collection type.
`count`	`(count coll)` - Returns the number of elements in `coll`.