Clojure

Clojure is a Lisp family language developed for the Java Virtual Machine. It has a much stronger emphasis on pure functional programming than Common Lisp, but includes several STM utilities to handle state as it comes up.

This combination allows it to handle concurrent processing very simply, and often automatically.

(You need a version of Clojure 1.2 or newer)

```
; Comments start with semicolons.

; Clojure is written in "forms", which are just
; lists of things inside parentheses, separated by whitespace.
;
; The clojure reader assumes that the first thing is a
; function or macro to call, and the rest are arguments.

; The first call in a file should be ns, to set the namespace
(ns learnclojure)

; More basic examples:

; str will create a string out of all its arguments
(str "Hello" " " "World") ; => "Hello World"

; Math is straightforward
(+ 1 1) ; => 2
(- 2 1) ; => 1
(* 1 2) ; => 2
(/ 2 1) ; => 2

; Equality is =
(= 1 1) ; => true
(= 2 1) ; => false

; You need not for logic, too
(not true) ; => false

; Nesting forms works as you expect
(+ 1 (- 3 2)) ; = 1 + (3 - 2) => 2

; Types
;;;;;;;;;;;;;

; Clojure uses Java's object types for booleans, strings and numbers.
; Use `class` to inspect them.
(class 1) ; Integer literals are java.lang.Long by default
(class 1.); Float literals are java.lang.Double
(class ""); Strings always double-quoted, and are java.lang.String
(class false) ; Booleans are java.lang.Boolean
(class nil); The "null" value is called nil

; If you want to create a literal list of data, use ' to stop it from
; being evaluated
```

```
'(+ 1 2) ; => (+ 1 2)
; (shorthand for (quote (+ 1 2)))

; You can eval a quoted list
(eval '(+ 1 2)) ; => 3

; Collections & Sequences
;;;;;;;;;;;;;;;;;;

; Lists are linked-list data structures, while Vectors are array-backed.
; Vectors and Lists are java classes too!
(class [1 2 3]); => clojure.lang.PersistentVector
(class '(1 2 3)); => clojure.lang.PersistentList

; A list would be written as just (1 2 3), but we have to quote
; it to stop the reader thinking it's a function.
; Also, (list 1 2 3) is the same as '(1 2 3)

; "Collections" are just groups of data
; Both lists and vectors are collections:
(coll? '(1 2 3)) ; => true
(coll? [1 2 3]) ; => true

; "Sequences" (seqs) are abstract descriptions of lists of data.
; Only lists are seqs.
(seq? '(1 2 3)) ; => true
(seq? [1 2 3]) ; => false

; A seq need only provide an entry when it is accessed.
; So, seqs which can be lazy -- they can define infinite series:
(range 4) ; => (0 1 2 3)
(range) ; => (0 1 2 3 4 ...) (an infinite series)
(take 4 (range)) ;  (0 1 2 3)

; Use cons to add an item to the beginning of a list or vector
(cons 4 [1 2 3]) ; => (4 1 2 3)
(cons 4 '(1 2 3)) ; => (4 1 2 3)

; Conj will add an item to a collection in the most efficient way.
; For lists, they insert at the beginning. For vectors, they insert at the end.
(conj [1 2 3] 4) ; => [1 2 3 4]
(conj '(1 2 3) 4) ; => (4 1 2 3)

; Use concat to add lists or vectors together
(concat [1 2] '(3 4)) ; => (1 2 3 4)

; Use filter, map to interact with collections
(map inc [1 2 3]) ; => (2 3 4)
(filter even? [1 2 3]) ; => (2)

; Use reduce to reduce them
(reduce + [1 2 3 4])
; = (+ (+ (+ 1 2) 3) 4)
; => 10
```

```clojure
; Reduce can take an initial-value argument too
(reduce conj [] '(3 2 1))
; = (conj (conj (conj [] 3) 2) 1)
; => [3 2 1]

; Functions
;;;;;;;;;;;;;;;;;;;

; Use fn to create new functions. A function always returns
; its last statement.
(fn [] "Hello World") ; => fn

; (You need extra parens to call it)
((fn [] "Hello World")) ; => "Hello World"

; You can create a var using def
(def x 1)
x ; => 1

; Assign a function to a var
(def hello-world (fn [] "Hello World"))
(hello-world) ; => "Hello World"

; You can shorten this process by using defn
(defn hello-world [] "Hello World")

; The [] is the list of arguments for the function.
(defn hello [name]
  (str "Hello " name))
(hello "Steve") ; => "Hello Steve"

; You can also use this shorthand to create functions:
(def hello2 #(str "Hello " %1))
(hello2 "Fanny") ; => "Hello Fanny"

; You can have multi-variadic functions, too
(defn hello3
  ([] "Hello World")
  ([name] (str "Hello " name)))
(hello3 "Jake") ; => "Hello Jake"
(hello3) ; => "Hello World"

; Functions can pack extra arguments up in a seq for you
(defn count-args [& args]
  (str "You passed " (count args) " args: " args))
(count-args 1 2 3) ; => "You passed 3 args: (1 2 3)"

; You can mix regular and packed arguments
(defn hello-count [name & args]
  (str "Hello " name ", you passed " (count args) " extra args"))
(hello-count "Finn" 1 2 3)
; => "Hello Finn, you passed 3 extra args"
```

```
; Maps
;;;;;;;;;;

; Hash maps and array maps share an interface. Hash maps have faster lookups
; but don't retain key order.
(class {:a 1 :b 2 :c 3}) ; => clojure.lang.PersistentArrayMap
(class (hash-map :a 1 :b 2 :c 3)) ; => clojure.lang.PersistentHashMap

; Arraymaps will automatically become hashmaps through most operations
; if they get big enough, so you don't need to worry.

; Maps can use any hashable type as a key, but usually keywords are best
; Keywords are like strings with some efficiency bonuses
(class :a) ; => clojure.lang.Keyword

(def stringmap {"a" 1, "b" 2, "c" 3})
stringmap  ; => {"a" 1, "b" 2, "c" 3}

(def keymap {:a 1, :b 2, :c 3})
keymap ; => {:a 1, :c 3, :b 2}

; By the way, commas are always treated as whitespace and do nothing.

; Retrieve a value from a map by calling it as a function
(stringmap "a") ; => 1
(keymap :a) ; => 1

; Keywords can be used to retrieve their value from a map, too!
(:b keymap) ; => 2

; Don't try this with strings.
;("a" stringmap)
; => Exception: java.lang.String cannot be cast to clojure.lang.IFn

; Retrieving a non-present key returns nil
(stringmap "d") ; => nil

; Use assoc to add new keys to hash-maps
(def newkeymap (assoc keymap :d 4))
newkeymap ; => {:a 1, :b 2, :c 3, :d 4}

; But remember, clojure types are immutable!
keymap ; => {:a 1, :b 2, :c 3}

; Use dissoc to remove keys
(dissoc keymap :a :b) ; => {:c 3}

; Sets
;;;;;;

(class #{1 2 3}) ; => clojure.lang.PersistentHashSet
(set [1 2 3 1 2 3 3 2 1 3 2 1]) ; => #{1 2 3}
```

```
; Add a member with conj
(conj #{1 2 3} 4) ; => #{1 2 3 4}

; Remove one with disj
(disj #{1 2 3} 1) ; => #{2 3}

; Test for existence by using the set as a function:
(#{1 2 3} 1) ; => 1
(#{1 2 3} 4) ; => nil

; There are more functions in the clojure.sets namespace.

; Useful forms
;;;;;;;;;;;;;;;;;

; Logic constructs in clojure are just macros, and look like
; everything else
(if false "a" "b") ; => "b"
(if false "a") ; => nil

; Use let to create temporary bindings
(let [a 1 b 2]
  (> a b)) ; => false

; Group statements together with do
(do
  (print "Hello")
  "World") ; => "World" (prints "Hello")

; Functions have an implicit do
(defn print-and-say-hello [name]
  (print "Saying hello to " name)
  (str "Hello " name))
(print-and-say-hello "Jeff") ;=> "Hello Jeff" (prints "Saying hello to Jeff")

; So does let
(let [name "Urkel"]
  (print "Saying hello to " name)
  (str "Hello " name)) ; => "Hello Urkel" (prints "Saying hello to Urkel")


; Use the threading macros (-> and ->>) to express transformations of
; data more clearly.

; The "Thread-first" macro (->) inserts into each form the result of
; the previous, as the first argument (second item)
(->
   {:a 1 :b 2}
   (assoc :c 3) ;=> (assoc {:a 1 :b 2} :c 3)
   (dissoc :b)) ;=> (dissoc (assoc {:a 1 :b 2} :c 3) :b)

; This expression could be written as:
; (dissoc (assoc {:a 1 :b 2} :c 3) :b)
; and evaluates to {:a 1 :c 3}
```

```
; The double arrow does the same thing, but inserts the result of
; each line at the *end* of the form. This is useful for collection
; operations in particular:
(->>
   (range 10)
   (map inc)      ;=> (map inc (range 10)
   (filter odd?) ;=> (filter odd? (map inc (range 10))
   (into []))    ;=> (into [] (filter odd? (map inc (range 10)))
                 ; Result: [1 3 5 7 9]

; Modules
;;;;;;;;;;;;;;;

; Use "use" to get all functions from the module
(use 'clojure.set)

; Now we can use set operations
(intersection #{1 2 3} #{2 3 4}) ; => #{2 3}
(difference #{1 2 3} #{2 3 4}) ; => #{1}

; You can choose a subset of functions to import, too
(use '[clojure.set :only [intersection]])

; Use require to import a module
(require 'clojure.string)

; Use / to call functions from a module
; Here, the module is clojure.string and the function is blank?
(clojure.string/blank? "") ; => true

; You can give a module a shorter name on import
(require '[clojure.string :as str])
(str/replace "This is a test." #"[a-o]" str/upper-case) ; => "THIs Is A tEst."
; (#"" denotes a regular expression literal)

; You can use require (and use, but don't) from a namespace using :require.
; You don't need to quote your modules if you do it this way.
(ns test
  (:require
    [clojure.string :as str]
    [clojure.set :as set]))

; Java
;;;;;;;;;;;;;;;;;

; Java has a huge and useful standard library, so
; you'll want to learn how to get at it.

; Use import to load a java module
(import java.util.Date)

; You can import from an ns too.
(ns test
```

```
  (:import java.util.Date
           java.util.Calendar))

; Use the class name with a "." at the end to make a new instance
(Date.) ; <a date object>

; Use . to call methods. Or, use the ".method" shortcut
(. (Date.) getTime) ; <a timestamp>
(.getTime (Date.)) ; exactly the same thing.

; Use / to call static methods
(System/currentTimeMillis) ; <a timestamp> (system is always present)

; Use doto to make dealing with (mutable) classes more tolerable
(import java.util.Calendar)
(doto (Calendar/getInstance)
  (.set 2000 1 1 0 0 0)
  .getTime) ; => A Date. set to 2000-01-01 00:00:00

; STM
;;;;;;;;;;;;;;;;

; Software Transactional Memory is the mechanism clojure uses to handle
; persistent state. There are a few constructs in clojure that use this.

; An atom is the simplest. Pass it an initial value
(def my-atom (atom {}))

; Update an atom with swap!.
; swap! takes a function and calls it with the current value of the atom
; as the first argument, and any trailing arguments as the second
(swap! my-atom assoc :a 1) ; Sets my-atom to the result of (assoc {} :a 1)
(swap! my-atom assoc :b 2) ; Sets my-atom to the result of (assoc {:a 1} :b 2)

; Use '@' to dereference the atom and get the value
my-atom  ;=> Atom<#...> (Returns the Atom object)
@my-atom ; => {:a 1 :b 2}

; Here's a simple counter using an atom
(def counter (atom 0))
(defn inc-counter []
  (swap! counter inc))

(inc-counter)
(inc-counter)
(inc-counter)
(inc-counter)
(inc-counter)

@counter ; => 5

; Other STM constructs are refs and agents.
; Refs: http://clojure.org/refs
; Agents: http://clojure.org/agents
```

Further Reading

This is far from exhaustive, but hopefully it's enough to get you on your feet.

Clojure.org has lots of articles: http://clojure.org/

Clojuredocs.org has documentation with examples for most core functions: http://clojuredocs.org/quickref/Clojure%20Core

4Clojure is a great way to build your clojure/FP skills: http://www.4clojure.com/

Clojure-doc.org (yes, really) has a number of getting started articles: http://clojure-doc.org/