Clojure

https://clojure.org/guides/getting_started https://clojure.org/guides/learn/syntax

Language, Clojure, JVM

환경 구축

Console 기반

https://clojure.org/guides/getting_started

Leiningen

- https://leiningen.org/
  - Mac : http://www.hildeberto.com/2015/07/installing-leiningen-on-mac-os.html
    - 사전에 brew 설치 필요

Build 및 의존성 관리 도구. npm과 같은 역할. (Windows의 경우 자동 Installer도 제공.)

```
# Project 생성
lein new app project-name

# Project 실행
## 해당 Project Directory 진입 후에
lein run

# 프로젝트 배포 (실행가능한 jar 파일 만들기)
## 해당 Project Directory 진입 후에
lein uberjar
java -jar ..\target\uberjar\~~-standalone.jar
```

Reads Evaluates Prints Loops

코드 테스트 도구

```
lein repl
```

Luminus

- http://www.luminusweb.net/

MVC Framework

```
#Leiningen으로 쉽게 프로젝트 구성!
lein new luminus my-app
cd my-app
lein run
```

IntelliJ

https://cursive-ide.com/ http://manmyung.github.io/2015/03/17/post.html http://archive.is/CseCi

- 하단 오른쪽에 “Strucural Off”를 눌러야 괄호 편집이 쉬워진다.

단축키 설정

- **Load file in REPL** 에 Alt+L
  - 현재 활성화 된 clj를 repl과 연결시킨다.
- **Send form before caret to REPL** 에 Alt+J
  - 커서 위치 바로 전의 함수를 평가한다.
- **Send top form in REPL** 에 Alt+K
  - 현재 커서를 포함하고 있는 함수를 평가한다.
  - Drag로 선택중인 함수를 평가한다.

기초 용어

- Form (형식) : 유효한 코드
- Expression (식) : Clojure의 Form
- **Evaluate (평가) : Clojure의 Form을 계산**
- var : def/defn으로 생성된 Immutable value
- Lexical Scope : 함수 몸체 안에서만 바인딩이 유효한 범위
- Higher-order function(고차 함수) : 함수를 인자로 취하거나 함수를 반환하는 함수
- first-class function(1급 함수) : 고차 함수를 1급 객체(언어 내부에서 값으로 표현되고 전달될 수 있는 자료형)[(https://namu.wiki/w/%EA%B3%A0%EC%B0%A8%20%ED%95%A8%EC%88%98)]으로 취급하는 표현

문법

관용적 표현

1. 단어간의 구분은 “-”
2. 함수 뒤에 붙는 기호의 의미
    1. ? : 함수의 반환값이 Boolean
    2. ! : 함수가 상태를 바꾼다는 의미

3. !! : 대기 호출

자료 구조

https://clojure.org/reference/data_structures

- 모든 자료구조는 Immutable!! (ex. indexing을 이용하여 값을 변경 불가)

Symbol

예약어와 비슷한 개념. Java나 Clojure에서 제공하는 패키지/클래스/메서드/함수명들.

Keyword

:(콜론)으로 표기. Keyword를 평가하면 그 자신을 반환하는 특징때문에 struct(구조체)나 Map의 Key로 사용하기에 좋다.

```
(def person {:name "Luke" :age 33})
person
=> {:name "Luke", :age 33}
```

Collections

- Immutable!
  - 원소를 수정하는 함수를 사용하면 새로운 Collections 반환

Lists

- '()
- Linked List
  - 맨 앞부터, 순차적으로 접근할 때 사용하기 적합.
  - 임의 index 참조 불가
  - get이 아닌 nth 함수로 원소를 가져온다. 이 때, 순차탐색을 하기 때문에 get보다 느리다!
- conj를 사용하면 **"앞"**에 원소가 추가된다!

```
;생성. 이 때, 어떤 요소든 추가 가능.
'(1 "a" :key +)
=> (1 "a" :key +)

(list 1 2 3 4)
=> (1 2 3 4)
```

```
;생성 및 원소 추가
(cons 1 '())
=> (1)


(first '(1 "a" :key +))
=> 1

;첫 번째 요소 제외 나머지 요소 평가
(rest '(1 "a" :key +))
=> ("a" :key +)

(last '(1 "a" :key +))
=> +


(nth '(1 2 3 4) 3)
=> 4

(conj '(1 2 3 4) 999)
=> (999 1 2 3 4)
```

Vectors

- []
- Array/Array List와 비슷한, 0-indexed collection.
    - 임의 index 접근 가능.

```

; 생성
[1 "a" :key]
=> [1 "a" :key]

(cons 1 [])
=> (1)

(vector 1.0 1 "hello")
=> [1.0 1 "hello"]


(get [3 2 1] 0)
=> 3

(get ["hello" {:key "value"} 1.0] 1)
=> {:key "value"}


;원소 추가
(conj [1 2 3] 4)
```

```
=> [1 2 3 4]
```

Map

- {}
- hash-map, Ordered Map 두 가지.

```
{:name "Luke" :age 30}
(hash-map :name "Luke" :age 30)


(get {:a 0 :b {:c "test"}} :b)
=> {:c "test"}

(get-in {:a 0 :b {:c "test"}} [:b :c])
=> "test"


; key가 없는 경우 default 반환. 만약 default 없으면 nil.
; 아래 식은 모두 같다.
(get {:a 0 :b 1} :c "This is default!")
({:a 0 :b 1} :c "This is default!")
(:c {:a 0 :b 1} "This is default!")
=> "This is default!"


;key만 반환
(keys {:key1 111 :key2 "hello"})
=> (:key1 :key2)

;값만 반환
(vals {:key1 111 :key2 "hello"})
=> (111 "hello")



;추가/수정
(assoc {:key1 111 :key2 "hello"} :key3 12.3)
=> {:key1 111, :key2 "hello", :key3 12.3}

(assoc {:key1 111 :key2 "hello"} :key2 "world")
=> {:key1 111, :key2 "world"}

;삭제
(dissoc {:key1 111 :key2 "hello"} :key2)
=> {:key1 111}

;병합
```

```
(merge {:key1 111 :key2 "hello"}
       {:key1 777 :keyX 123456})
=> {:key1 777, :key2 "hello", :keyX 123456}
```

Sets

- #{}

```
;생성
#{1 2 3}

(hash-set 1 1 2 2)
=> #{1 2}

(set [1 1 2 2])
=> #{1 2}


;추가
(conj #{:a :b} :b)
=> #{:b :a}

;제거
(disj #{:a :b} :b)
=> #{:a}


;원소 포함 여부
(contains? #{:a :b} 1)
=> false

;원소 가져오기
;아래 식은 모두 같다
(get #{:a :b} :a)
(#{:a :b} :a)
(:a #{:a :b})
=> :a
```

흐름 제어

- nil, false는 false. 그외 모든 값은 true.
- not

```
(not (= :hello :hello))
;축약 표현
```

```
(not= :hello :hello)

=> false
```

논리식

```

```

and/or

and

- and 조건에 만족하면(모두 참)? 마지막값 반환
- and 조건에 만족하지 못하면? 처음 나온 False 값 반환

```
;조건 만족
(and "hello" 111 :aaa)
=> :aaa

;조건 불만족
(and "hello" nil 123)
=> nil
```

or

- or 조건에 만족하면(하나라도 참)? 처음 나온 True 값 반환
- or 조건에 만족하지 못하면? 마지막 값 반환

```
;조건 만족
(or false "hello" 111)
=> "hello"

;조건 불만족
(or false nil)
=> nil

(or false)
=> false
```

Collections에 사용되는 논리 검사

```
;;Collections가 비어있는 지 확인

(empty? [])
=> true

;(not (empty? x))보다 관용적.
(seq [])
=> nil


;;모든 요소에 대한 조건 검사

;every : 모든 요소에 대해 검사 결과가 True?
(every? even? [2 4 6])
=> true

(every? (fn [x] (= x :apple)) [:apple :banana])
(every? #(= % :apple) [:apple :banana])
=> false

;not-any : 모든 요소에 대해 검사 결과가 False?
(not-any? #(= % :apple) [:banana :kiwi])
=> true

;some : 요소 중 일부라도 True?
;진위 함수가 평가한 값이 반환되며, 이 때 진위 함수 값이 여러개이면 첫 번째로 True인 값이 반환된다.
;진리 함수가 논리적으로 false이면 nil(논리적 false).

(some #{0} [1 2 3 4 5])
=> nil
(some #{3 5} [1 2 3 4 5])
=> 3

(some #{nil} [1 2 3 4 5])
=> nil
(some #{false} [1 2 3 4 5])
=> nil
```

제어문

if

```
(if true "TRUE!" "FALSE!")
=>"TRUE!"

(if false "TRUE!")
=> nil
;false에 해당하는 값이 존재하지 않으므로
```

if-let

```
(let [result (> 1 3)]
  (if result "TRUE" "FALSE"))

(if-let [result (> 1 3)] "TRUE" "FALSE")
```

do

if문에서 **여러** Form을 괄호로 묶어 실행할 때 사용.

```
(if true
  (do (println "TRUE!")
      "TRUE return value!")
  (do (println "FALSE!")
      "FALSE reutrn value!"))

;결과
TRUE!
=> "TRUE return value!"
```

when

True이면 본문(함수 내용)을 평가하고, **False이면 nil 반환**

- 언제 쓰는가?
  - if처럼 참/거짓 두 경우 모두 처리할 필요 없을 때
  - 검사가 참일 때만 처리할 때

```
(when true (println "TRUE!")
           "TRUE return value")

;결과
TRUE!
```

```
=> "TRUE return value"


(defn test-when [arg]
  (when arg "return value"))

(test-when 0)
(#(when % "return value") 0)
=> "return value"

(test-when false)
=> nil
```

when-let

```
(let [result (> 3 1)]
  (when result "TRUE"))
=> "TRUE"

(when-let [result (> 3 1)] "TRUE")
=> "TRUE"
```

cond

- switch문과 비슷.
- 조건식을 여러 개 쓰고 싶을 때 사용.
- 단, 나열 순서가 중요하다. **맨 처음 만족하는 조건 평가하고 종료됨**.
- default절은 :else …이지만 나열 순서상 맨 마지막에 위치하면서 저 키워드 자체가 "참"으로 평가되기 때문에 "nil, false" 제외한 어떤 값이 와도 상관없다. 그러나 관용적으로나 가독성으로 :else가 낫다.

```
(let [fruit "apple"]
  (cond
    (= fruit "banana") "yellow"
    (= fruit "kiwi") "brown"
    (= fruit "melon") "green"
    :else "nothing"))
=> "nothing"

;아래 조건식이 모두 만족하지만 첫 번째 조건 평가하고 종료.
(let [x 10]
  (cond
    (> x 1) "first"
    (> x 2) "second"
```

```
    (> x 3) "third"))
=> "first"

;default절의 원리. 만약, 기술안하면 nil.
(let [x 10]
  (cond
    (> x 111) "first"
    (> x 222) "second"
    (> x 333) "third"
    0 "DEFAULT!"))
```

condp

- 하위에 서술된 식이 한 조건만 검사하는 경우

```
(defn test-condp [num]
  (condp > num
    10 "A"
    20 "B"
    30 "C"
    "X"))

(test-condp 11)
;=> "B"
```

case

- switch문과 같은 역할.
- 검사할 심볼이 같고, Equal 조건만 검사할 경우 사용
- 단, True가 없는 경우 Exception를 던진다. (cond는 nil을 반환)
    - 맨 마지막에 식 하나를 기술하면 Exception을 던지지 않고, 그 식을 반환.

```
(let [fruit "apple"]
  (case fruit
    "banana" "yellow"
     "kiwi" "brown"
     "melon" "green"
     "nothing"))
=> "nothing"
```

Binding

def(전역) vs. let(지역)

def

- **def** SYMBOL VALUE
  - Global binding
  - Immutable global value. (Java의 static final과 비슷한 개념)
  - 전역적으로 사용할 수 있기에 "/"를 이용하여 namespace를 지정할 수 있다. 그리고 그 namespace로 참조 가능.
    - def company/employee "luke"

let

- (**let** [SYMBOL VALUE ...] SYMBOLS)
  - Local binding
  - Lexical Scope. let 안에서 일어나는 일은 let 안에서만 유효.
  - 함수 내부적으로 공통된 계산 값을 이용할 때 사용.

```
(let [sym1 val1
      sym2 val2
      sym3 val3
      ... ]
    <body>)

;symN은 valN이 Binding 된다.
;<body>은 let의 결과인 symN을 재이용할 수 있다. (Lexical Scope)
;마지막에 기술된 <body>는 Return value.
```

```
;let에서 계산된 결과 재활용
(defn square-corners [bottom left size]
  (let [top (+ bottom size)
        right (+ left size)]
    [[bottom left] [top left] [top right] [bottom right]]))

(square-corners 111 222 333)
=> [[111 222] [444 222] [444 555] [111 555]]
```

```
(def my-vector ["A" "B" "C" "D"])

; 1. sym1에 my-vector의 첫 번째 원소와 binding
; 2. elements에 my-vector의 나머지 원소들과 binding
(let [[sym1 & elements] my-vector]
  [sym1 elements])
=> ["A" ("B" "C" "D")]
```

반복문

loop & recur

- 재귀를 이용한 반복.
- 단순 재귀를 사용하면 Stack overflow 발생!
- loop & recur는 Stack을 소모하지 않음.
    - Java의 “label”과 비슷한 문법.
    - loop는 recursion point이고, recur를 만나면 다시 loop문으로 돌아가서 반복.

```
;단순 재귀
;;StackOverFlow
(defn factorial1 [n result]
  (if (= n 1)
    result
    (factorial1 (dec n) (*' result n))))

(factorial1 10000 1)


;loop & recur
(defn factorial2 [n]
  (loop [x n, result 1]
    (if (= x 1)
      result
      (recur (dec x) (*' result x)))))

(factorial2 10000)
```

```
(defn fibonacci [n]
  (loop [x 0, result [1 1]]
         (if (= x n)
           result
           (recur (inc x) (conj result (+' (get result x) (get result (inc
x))))))))

(fibonacci 20)
=> [1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711]
```

```
(defn multiplicationTable [n]
```

```
  (loop [x 1]
    (println n "x" x "=" (* n x))
    (if (= x 9)
      :end!
      (recur (inc x)))))

(multiplicationTable 3)

;결과
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
=> :end!
```

```
(loop [iteration 0]
  (println (str "Iteration : " iteration))
  (if (> iteration 3)
    (println "End!")
    (recur (inc iteration))))


(defn recursive-printer
  ([]
    (recursive-printer 0))
  ([iteration]
    (println (str "Iteration : " iteration))
    (if (> iteration 3)
      (println "End!")
      (recursive-printer (inc iteration)))))

(recursive-printer)


;위 둘다 같은 결과
Iteration : 0
Iteration : 1
Iteration : 2
Iteration : 3
Iteration : 4
End!
=> nil
```

for

Sequence 순회. (foreach)

```
(for [i (range 5)]
  i)
=> (0 1 2 3 4)

;name : symbol이나 키워드의 이름을 가져온다.
(for [alphabet [:a :b :c]
      number [:1 :2]]
  (str (name number) (name alphabet)))
=> ("1a" "2a" "1b" "2b" "1c" "2c")


;:let을 이용한 binding
(for [alphabet [:a :b :c]
      number [:1 :2]
  :let [n (name number)
        a (name alphabet)
        all (str n a)]]
  all)
=> ("1a" "2a" "1b" "2b" "1c" "2c")


;:when 조건을 만족할 때만 반복
(for [alphabet [:a :b :c]
      number [:1 :2]
      :let [n (name number)
            a (name alphabet)
            all (str n a)]
      :when (= number :1)]
  all)
=> ("1a" "1b" "1c")
```

namespace

var에 대한 접근을 조직하고 제어하는 방법. java의 package와 같은 개념.

```
;생성/전환
(ns luke.favfoods)

;현재 namespace 조회
*ns*
```

```
;호출
(def fav-food "noodle")
fav-food
first-clojure.core/fav-food
=> "noodle"
```

함수

- defn

```
; 가장 마지막에 계산한 값을 반환한다!

(defn func-name
  "The docstring(comment) is optional."
  [arg1 arg2 arg3]
  (println arg1 arg2 arg3)
  "This text is garbage."
  "The return value is optional.")

(func-name "hello" "world" "!!!")

;결과
hello world !
=> "The return value is optional."
```

arity overloading

OOP의 Method Overloading과 같은 개념

```
(defn func-name
  "The docstring(comment) is optional."
  ([] (println "Hello Korea!"))
  ([arg1 arg2 arg3] (println arg1 arg2 arg3)))

(func-name)

;결과
Hello Korea!
=> nil
```

variable-arity

가변 인자

- &

```
(defn varargs [& args]
  (str args))

(varargs "hello" "world" "!")
=> "(\"hello\" \"world\" \"!\")"
```

Destructuring

Function의 arguments에서 Collections를 **추려 뽑기**. Collections 일부만 Binding 하기.

- Vectors

```
(let [[color size] ["blue" "small"]]
  (str color " " size))
=> "blue small"

;별칭을 지정하여 전체 출력
(let [[color size :as all] ["blue" "small"]]
  (str color " " size " | " all))
=> "blue small | [\"blue\" \"small\"]"



(defn destructuring-vector1 [[arg1 arg2]]
  arg2)

(destructuring-vector1 [1 2 3 4 5])

;결과
=> 2


(defn destructuring-vector2 [[arg1 arg2 & args]]
  (println (str "arg1 : " arg1))
  (println (str "arg2 : " arg2))
  (println (str "args : " (clojure.string/join args))))

(destructuring-vector2 ["Hello " "World" "Korea" "China" "Japan"])

;결과
arg1 : Hello
arg2 : World
```

```
args : KoreaChinaJapan
=> nil


;키워드 ":as"를 쓰면 Collections 전체 Binding
(defn destructuring-vector3 [[val1 val2 :as nickName]]
  (str val1 " " val2 " " (count nickName)))

(destructuring-vector3 [1 2 3 4])
=> "1 2 4"
```

- Map

```
(defn destructuring-map1 [{val1 :key1 val2 :key2}]
  (println (str "val1 : " val1))
  (println (str "val2 : " val2)))

(destructuring-map1 {:key1 28.22 :key2 81.33})
;결과
val1 : 28.22
val2 : 81.33
=> nil


(defn destructuring-map2 [arg]
  (println (:key2 arg)))

(destructuring-map2 {:key1 "Hello" :key2 "World"})

;결과
World
=> nil
```

```
;:or를 쓰면 해당 요소가 없을 때를 위한 default 값 설정 가능.
(let [{fruit1 :fruit1, fruit2 :fruit2  :or  {fruit2 "banana"}}
      {:fruit1 "apple"}]
  (str fruit1 " " fruit2))
=> "apple banana"


;:as를 쓰면 Collections 전체 binding 가능.
(let [{fruit :fruit  :as  all}
      {:fruit "apple"}]
  (str fruit " | " all))
=> "apple | {:fruit \"apple\"}"
```

```
;;:keys를 이용한 간략화(추상화).
;;;보통 심볼 이름과 key 이름을 갖게 하므로 아래와 같은 식 사용 가능.
;;;Destructuring의 가장 흔한 방식

;변경전
(let [{fruit1 :fruit1, fruit2 :fruit2}
      {:fruit1 "apple", :fruit2 "banana"}]
  (str fruit1 " " fruit2))

;변경후
(let [{:keys [fruit1 fruit2]}
      {:fruit1 "apple", :fruit2 "banana"}]
  (str fruit1 " " fruit2))
=> "apple banana"
```

- etc

```
(let [[x y] [1 2 3]]
          [x y])
=> [1 2]


;Underscore(_)의 의미는 "이 Binding은 Skip!"
(let [[_ _ z] [1 2 3]]
          z)
=> 3


;반환값이 Underscore인 경우 가장 마지막 Underscore에 대응되는 값 반환.
;관용적이지 않다!
(let [[_ _ z] [1 2 3]]
          _)
=> 2


;함수에서 :keys를 이용한 간략화
(defn fruit-kind [{:keys [fruit1 fruit2]}]
  (str fruit1 " " fruit2))

(fruit-kind {:fruit1 "apple", :fruit2 "banana"})
```

Anonymous Functions

고차 함수로 이용.

- 표기법
  - fn [arg] (arg)

    - #(%)
        - 매개 변수가 여러개인 경우 : #(%1 %2 %3)

```
((fn [a] (* a 3)) 8)
(#(* % 3) 8)
=> 24

(map (fn [name] (str "Hi, " name))
     ["Luke", "James"])
(map #(str "Hi, " %)
     ["Luke", "James"])
=> ("Hi, Luke" "Hi, James")


;매개 변수가 여러개인 경우
(#(* %1 %2 %3) 2 3 4)
=> 24
```

Returing Functions

- Clojures : 함수 범위 안에 속하는 모든 변수에 접근할 수 있는 영역

```
(defn adder-builder [arg]
  #(+ % arg))

;함수를 인자로 넣는다.
;이 때 adder는 함수를 반환값으로 받으므로 함수가 아닌 값! 그래서 def
(def adder (adder-builder 1))

(adder 10)

;결과
=> 11
```

분해 및 합성

partial

- Clojure에서 Currying하는 방법.
    - Currying?
        - 다중 인수를 갖는 함수를 여러 개의 단일 인수 함수들로 연결(chain)하는 방식으로 변환하는 기법.
        - 인수를 부분적으로 적용해서 새로운 함수를 만들어내는 기법.

```
(defn grow [name direction]
  (if (= direction :small)
    (str name " is growing smaller")
    (str name " is growing bigger")))

((partial grow "hyuk") :small)
=> "hyuk is growing smaller"
```

comp

여러 함수들을 합성하여 하나의 새로운 함수 생성.

- 단, argument 개수가 같아야 한다.
- comp의 argument로 받은 함수들은 **오른쪽부터 왼쪽으로** 실행한다.

```
(defn toggle-grow [direction]
  (if (= direction :small) :small :big))

(defn show-direction [direction]
  (str "Direction : " direction))

(defn surprise [direction]
  ((comp show-direction toggle-grow) direction))

(surprise :small)
=> "Direction : :small"
```

Lazy Sequences

- 지연(lazy)?
  - 결과가 필요하기 전에는 실행되지 않음. 최종 연산을 수행하기 전까지 중간 연산을 수행하여 미룸.
  - **실제 실행이 될때**까지 최종 처리를 미루는 작업.
  - 무거운 작업(응답이 느린 작업)을 수행할 때, 그 결과를 나누거나 걸러내어 최종 처리 결과를 받을 때 사용한다. (Java의 Stream API, Rx와 같은 개념)
- 처리 결과가 clojure.lang.LazySeq인 경우.

```
;;가장 안쪽에 있는 함수들이 무한 처리 작업

(take 5 (range))
=> (0 1 2 3 4)

(take 5 (repeat "x"))
=> ("x" "x" "x" "x" "x")
```

```
;repeat는 중간처리 결과를 한번만 실행하고 반복
;repeatedly는 인수로 받은 함수를 반복 실행
(take 5 (repeatedly #(rand-int 10)))
=> (0 8 8 0 0)

;cycle은 Collections를 인수로 받는다
(take 5 (rest (cycle ["a" "b" "c"])))
=> ("b" "c" "a" "b" "c")
```

map

결과값은 입력 Collections의 각 요소에 특정 기능의 함수가 적용된 새로운 Collections.

- 만약 여러 개의 Collections에 적용하면 **가장 짧은 Collections**에 맞춰 종료.

```
(def data [:a :b :c :d :e])
(map #(str %) data)
=> (":a" ":b" ":c" ":d" ":e") ;각각의 요소에 대해 문자열로 변경된 Collections.


(defn get-text [alphabet number]
  (str alphabet number))

(def data1 ["a" "b" "c" "d" "e"])
(def data2 [1 2 3])

(map get-text data1 data2)
=> ("a1" "b2" "c3")
```

reduce

- 점화적으로 연산되기 때문에 유한. 즉, 무한 Sequence를 다룰 수 없다.
- “누적” 연산

```
;;factorial
;range 1 6 : 1~5
(reduce #(* %1 %2) (range 1 6))
(reduce #(* %1 %2) (map #(inc %) (take 5 (range))))
=> 120
```

Concurrency

- future : fork와 같은 개념. 다른 Thread 생성.

Type	Communication	Coordination(≒ Thread-safe)
atom	Synchronous	Uncoordnated
ref	Synchronous	Coordnated
agent	Asynchronous	Uncoordnated

atom

독립적인 동기적 변경

- reset! : argument가 값
- swap! : argument가 함수

```
(def who-atom (atom :before-change))

;값으로 변경
(reset! who-atom :after-change)
@who-atom
=> :after-change


;함수로 변경
(swap! who-atom
       #(case %
          :before-change "before!"
          :else))
@who-atom
=> :else
```

ref & dosync

Transaction이 필요한 동기적 변경 (Critical Section에 접근하는 경우)

- alter : Transaction동안 **재시도**. (성능 고려 필요)
- commute : Transaction 동안 **재시도를 하지 않음**. 그리고 **반드시** commute를 적용할 함수는 가환적(commutative)(즉, 더하기처럼 연산 순서에 따라 결과가 바뀌지 않는다)이거나 마지막 Thread의 결과를 사용하는 성질이 있어야 한다.

```
(def acc1 (ref 100))
(def acc2 (ref 200))

(println @acc1 @acc2)
;100 200

(defn transfer-money [a1 a2 amount]
  (dosync
```

```
    (alter a1 - amount)
    (alter a2 + amount)
    amount))

(let [n 2]
  (future (dotimes [_ n] (transfer-money acc1 acc2 10)))
  (future (dotimes [_ n] (transfer-money acc1 acc2 10)))
  (future (dotimes [_ n] (transfer-money acc1 acc2 10))))

(println @acc1 @acc2)
;40 260
```

agent

독립적인 비동기적 변경. 아래 변경하는 함수들은 **argument로 함수**를 받는다.

- send : cpu 작업이 많은 연산에 적합한 고정 Thread-pool 사용.
- send-off : I/O 작업이 많은 Thread-pool이 대기 상태에 빠지지 않도록 확장 Thread-pool 사용.

```
(def who-agent (agent :before-change))
(send who-agent
      #(case %
         :before-change "before!"
         :else))
@who-agent
=> "before!"
```

Transaction 처리

```
(def who-agent (agent :before-change))
(send who-agent #(throw (Exception. "Exception!"))) ;의도적인 Exception
(send who-agent #(case %
                   :before-change "before!"
                   :else))
;CompilerException java.lang.RuntimeException: Agent is failed, needs
restart, compiling:(...)

@who-agent
;값이 변경되지 않음.
;=> :before-change



;;이 경우 restart-agent를 사용해야 한다.
```

```
;;restart-agent 함수로 재시작하기전까지 실패한 상태 유지.
;;;restart-agent는 Error를 제거하고 agent의 초기값을 재설정한다.

(restart-agent who-agent :before-change)
(send who-agent #(case %
                   :before-change "before!"
                   :else)) ;성공!
@who-agent
;=> "before!"
```

Java interop

Java 라이브러리 가져다 쓰기 https://clojure.org/reference/java_interop

- method 호출 : .METHOD
- Instance 생성 : INSTANCE. 혹은 new INSTANCE
- static method 호출 : CLASS/METHOD

Polymorphism

multimethods

한개의 함수를 대상으로 다형성 구현. Java에서 Abstract Method를 정의하고, 상속된 Class에서 이를 구현하는 것과 비슷함.

- 다양하게 유연한 방법으로 분기가 가능하지만 protocol에 비해 성능 저하가 따른다.

```
;① 분기(dispatch)를 결정하는 함수. 마지막 argument가 분기하는 기준.
(defmulti polymorphism1 class)

;② 구현체
(defmethod polymorphism1 String [input]
  (type input))

(defmethod polymorphism1 Keyword [input]
  (type input))

(defmethod polymorphism1 Long [input]
  (type input))

(defmethod polymorphism1 :default [input]
  "NONE!")

;;실행
(polymorphism1 213)
;=> java.lang.Long
```

```
(polymorphism1 :hello)
;=> clojure.lang.Keyword
```

```
;① 분기(dispatch)를 결정하는 함수. 마지막 argument가 분기하는 기준.
;;이 때 인자는 Map
(defmulti greeting #(% :language))

;② 구현체
(defmethod greeting "english" [parmas]
  "Hello!")

(defmethod greeting "french" [parmas]
  "Bonjour!")

(defmethod greeting :default [params]
  (throw (IllegalArgumentException. (str "Unknown language : " (params
:language)))))


;실행
(greeting {:id 1, :language "english"})
;=> "Hello!"
(greeting {:id 2, :language "french"})
;=> "Bonjour!"
(greeting {:id 3, :language "korean"})
;CompilerException java.lang.IllegalArgumentException: Unknown language :
korean, compiling:(...)
```

```
;① 분기(dispatch)를 결정하는 함수 (≒ Interface)
;; 함수로도 argument를 받을 수 있다.
(defmulti polymorphism2 #(if (< % 3) :fn1 :fn2))

;② 구현체
(defmethod polymorphism2 :fn1 [_]
  "Function 1")

(defmethod polymorphism2 :fn2 [_]
  "Function 2")

;실행
(polymorphism2 5)
;=> "Function 2"
```

Protocol

다수의 함수를 대상으로 다형성 구현.

- Type으로 분기
- 자주 사용하지 않는 것이 좋다. 대부분의 상황에서 순수 함수나 Multi Method로 대신 사용할 수 있다.

```
(defprotocol abstraction (impl [this]))

(extend-protocol abstraction
  String
  (impl [this]
    (type this))

  Keyword
  (impl [this]
    (type this))

  Long
  (impl [this]
    (type this)))



(impl "test")
;=> java.lang.String
(impl :test)
;=> clojure.lang.Keyword
(impl 1)
;=> java.lang.Long
```

```
(defprotocol Fly (fly [this] "Method to fly"))

(defrecord Bird [name species] Fly ;상속
  (fly [this]
    (str (:name this) " flies...")))

(extends? Fly Bird)
;=> true

(def crow (Bird. "Crow" "Black!"))

(fly crow)
;=> "Crow flies..."
```

새로운 Type 정의

- defrecord vs. deftype
  - 구조화된 데이터가 필요하면 defrecord
  - 단순히 Type만 필요한 경우 deftype. (Java의 argument 없는 생성자의 객체의 Type이 필요할 때)

defrecord

```
(defrecord Person [name age])
(def hyuk (Person. "hyuk" 30))

(.-name hyuk)
=> "hyuk"
(.-age hyuk)
=> 30
```

```
(defprotocol Action
  (eat [this])
  (said [this]))

(defrecord Person [name age]
  Action
  (eat [this]
    (str name " eat somthing..."))
  (said [this]
    (str "My age is " age)))

(def hyuk (Person. "hyuk" 30))


(.-name hyuk)
;=> "hyuk"
(.-age hyuk)
;=> 30

(eat hyuk)
;=> "hyuk eat somthing..."
(said hyuk)
;=> "My age is 30"
```

deftype

```
(defprotocol Action
  (eat [this])
  (said [this]))

(deftype Person []
  Action
  (eat [this]
    (str "Somebody eat somthing..."))
  (said [this]
    (str "My age is secret.")))

(def somebody (Person.))


(eat somebody)
;=> "Somebody eat somthing..."
(said somebody)
;=> "My age is secret."
```

Macro

Threading Macro

코드 재배열을 하여 가독성을 높이는 매크로

- https://clojure.org/guides/threading_macros

Thread-first Macro

https://clojuredocs.org/clojure.core/-%3E

- (-> x forms)
  - forms에 x를 연쇄적으로 두 번째 아이템으로써 뒤에 삽입.
- 함수를 여러번 중첩하는 경우 유용.

```
(get (.split (.toUpperCase "a b c d") " ") 1)
(-> "a b c d" .toUpperCase (.split " ") (get 1))

=> "B"
```

```
(second (next [1 2 3 4 5]))
(-> [1 2 3 4 5] next second)
=> 3
```

Thread-last Macro

https://clojuredocs.org/clojure.core/-%3E%3E

- (->> x forms)
  - forms에 x를 연쇄적으로 첫 번째 아이템으로써 앞에 삽입.
- Collections를 마지막 argument로 받는 함수들에 유용

```
(take 3 (filter odd? (range)))
(->> (range) (filter odd?) (take 3))

=> (1 3 5)
```

var 삭제 방법

```
;현재 namespace 사용
(ns-unmap *ns* 'var)

;직접 "user"라는 namespace 지정
(ns-unmap (find-ns 'user) 'var)
```

From:
https://wiki.ledyx.xyz/ - **Ledyx WIKI**

Permanent link:
https://wiki.ledyx.xyz/doku.php?id=language:clojure&rev=1513907589

Last update: **2026/10/24 07:03**