**INTRODUCTION**

|  |  |  |
| --- | --- | --- |
| PROJECT RESOURCES | | GIT REPO OR ZIP ARCHIVE |
| ENVIRONMENT SETUP | | JAVA, CLOJURE ON MAC AND PC |
| REPL | | TEST ENVIRONMENT |
| INTELLIJ IDE | | PROGRAMMING EDITOR |
| FIRST PROJECT | | HELLO WORLD |
| LANGUAGE BASICS | | FUNCTIONS, DATA TYPES, ETC., |
| CODING CHALLENGE | | **--** |
| LANGUAGE FUNDAMENTALS | | CONDITIONALS, LOOPS, ETC., |
| Coding challenge | **--** | | |
| CLOJURE COMPLEXITY | CONCURRENT PROGRAMMING | | |
| CODING CHALLENGE |  | | |

WHAT IS CLOJURE??

* PROGRAMMING LANGUAGE THAT RUN ON TOP OF THE JVM AND JAVASCRIPT.
* WORKS WITH YOUR EXISTING SYSTEMS AND CODEBASE
* A LISP LANGUAGE
* IT IS COMPACT AND FAST

SIZE IS 1/10TH OF JAVA CODE AND IT WILL PROVIDES HIGH PERFORMANCE.

WHAT IS IT USED FOR??

1. SMALL WEB PAGES
2. SHELL SCRIPTS
3. LARGE WEBSITES
4. MACHINE LEARNING
5. SEARCH ENGINES
6. DATA MINING
7. PARALLEL PROCESSING etc.,

TOOLS:

* TEXT EDITOR
* LEININGEN => FEW SHORTCUTS IN THE PORJECT
* REPL
* INTELLIJ IDEA

GET THE CODE:

ZIP FILE/ GITHUB REPOSITORY.

EXPLANATION OF CODE:

COMPLIOR:

* EVERYTHING IS A LIST

(VERB PARAMETER1 PARAMETER2 ……)

* EVERYTHING HAS A RETURN VALUE
* THERE IS NO ‘BREAK’ OPERATOR
* CHAINING OPERATIONS

(Println ‘ALICE HAS’ (+ 2 3) ‘APPLES’)

O/p: ALICE HAS 5 APPLES

FOR EVERY PROJECT WE NEED TO CREATE SEPARATE NAMESPACE

namespace

(ns Tutorial.Core

(: gen-class))

it will generate Java class

1. (defn -main

“I don’t do a whole lot… yet.”🡪 this line skip the execution of statement.

[& args] 🡪 this is for zero & more parameters.

(println “Hello world!”)

1. (defn -main

[x]

(println “Hello world!” x)

(-main “Madhu”) 🡪 It should be either value or string

Functions:

1. (ns tutorial.functions

(:gen-class))

(defn -main

“First Function”

[]

(println “My name is Madhu”) O/p: My name is Madhu

(println “Loving clojure so far”) Loving Clojure so far

(+ 2 5)

)

1. (# (println “hello” %) “Madhu”) O/p: hello Madhu

(# (println “hello” %1 %2 %3) “Madhu” “your age is” “24”)

O/p: hello Madhu your age is 24

(# (println “hello” %1 “your age is” %2) “Madhu” “24”)

O/p: hello Madhu your age is 24

1. (def increment (fn [x] (+x 1)))

(defn increment-set

[] O/p: (2 3 4)

(map increment [1 2 3])

(increment-set)

1. (defn increment-set

[x]

(map oncrement x) O/p: (2 6 3 8 10 5)

)

(increment-set [1 5 2 7 9 4])

Data Types:

(defn Datatypes

[]

(def a 1)

(def b 1.25)

(def c 1.25e-12)

(def d 0Xfbfc)

(def e nil)

(def f true)

(def g “Hello”)

(def h ‘thanks)

(println a) O/p: 1

(println b) O/p: 1.25

(println c) O/p: 1.25E-12

(println d) O/p: 64508

(println e) O/p: nil

(println f) O/p: true

(println g) O/p: Hello

(println h) O/p: thanks

)

(Datatypes)

Integers:

* Short
* Long
* Octal
* Hexadecimal
* Atoms

Java Datatypes:

* Java.Lang.byte
* Java.Lang.short
* Java.Lang.integer
* Java.Lang.long
* Java.Lang.float
* Java.Lang.double

Variables:

(defn variables

[]

(def status true)

(def STATUS false)

(println status)

(println STATUS)

)

(variables)

Operators:

* **ARITHMETIC OPERATORS**
  + **+ {(+ 2 3)}**
  + **– {(- 3 2)}**
  + **\* {(\* 2 3)}**
  + **/ {(/ 4 2)}**
  + **Inc {(println (inc 3))}**
  + **Dec {(println (dec 3))}**
  + **Max {** **(println (max 2 3 4 5 8 9 7 6))}**
  + **Min {** **(println (min 2 3 4 5 8 9 7 6))}**
  + **Rem {** **(println (rem 3 2))}**
* **RELATIONAL OPERATORS**
* **=**
* **NOT=**
* **<**
* **<=**
* **>**
* **>=**
* **LOGICAL OPERATORS**
* **AND**
* **OR**
* **NOT**
* **BITWISE OPERATORS**

|  |  |  |
| --- | --- | --- |
| **BIT-AND** | **0101 1100** | **0100** |
| **BIT-OR** | **0101 1100** | **1101** |
| **BIT-XOR** | **0101 1100** | **1001** |
| **BIT-NOT** | **0101 1100** | **1010** |

* **OPERATOR PRECEDENCE**

JAVA =🡺 2+5/3-1

CLOJURE =🡺 (- (1(+ 2 5) 3) 1)

* **COMPOSITE/COMPLEX DATATYPES**
* Contain one or more simple datatypes.

**Set:**

* Set of different types of values
* Immutable
* # {} 🡺empty set
* #{1 4 2 1.5 “PET” ‘Cat}

**Map:**

{:KEY VALUE1,:KEY2 VALUE2}

* Immutable
* Efficient

# {:Key1 “Value1”}

# {1 42, 2 1.5, “PET” ‘Cat}

**Vector:**

* Arrays
* Immutable
* Efficient
* Indexed by position
* [1 2 3 4]
* [1 “two” ‘three]
* []

**List:**

* Make up the code
* Immutable
* Efficient
* ( 1 2 3 4)
* (1 “two” ‘three (1 2 3 4))
* (defn Foo[]

(println “Hello”)

)

(Foo)

**::Coding Challenge::**

* Function that gives us the age of animal in human years
* Tip:

Dog Cat Fish

\*7 \*5 \*10

**Code:**

(defn petToHumanAge [x]

(def petStore {'dog 7, 'cat 5, 'goldfish 10})

(get petStore x)

)

(defn age [petName petType petAge]

(def ratio (petToHumanAge petType))

(println petName "is" (\* ratio petAge) "years old in Human Years")

)

(age "fido" 'dog 4)

(age "fifi" 'cat 2)

(age "Bubble" 'goldfish 10)

**::Conditionals::**

(defn condIf []

(println "\ncondif:")

(if (= 5 5)

(println "equal")

(println "not Equal")

)

)

(condIf)

(defn CondIfDo

[]

(println "\nCondIfDo:")

(if (= 5 5)

(do (println "Equal first statement")

(println "Second statement"))

(do (println "not equal first statement")

(println "second statement"))

)

)

(CondIfDo)

(defn CondNestIf

[]

(println "\n NestedIf:")

(if (and (= 5 5) (or (= 2 2) (not true)))

(println "true")

(println "false")

)

)

(CondNestIf)

(defn CondCase

[pet]

(println "\nCondCase:")

(case pet

"cat" (println "I have a cat")

"dog" (println "I have a dog")

"fish" (println " I have a goldenfish")

)

)

(CondCase "fish")

(defn CondCond

[amount]

(println "\nCondCond:")

(cond

(<= amount 2)(println "few")

(<= amount 10)(println "several")

(<= amount 100)(println "many")

)

)

(CondCond 50)

**::Loops::**

(defn Loop

[]

(println "\nLoop:")

(loop [x 0]

(when (< x 10)

(println x)

(recur (inc x))

)

)

)

(Loop)

(defn DoTimes

[]

(println "\nDoTimes")

(dotimes [x 10]

(println x)

)

)

(DoTimes)

(defn WhileDo

[count]

(println "\nWhileDo")

(def x (atom 0))

(while (< @x count)

(do

(println @x)

(swap! x inc)

)

)

)

(WhileDo 10)

(defn Doseq

[seq]

(println "\nDoseq:")

(doseq [x seq]

(println (inc x))

)

)

(Doseq [6 3 5 8 4 2 6])

**::Atoms::**

* Atoms are thread safe
* Networks oriented and thread oriented so the principle different thread accessing the same values and changing information simultaneously.

(defn Atoms

[]

(def amount (atom 100))

(println "The atom value is:" @amount)

(swap! amount inc)

(println "The amount value after inc is:" @amount)

(reset! amount 110)

(println "The reseted value of the atom is:" @amount)

(compare-and-set! amount 110 120)

(println "the updated value of the atom is:" @amount)

(compare-and-set! amount 110 150)

(println "the updated value of the atom is:" @amount)

)

(Atoms)

**::Sequences::**

(ns tutorial.Sequences)

(defn Seq

[]

(def colors (seq ["red" "green" "blue"]))

(println colors) O/p: (red green blue)

(println (cons "orange" colors)) O/p: (orange red green blue)

(println (cons colors "orange")) 🡪colors is seq then yellow will add in seq form

O/p: ((red green bule) o r a n g e)

(println (conj colors "orange")) O/p: (orange red green blue)

(println (conj ["red" "yellow" "green"] "orange")) O/p: [red yellow green orange]

(println (concat colors (seq ["black" "white"]))) O/p:(red green blue black white)

(println (distinct (seq [1 2 3 5 3 5 2 4]))) O/p: (1 2 3 5 4)

(println (reverse colors)) O/p: (blue green red)

(println (first colors)) O/p: red

(println (rest colors)) O/p: (green blue)

(println (last colors)) O/p: blue

(println (sort (seq [1 2 3 5 3 5 2 4]))) O/p: (1 2 2 3 3 4 5 5)

)

(Seq)

**::StructMaps::**

It structure the data and it provides more information and more lower level information.

**Code:**

(ns tutorial.StructMaps)

(defn Pets

[]

(defstruct pet :PetType :PetName)

(def myPet (struct pet "dog" "Fido"))

(println myPet)

(def myOtherPet (struct-map pet :PetName "fifi" :PetType "cat"))

(println myOtherPet)

(println (:PetName myPet))

(println (:PetType myOtherPet))

(def myNewPet (assoc myPet :PetName "maxi"))

(println myNewPet)

(def myNewOtherPet (assoc myOtherPet :petAge 10))

(println myNewOtherPet)

)

(Pets)

**::Destructuring::**

(ns tutorial.Destructuring)

(defn Destruct

[]

(def myVect [1 2 3 4 5])

(let [[a b c] myVect] (println a b c))

(let [[a b &rest] myVect] (println a b rest))

(let [[a b c d e f] myVect] (println a b c d e f))

(def myMap {'name "john" 'lastname "smith"})

(let [{a 'name b 'lastname} myMap] (println a b))

(let [{a 'name b 'lastname c 'noname} myMap] (println a b c))

)

(Destruct)

**::Exception Handling::**

(ns tutorial.Exceptions)

(defn ExHandling

[x]

(try

(inc x)

(println "no exception occured, the value is" (inc x))

(catch ClassCastException e (println "caught exception:" (.getMessage e)))

(catch Exception e (println "Caught generic exception:"))

(finally (println "cleanup and move on"))

)

)

(ExHandling "hello")

(ExHandling 1)

**::Coding Challenge::**

* **CAR** Dealership with 3 types of cars

**BMW 60,000** COUPON **VALID**

**Ferrari 1,00,000** BUDGET 50,000

**Fiat 20,000** THE COUPON IS VALID

Coupon code with **20% discount**  BMW 48,000

Function[Budget code] returns car prices. FIAT 16,000

COUPON **NOT VALID**

BUDGET 50,000

THE COUPON IS INVALID

FIAT 20,000

**Code:**

(ns tutorial.CarDealership)

(defn isCodeValid

[code]

(defstruct Coupon :Name :Discount)

(def validCoupon (struct coupon "20Percent" 0.8))

(if (= (:Name validCoupon) code)

true

false

)

)

(defn getCarPrices

[budget code]

(def Cars {"BMW" 60000,"Farrari" 100000,"Fiat" 20000})

(if (isCodeValid code)

(do

(println "the code is valid")

(def discount (:Discount validCoupon))

(doseq [car Cars]

(def CarType (first car))

(def price (last car))

(def priceDiscount (\* price discount))

(if (<= priceDiscount budget)

(println "the" CarType "costs" priceDiscount)

)

)

)

(do

(println "the code is invalid")

(doseq [car Cars]

(def CarType (first car))

(def price (last car))

(if (<= price budget)

(println "the" CarType "costs" price)

)

) O/p: The code is valid

) The BMW costs 48000.0

) The Fiat costs 16000.0

)

(getCarPrices 50000 "20Percent")

**::Namespaces::**

* It defines certain number of functions and number of variables.

(ns tutorial.Namespaces

(:require [clojure.string :refer :all])

)

(defn -main

[]

(println (capitalize "hello"))

)

(-main)

**::Macros::**

* Macros are the functions that can do transfer one code to one code.
* It can accepts the list of code expressions and produces the list of code expression.

**Code:**

(println (macroexpand-1 '(when (= 2 2) (println "hello"))))

**o/p: (if (= 2 2) (do (println hello)))**

(println (macroexpand-1 '(when true (println "hello"))))

**o/p: (if true (do (println hello)))**

**::Watchers::**

* It is a very co-feature of Clojure, it allows to attach a function to certain datatypes like atom, those functions will be invoked when ever that datatypes changed.

(ns tutorial.Watchers)

(defn Watch

[]

(def x (atom 5))

(add-watch x :xWatcher

(fn [key atom old-state new-state]

(println key)

(println atom)

(println old-state)

(println new-state)

)

)

(reset! x 10) in this line it is resetting value

(remove-watch x :xWatcher) in this line it is remove the watcher

(reset! x 15) in this line it not changing the value because the watcher removed

)

(Watch)

O/p:- :xWatcher

#object[clojure.lang.Atom 0x4f7c0be3 {:status :ready, :val 10}]

5

10

**::Agents::**

* Agents are another datatype in clojure that we can use update the value.
* In atoms we are waiting for datatype to update the value, with agents we just notify the system that we want to update that in background.

(ns tutorial.Agents)

(defn Agents

[]

(def amount (agent 100))

(println @amount) 100

(send amount inc)

(println @amount) 100

(println "some time must pass") some time must pass

(println @amount) 101

(send amount inc)

(await-for 1000 amount)

(println @amount) 102

(println (agent-error amount)) nil

)

(Agents)

**Reference values**

* There is coordinate changes which means that we will able to effect changes on the datatypes only certain situations.

(ns tutorial.References)

(defn Refs

[]

(def amount (ref 100))

(println @amount) 100

(dosync

(ref-set amount 110)

)

(println @amount) 110

(dosync

(alter amount inc)

)

(println @amount) 111

)

(Refs)

**::Coding Challenge::**

* Bank with 2 accounts
  + Buyer Account 100
  + Merchant Account 0
* Items for sale
  + Pen 1
  + Notebook 5
  + Backpack 10

**Taks:** Build a function to buy an item is able to be called multiple items to purchase multiple items.

**Solution:**

(ns tutorial.BankTransactions)

(def buyerAccount (ref 100))

(def merchantAccount (ref 0))

(def prices {'pen 1,'notebook 5, 'backpack 90})

(def items (ref []))

(defn PriceInfo

[]

(println "\n priceInfo:")

(println "buyerAccount:" @buyerAccount)

(println "merchantAccount:" @merchantAccount)

(println "items:" @items)

)

(defn buy

[item]

(def itemprice (get prices item))

(if (<= itemprice @buyerAccount)

(dosync

(ref-set merchantAccount (+ @merchantAccount itemprice))

(ref-set buyerAccount (- @buyerAccount itemprice))

(def newItmes (cons item @items))

(ref-set items newItmes)

)

(println "Insufficient funds")

)

(PriceInfo)

)

(buy 'pen)

(buy 'notebook)

(buy 'backpack)

(buy 'notebook)

(buy 'pen)

**O/p:**

priceInfo:

buyerAccount: 99

merchantAccount: 1

items: (pen)

priceInfo:

buyerAccount: 94

merchantAccount: 6

items: (notebook pen)

priceInfo:

buyerAccount: 4

merchantAccount: 96

items: (backpack notebook pen)

Insufficient funds

priceInfo:

buyerAccount: 4

merchantAccount: 96

items: (backpack notebook pen)

priceInfo:

buyerAccount: 3

merchantAccount: 97

items: (pen backpack notebook pen)

**XOX game:**

(defn triple-winner? [triple]

(if (every? #{:x} triple) :x

(if (every? #{:0} triple) :o))

)

(declare triples)

(defn winner? [board]

(first (filter #{:x :o} (map triple-winner? (triples board))))

)

(defn triples [board]

(concat

(partition-all 3 board)

(list

(take-nth 3 board)

(take-nth 3 (drop 1 board))

(take-nth 3 (drop 2 board))

(take-nth 4 board)

(take-nth 2 (drop-last 2 (drop 2 board)))

)

)

)

(defn full-board? [board]

(every? #{:x :o} board))

(defn print-board [board]

(let [board (map #(if (keyword? %) (subs (str %) 1)%) board)]

(println (nth board 0) (nth board 1) (nth board 2))

(println (nth board 3) (nth board 4) (nth board 5))

(println (nth board 6) (nth board 7) (nth board 8))

)

)

(defn player-name [player]

(subs (str player) 1)

)

(def starting-board [1 2 3 4 5 6 7 8 9])

(def player-sequence (cycle [:x :o]))

(defn get-move [board]

(let [input (try

(. Integer parseInt (read-line))

(catch Exception e nil))]

(if (some #{input} board)

input

nil)))

(defn take-turn [player board]

(println "select your move, player" (player-name player) "(press 1-9 and hit enter):")

(loop [move (get-move board)]

(if move

(assoc board (dec move) player)

(do

(println "move was invalid. select your move, player" (str (player-name player) ":"))

(recur (get-move board))

)

)

)

)

(defn play-game []

(loop [board starting-board player-sequence player-sequence]

(let [winner (winner? board)]

(println "current board:")

(print-board board)

(cond

winner (println "player" (player-name winner) "wins!")

(full-board? board) (println "game is a draw.")

:else

(recur

(take-turn (first player-sequence) board)

(rest player-sequence)

)

)

)

))

(play-game)

**Snake game:**

(ns example snake

(:import

(java awt Color Dimension)

(javax swing JPanel JFrame Timer JOptionPane)

(java awt event event ActionListener KeyListener KeyEvent)))

(def field-width 50)

(def field-height 30)

(def point-size 15)

(def turn-millis 100)

(def win-length 10)

(def directions

{KeyEvent/VK\_LEFT [-1 0]

KeyEvent/VK\_RIGHT [1 0]

KeyEvent/VK\_UP [0 -1]

KeyEvent/VK\_DOWN [0 1]})

(defn create-snake []

{body (list [3 0] [2 0] [1 0] [0 0])

:direction [1 0]

:type :snake

:color (Color. 15 160 70)})

(defn create-apple []

{:location [(rand-int field-width) (rand-int field-height)]

:color (Color. 210 50 90)

:type :apple

}

)

(defn point-to-screen-rect [[pt-x pt-y]]

[(\* pt-x point-size) (\* pt-y point-size) point-size point-size]

)

(defn move [{:keys [body direction] :as snake} & grow]

(assoc snake :body

(cons

(let [[head-x head-y] (first body)

[dir-x dir-y] direction]

[(+ head-x dir-x) (+ head-y dir-y)])

(if grow body (butlast body)))))

(defn turn [snake direction]

(assoc snake :direction direction)

)

(defn win? [{body :body}]

(>= (count body) win-length))

(defn head-overlaps-body? [head body]

(contains? (set body) head)

)

(defn head-outside-bounds? [[head-x head-y]]

(or

(> head-x field-width)

(< head-x 0)

(> head-y field-height)

(< head-y 0)

))

(defn lose? [{[head & body] :body}]

(or (head-overlaps-body? head body)

(head-outside-bounds? head)

))

(defn eats? [{[head] :body} {apple :location}]

(= head apple)

)

(defn update-positions [snake apple]

(dosync

(if (eats? @snake @apple)

(do

(ref-set apple (create-apple))

(alter snake move :grow))

(alter snake move)

))

nil)

(defn update-direction [snake direction]

(dosync (alter snake turn direction))

nil)

(defn reset-game [snake apple]

(dosync

(ref-set snake (create-snake))

(ref-set apple (create-apple))

)

nil)

(defn fill-point [g pt color]

(let [[x y width height] (point-to-screen-rect pt)]

(.setColor g color)

(.fillRect g x y width height)

))

(delmulti paint (fn [g object] (:type object)))

(defmethod paint :apple [g {:keys [location color]}]

(fill-point g location color))

(defmethod paint :snake [g {:keys [body color]}]

(doseq [point body]

(fill-point g point color)))

(defn game-panel [frame snake apple]

(proxy [JPanel ActionListener KeyListener] []

(paintComponent [g]

(proxy-super paintComponent g)

(paint g @apple)

(paint g @snake))

(getPrefeeredSize []

(Dimension. (\* (inc field-width) point-size)

(\* (inc field-height) point-size)))

(actionPerformed [e]

(updated-positions snake apple)

(if (lose? @snake)

(do

(reset-game snake apple)

(JOptionPane/showMessageDialog frame "You lose!")))

(if (win? @sanke)

(do

(reset-game snake apple)

(JOptionPane/showMessageDialog frame "You win!")))

(.repaint this))

(keypressed [e]

(let [direction (directions (.getKeyCode e))]

(if direction (update-direction snake direction))))

(keyReleased [e])

(keyTyped [e])))

(defn game []

(let [snake (ref (create-snake))

apple (ref (create-apple))

frame (JFrame. "Snake")

panel (game-panel frame snake apple)

timer (Timer. turn-millis panel)]

(.setFocusable panel true)

(.addKeyListener panel panel)

(.add frame panel)

(.pack frame)

(.setDefaultCloseOperation frame JFrame/EXIT\_ON\_CLOSE)

(.setVisible frame true)

(.start timer)))

(game)