import UIKit

import Foundation

var name1 = "Paulo"

var name2 = "Paulo"

name1 == name2

// ARRAY

// Declaracao de array (Array de String com inferencia)

var invites = [

"Paulo",

"Fernanda",

"João",

"Lulu"

]

//inserindo elemento em um Array

invites.append("Lala")

// ou inserir varios elementos

invites += ["Jaber", "Marcos"]

//print(invites)

// Pegar uma posicao do Array, neste caso esta pegando o indice 1

//print("A \(invites[1]) foi convidada")

// Remover um elemento do Array pelo indice

invites.remove(at: 1)

// Verificar se o Array contem certo elemento. Retorna Boolean

invites.contains("Fernanda")

// Verificar o indice do elemento. Retorna um Optional

invites.index(of: "Jaber")

// Então, para remover o elemento "Jaber",

// obtemos o indice dele, desembrulhando o retorno do metodo index(of:),

// para depois usar o metodo remove(at:)

if let billyIndex = invites.index(of: "Jaber"){

 invites.remove(at: billyIndex)

}

// Duas formas de criar um Array vazio

var grades: [Double] = []

var grade1 = [Double]()

// DICTIONARY (Dicionário)

/\*

  Um dicionario eh composto por chave e valor.

  Pode-se deixar o tipo de dado por inferencia ou tipar logo na criacao,

mas caso o dicionario seja criado vazio, precisa ser tipado na criacao.

  Chave e valor podem ser de varios tipos.

\*/

// Declarando um dicionario por inferencia

var aurelio =

    [ "yudi": "um babaca",

        "swift": "legal" ]

var states = [

    "SP": "São Paulo",

    "RJ": "Rio de Janeiro"

]

// Declarando um dicionario definindo o tipo

var numberString: [Int : String] = [

    1: "One",

    2: "Two",

    3: "Three"

]

// Duas formas de declarar um dicionario vazio

var emptyDictionary: [Int: String] = [:]

var emptyDict = [Int: String]()

// Para acessar o valor de um elemento, usa-se a chave, ou a posicao dentro dos colchetes []

//print("O Eric nasceu no estado \(states["SP"]!)")

// Como nao eh certeza que existe um valor com a chave informada nos [], o correto eh usar o Optional Bind para desembrulhar.

// Entao, podemos desembrulhar states["SP"] colocando dentro da constante state, e entao imprimir a constante.

if let state = states["SP"]{

    print("O Eric nasceu no estado \(state)")

}

// Incluindo novo valor dentro do dicionario.

states["AM"] = "Amazonas"

// Para remover um elemento, atribui-se nil à chave que se quer remover.

states["AM"] = nil

// Count eh usado para retornar a quantidade de elementos do Array ou dicionario

print("O meu dicionário de estados tem \(states.count) elementos")

print("Já estou com \(invites.count) convidados para a minha festa")

// Tambem pode ser usado como o .lenght(), contador de quantas letras existem na string

"Jaber Shamali".count

/\*-----------------------------------------\*/

// CONTROLE DE FLUXO

// For in (equivale ao for each)

for person in invites {

    //print("O(a) \(person) foi convidado")

}

// While

var index = 0

while index < invites.count{

    //print("O \(invites[index]) foi convidado")

    index += 1

}

/\*

// Range = intervalo entre dois numeros (nao pode ser decrescente)

for i in 0...invites.count {

    print("O(a) convidado de numero \(i+1) é \(invites[i])")

}

\*/

// Half Open Range = intervalo entre dois numeros -1

for i in 0..<invites.count {

    //print("O(a) convidado de numero \(i+1) é \(invites[i])")

}

// Laço comum imprimindo os valores do Dicionario

for state in states{

    print(state.value)

}

// Laço comum, decompondo a tupla (linha do dicionario), para usar chave e valor separados.

for (key, value) in states {

    print(key, value)

}

// Exemplo de tupla

let address: (street: String, number: Int, city: String) = ("Rua Uma", 100, "Rio de Janeiro")

// Decompondo a tupla criada acima em variaveis

//var (street, number, city) = address

// Se não for usar algum elemento da tupla, use underline(\_) no lugar da variavel que receberia seu valor.

var (street, \_, city) = address

/\*-----------------------------------------\*/

// CRIANDO OPERADORES

// PREFIX

// Declara o operador, definindo se eh Prefix, Postfix ou Infix

prefix operator >-

// Cria a funcao que define o comportamento do operador

prefix func >- (rhs: Int) -> Int{

    return rhs \* rhs

}

// usando o operador prefix criado

>-4

// POSTFIX

postfix operator <-

postfix func <- (lhs: Int) -> Int{

    return lhs \* lhs \* lhs

}

// usando o operador postfix criado

5<-

// INFIX (na funcao de um operador infix nao precisa colocar infix antes da palavra reservada func)

infix operator >-<

func >-< (lhs: Int, rhs: Int) -> Int{

    return (lhs + lhs) \* (rhs + rhs)

}

// Usando o operador infix criado

2 >-< 3

arc4random\_uniform(20)

var a = 6 >-< 60

print(a)

/\*-----------------------------------------\*/

// FUNCOES

/\*

 func nomeDaFuncao (parametro: tipoDoParametro) -> tipoDeRetorno {

    Corpo da funcao

    return Retorno da funcao

 }

 \*/

// Exemplo

func minhaFuncao(num: Int) -> Int {

    return 0

}

/\*-----------------------------------------\*/

// EXERCICIO

// Criar um operador que receba dois parametros (exemplo 6 e 60, da mega sena)

// Na funcao, gerar tantos numeros aleatorios, entre 1 e o segundo parametro

// Para que assim, seja retornado os numeros da mega sena.

infix operator <->

func <-> (lhs: Int, rhs: Int) -> [Int]{

    var x:[Int] = []

    var i = 0

    while i < lhs{

        x.append(Int(arc4random\_uniform(UInt32(rhs)) + 1))

        i += 1

    }

    return x

}