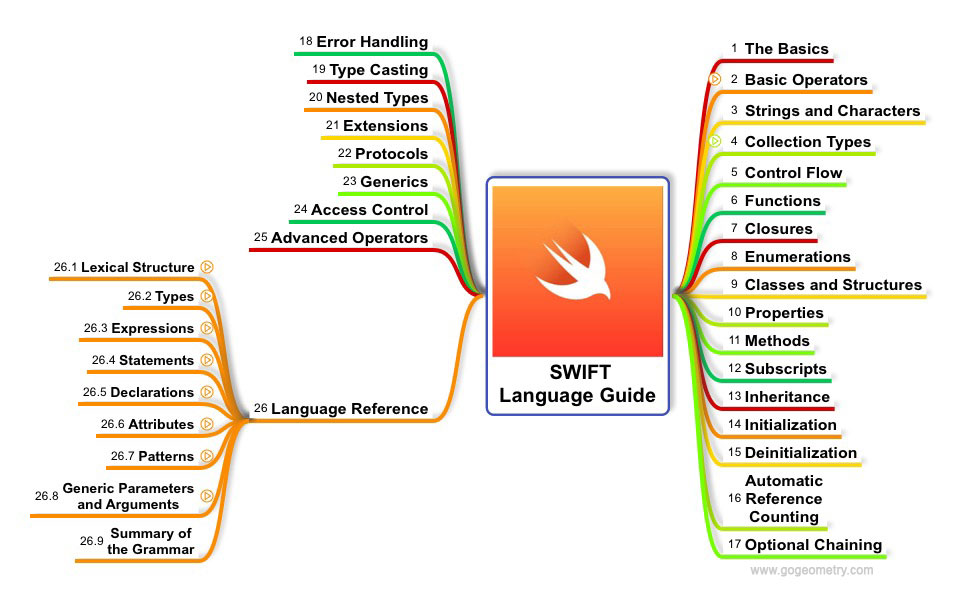
Swift

# Criar Projeto Playground

É um ambiente de test, onde os variaveis e UI são mostrados na barra lateral e console para mostrar print();

# Passagem de argumentos por valores vs por referência (inout)

var num1: Int = 1

var char1: Character = "a"

//Passagem por Valor

func changeNumber(var num: Int) {

num = 2

print(num) // 2

print(num1) // 1

}

changeNumber(num1)

//Passagem por Referência

func changeChar(char: inout Character) {

char = "b"

print(char) // b

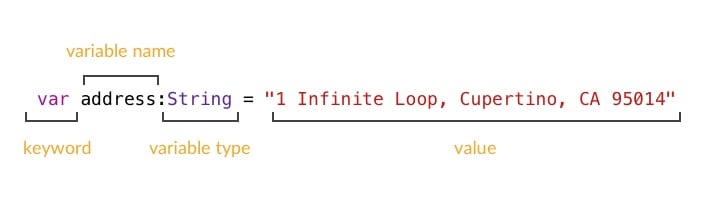
print(char1) // b

}

changeChar(&char1)

# Variáveis

Não é necessária a definição do tipo da variável na linguagem Swift, pois o compilador identifica automaticamente (Type Inference). Ao inicializar a variavel na primeira vez o tipo é definido e não pode ser alterado.



var score:Int = 0

var score = 0

variável nil não é uma variável nula, é a ausença da atribuição.

OBS: O Tipo default para números decimais é Double.

## Converter double para float

Float(numero)

## Converter String para int

Int (str)

## Converter int para String

String(numero)

## Var e Let

Ambos são referências de objetos de tipo qualquer, mas let é referência de constante. Ao inicializar um constante let não pode ser alterada a referência; enquando o var é referência de variável, onde pode ser alterada qualquer momento.

## Variaveis Opcionais e não Opcionais

Por default as variáveis são não opcionais, isto significa que as variáveis precisam serem inicializas. Enquanto as variáveis Opcionais podem ou não serem inicializadas.

var str : String? OU

var str : Optional<String>

String? É um objeto opcional que guarda um objeto String (Wrapper)

## Unwrappe Optional Variable

Nil Coalescing Operator(Melhor Jeito):

var name : String?

Print(name ?? “sem nome”)

Unwrappe forçada (Não segura por que pode retornar nil): variavel!

Optional Binding(Mais Comum):

if let numero = a{ //colocando a ‘a’ um let para verificar se é nil

Print (numero)

}

Guard Let(precisa ser dentro de função):

fun getAB(){

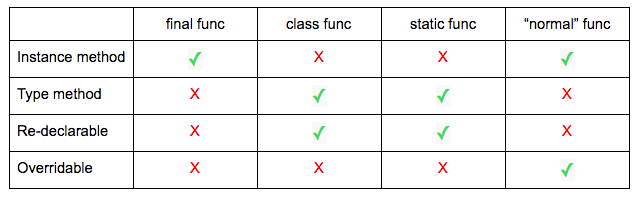
Guard let test = a else {return}

Guard let test = a, let test2 = b else {return}

Print(a+b)

}

# Funções



Func nomeFunção (NomeExterno(Opcional) NomeInterno : Tipo ){...}

Nome Externo: nome/label que é usado na passagem de argumentos (\_ para não colocar label argumento)

Nome Interno: é o nome do argumento utilizado dentro da função

func testFun1(){

print("")

}

func testFun2() -> Void{

print("")

}

func testFun3() -> String{

print("")

return ""

}

## Funções Sem Label de argumentos\_

Por default as funções utilizam o nome de seus argumentos para serem utilizados como label. Isto é usado na chamada da função definido valores de cada label de paramêtro,

Exemplo: func funcao (param : String){...}

Camada: funcao (param : “ola”) //param é label

Por isso as funções podem não utilizar labels de argumentos adicionando \_ antes do argumento. Exemplo:

func funcao ( \_ param : String){...}

Chamada: funcao (“ola”)

## class func

Um função class func é equivalente ao static func

## Subfunções

A linguagem Swift permite criar funções dentro de outras funções

func returnFifteen() -> Int {

var y = 10

func add() {

y += 5

}

add()

return y

}

## first-class Functions

As funções first-class podem serem referênciados numa variavél para serem utilizados, Exemplo:

var increment = makeIncrementer()

increment(7)

## Funções como Argumento

As funções pode ser passados como argumento de outras funções, Exemplo:

func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {

for item in list {

if condition(item) {

return true

}

}

return false

}

func lessThanTen(number: Int) -> Bool {

return number < 10

}

var numbers = [20, 19, 7, 12]

hasAnyMatches(list: numbers, condition: lessThanTen)

# Strings

Verificação de igualdade pode ser com ==

## String Multiplas Linhas

Usando: “””

Test

“””

## Concatenação Strings

var str1 = “ola”

var str2 = “mundo”

var str3 = var1 + var2

var str4 = “ola /(str2)”

str1.append(“ ” + “mundo”)

## Capitalize

var name = “john”

name = name.capitalized

## LowerCase

var name = “JOHN”

name = name.lowercased()

## Contains & Replace

var name = “John Carlos”

if name.contains(“Carlos”){

name = name.replaceOccurrences(“Carlos”,””)

)

# IF ELSE

If number = 1 {

...

} else {

}

# Iternário

Str = “ola” ? sim : não

# Class vs Structure

Classe é reference type, passando como parametro passa a referência do objeto (Objeto original é modificado).

Struct é value type, passando como parametro uma struct é uma cópia e não a referência (Objeto original Não é modificado)

# Classes

**A passagem de objetos como parametros é por referência**

class TestClass{

internal var test : String? //Default internal

private var test1: String?

public var test2 : String?

//Computed Properity

var test3 : String {

get{

return test3 + " test"

}

set{

self.test3 = newValue

if newValue == ""{

test3 = "Empty"

}

}

}

//Variable Observer

var test4 : String{

willSet{

print("Variavel vai mudar para: " + newValue)

}

didSet{

print("Variavel alterado, valor anterior: " + oldValue)

}

}

init() { //Construtor

test = ""

test1 = ""

test2 = ""

// test3 = "" //Não pode inicializar no construtor, pois é criada a variavel computada na segunda fase do compildor

test4 = ""

}

init (test : String, test1 : String, test2 : String){

self.test = test

self.test1 = test1

self.test2 = test2

// test3 = "" //Não pode inicializar no construtor, pois é criada a variavel computada na segunda fase do compildor

self.test4 = ""

}

/\*override init(){ //Sobre escrever o construtor do pai

}\*/

func getTest() -> String {

return test ?? ""

}

func printInfo() -> Void {

print("Info")

}

}

let test6 = TestClass()

test6.printInfo()

let test7 = TestClass(test : "", test1:"", test2: "")

## Construtor e Destrutor

Construtor: init()

Destrutor: deinit() – Usado para fazer Clean-up quando a instância está sendo destruída, Exemplo fehcar conexões, delocar objetos da memoria...etc

# Struct

struct TestStruct{

var test1 : String

var test2 = ""

var test3 : String{ //Computed property

get{

return test3

}

set{

test3 = newValue + ""

}

}

/\*func setTest1(test1 : String){ //Não Pode, no struct os variaveis são constantes,

self.test1 = test1

}\*/

mutating func setTest1(test1 : String){

self.test1 = test1

}

}

var test8 = TestStruct(test1: "", test2: "")

test8.test1 = ""

/\*let test9 = TestStruct(test1: "", test2: "")

test9.test1 = ""\*/ //Não pode, alterar propriedade de uma estrutura constante

## Mutating Functions

São funções utilizadads dentro de struct para modificar propriedades. Dentro de protocolos/classes não é necessário a utilização da palavra chave mutanting por que sempre são mutantes (podem alterar valores de propriedades).

# Comparação Por Idêntidade

var test10 = TestClass(test: "", test1: "", test2: "")

var test11 = TestClass(test: "", test1: "", test2: "")

print (test10 === test11) //False

# Extensions

- Extensions podem extender(adicionar funcionalidades) qualquer classe, protocol, struct e enumerations

- Extensions podem adicionar propridades, funções, criar instâncias, sobrecarregar funções e construtores, confirmar um tipo para um protocolo.

## Funções Mutativas

- Extensions permitem criar funções mutativas: funções que alteram as proprias instâncias.

self = ...

Criar Extension:

1. Criar Classe nomeClasseExt
2. Extender

# Sobrecarga de Operadores

Para sobrecarregar operadores basta criar uma função (local/global) com o mesmo nome do operador, exemplo

func + (intValue: Int, floatValue: Float) -> Float{

return Float(intValue) + floatValue

}

# Herança

Só pode herder de uma classe ou de varias interfaces (Protolos)

class Hero{

var name : String

var age : Int

var damage : Int

init(name: String, age : Int, damage : Int){

self.name = name

self.age = age

self.damage = damage

}

func attack(){

print("Attack!")

}

}

class Warrior : Hero{

var weapon : String

init(name: String, age : Int, damage : Int, weapon : String){

self.weapon = weapon //PRIMEIRO AS VARIAVEIS LOCAIS TEM QUE SER INICIALIZADOS

super.init(name:name, age:age, damage:damage) //Inicializar o construtor da superclasse

}

override func attack() { //Sobre Escrever o metodo da super classe

print("\(self.weapon) Attack!!!")

}

}

# Protocolos & Extensions

- Protolocos são Interfaces(metodos sem corpos)/Blueprints/Contratos contendo variaveis e assinaturas que serão implementadas. Os protocolos são blueprint de metodos para definir a funcionalidade de uma classe.

**- Os protocolos só aceitam propriedades computadas.**

**- Os metodos dos protocolos que alteram propriedades tem que ser mutáveis**

- Extensions são extensões de protocolos e classes, onde pode extender até tipos adicionando novo metodos/variáveis...

- Protocolos podem herder outros Protocolos

- Extension tabém podem herder de outro protocolos. **A classe que implementa o subprotocol tem que conformar os 2 protocols**

protocol Person{ //INTERFACE/PROTOCOL

var name:String {get set} //Na interface as propriedades precisam ter getter e setter

var age: Int {get set}

func run()

func walk()

func sleep()

}

//Extension podem extender qualquer tipo, mesmo aqueles que não tem código fonte.

extension Person { //Extension é a extensão do protocol, usada para definir metodos padrões do protocol

var isSick: Bool { return age > 100 }

func run(){

print("Running...")

}

}

class Painter : Person{ //A classe que implementa o protocol tem que implementar todos os variaveis e funções, menos aqueles que já foram implementadas na extension

var name:String

var age: Int

func walk(){}

func sleep(){}

/\*Podemos sobreescrever o metodo extendido

func run() {

print("Motasem is Running")

}\*/

init(name:String, age:Int) {

self.name = name

self.age = age

run()

}

}

var person1 = Painter(name:"Mota", age:23)

## Optional Method

Os Optional method nos protocols deixam a implemenção opcional, então eles não serão requisitos para satisfazer um protocolo.

# Delegate & Protocol

Delegate permitir que outra instância gerenciar a instância atual (executar métodos). Para executar um metodo precisa ter um protocol implementado.

1. Criar um procolo com o metodo com parametros que será chamado

Class LoginCompleteDelegate{

onUserCreated(username : String)

}

1. Implementar o protocol e seus metodos no ViewController que vai executar a ação

class HomeVC : UIViewController, LoginCompleteDelegate{

onUserCreated(username : String){...}

}

1. No ViewController que chamará o delegate criar variavel opcional(não inicializado no construtor) delegate:

var delegate : LoginCompleleteDelegate?

1. Definir o momento que será chamado o metodo

Delegate.onUserUpdate(“John”)

1. No ViewController que vai executar ação inicializar a variavel delegate quando vai trocar para ViewController que vai chamar o delegate

//Ao trocar de um ViewController para outro, se o ViewController destinário foi LoginVC então inicializa o delegate

override func prepare(for segue: UIStoryboardSegue, sender: Any?)

if(let destination = segue.destination as? LoginVC){

destination.delegate = self

}

}

# Arrays

Quando o tipo é definido implicitamente é do tipo [Any], então pode armazenar varios tipo em um único array, exemplo: [1, 2, “oi”, 5]

var array = [“”]

var array = [String]()

var array : [String] = []

var array : Array<String> = []

# Arrays (Dinâmicos)

var array1 = ["Olá", "Mundo"]

var emptyArray = [String]()

var array2 : [String] = ["Olá Mundo"]

array1.append("Olá") //Coloca na ultima posição

array1.insert("Mundo", at:1)//Coloca na segunda posição

array1.remove(at: 0) //Excluir o valor do aray

array1[0] = "JJ" //Atualizar o valor do array

# Array (Estáticos & Read Only)

let array3 = ["Olá", "Mundo"]

//array3.append("Olá") //Erro

//array3.insert("Mundo", at:1)//Erro

//array3.remove(at: 0) //Erro

//array3[0] = "JJ" //Erro

# Dicionários

var dic = ["ola":1, "mundo":2]

var dic2:Dictionary<String, Int> = ["ola":1, "mundo":2]

dic2["sd"] //NIL

dic2["sd"] = 4 //Adicionar novo item

dic2["ola"] = 3 //Atualizar um novo item

dic2.removeValue(forKey: "ola") //Remover um item

dic2

for key in dic.keys{...}

for value in dic.values{...}

for (key, value) in dic {...}

# Enumeradores

- Os Enumeradores pode conter enumerados, funções e computed properities e static properites. (Não pode conter stored properities)

- Por default, os enumeradores começam com valor 0

enum Tipo1 { //Enumerador Comum

case a ,b ,c //Pode ser na mesma linha ou linhas Separadas

case d

}

enum Tipo : Int{ //Enumerador com identificadores int

case a = 1 //Definir inicio da contagem do identificador

case b

case c

case d

}

var test4 = Tipo.a

print(Tipo.(rawValue : 0)!) //Obter o primeiro Valor

print(test4.rawValue)

print(Tipo.a) //Mesmo do Anterior

enum Tipo2 { //Enumerador com parametros

case a (Int, Int, String) // Enumerador com multiplos parametros

case d(Int) //Enumerador com 1 parametro

case l //Enumerador sem parametro

}

var test = Tipo2.a(1,2,”test”)

enum Tipo3 : String{ //Enumerador com identificadores String

case a = “Ola” //Definir inicio da contagem do identificador

case b

case c

case d

}

Var test2 = Tipo3.a

print(test2.rawValue) //Ola

var enum1 = Tipo.a

var enum2 : Tipo = .a //O Compilador já resolve pelo o tipo

if enum1 == .a {} //O Compilador já resolve pelo o tipo

if enum1 == Tipo.a{}

## Enumeradores Costumizados

- Enumeradores podem ter funções para descrever itens

enum Suit {

case spades, hearts

func simpleDescription() -> String {

switch self {

case .spades:

return "spades"

case .hearts:

return "hearts"

}

}

}

## Enumeradores com Parametros

enum ServerResponse {

case result(String, String)

case failure(String)

}

let success = ServerResponse.result("6:00 am", "8:09 pm")

switch success {

case let .result(sunrise, sunset):

print("Sunrise is at \(sunrise) and sunset is at \(sunset).")

case let .failure(message):

print("Failure... \(message)")

}

# Loops

var dic4:Dictionary<String, Int> = ["ola":1, "mundo":2]

var array5 : [String] = ["a", "b", "c", "d", "e"]

for (key, value) in dic4{

print("\(key) \(value)")

}

for strValue in array5{

print(strValue)

}

for i in 0...5{ //Contar de 0 até 5

// print (i)

}

for \_ in 1...3 { //Não liga para valor

// Do something three times.

}

for i in 0..<5{ //Contar de 0 até 4

print (i)

}

for i in stride(from: 0, to: 7, by: 2){ //Contar de 0 até 6 (ultimo não conta)

print(i)

}

var counter1 = 0

while counter1 < 5 {

counter1 += 1;

print(counter1);

}

var counter1 = 0

repeate{

counter1 += 1;

print(counter1);

} while (counter1 < 5)

# Iteradores de Arrays/Dictionaries/Collections

Notem que os iteradores abaixo recebem como paramêtro um clouser. Estes iteradores realmente são loops que iteraram cada item

## Map

Iterar cada item do array

var cityName = [“santos”, “são paulo”, “são vicente”]

let capitalizedCityNames = cityNames.map{ $0.capitalized } //$0 é o item do array

## FlatMap

Iterar collections/dictionaries e aplicar flat (converter em array)

var collections = [[1, 2, 3], [1, 4, 7], [44,99, 44]]

let singleCollection = collections.flatMap{$0} //$0 = array (poder ser iterado .map)

## CompactMap

Remover todos os Nil do array

var array : [Int?] = [1, 2, 3, nil, 4, 7, 44, nil, 44]

var array2 = array.compactMap{$0}

## Filter

Definir condição booleana para filtrar(remover) dados dos arrays

var array : [Int] = [1, 2, 3, 4, 7, 44, 43]

var filteredArray = array.filter{$0%2==0} //$0 é cada item, se for impar então a condição é falsa e não será adicionado no array

var cityName = [“santos”, “são paulo”, “são vicente”]

var filteredCities = cityName.filter{$0.first() == “s”} //só pegar strings que começam com s (condição verdadeira)

## Reduce

Aplicar uma operação aritimetica/lógica em todo array Númerico/Texto para obter 1 resultado só.

print("Hello World")

var test:[Int] = [3,4,5,6]

print (test.reduce(0, +)) //Somar todos os itens do array + 0

print (test.reduce(1, +)) //Somar todos os itens do array + 1

print(test.reduce(1,\*)) //Multiplicar todos os itens \* 1 = 1 \*3\*4\*5\*6

print(test.reduce(1,/)) //Dividir todos os itens \* 1 = 1 /3/4/5/6

print (test.reduce(1, &)) //Lógica And 1 & 3 & 4 & 5 & 6

print (test.reduce(1, |)) //Lógica Ou 1 | 3 | 4 | 5 | 6

var testStr:[String] = ["Olá ", "Mundo ", "lindo"]

print(testStr.reduce("test: ", +)) // juntar todos os strings = "test: ola mundo lindo"

## Zip

Juntar 2 arrays em 1 Array de tuple

var name:[String] = ["Motasem ", "John", "Test"]

var lastName:[String] = ["El Malat ", "Sina", "2"]

var test:Array<(String, String)> = Array(zip(name,lastName)) //[("Motasem ", "El Malat "), ("John", "Sina"), ("Test", "2")]

# Tuplas

Tuplas são listas ordenada, utilizados nas funções para retornar mutliplos valores

func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {

var min = 0

var max = 2

var sum = 3

return (min, max, sum)

}

let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])

print(statistics.sum)

Exemplo 2

let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]

for (x, 0) in points {

/\* ... \*/

}

# Error Handling

enum PrinterError: Error {

case outOfPaper

case noToner

case onFire

}

func send(job: Int, toPrinter printerName: String) throws -> String {

if printerName == "Never Has Toner" {

throw PrinterError.noToner

}

return "Job sent"

}

do {

let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")

print(printerResponse)

} catch {

print(error)

}

do {

let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")

print(printerResponse)

} catch PrinterError.onFire {

print("I'll just put this over here, with the rest of the fire.")

} catch let printerError as PrinterError {

print("Printer error: \(printerError).")

} catch {

print(error)

}

## Error to Optional

Com a utilização de try? Antes do metodo, ele retorna valor ou nil se throw exception

let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")

## Do Catch

struct RuntimeError: Error {

let message: String

init(message: String) {

self.message = message

}

public var localizedDescription: String {

return message

}

}

func testFunc1() throws -> String {

throw RuntimeError(message:"404")

return ""

}

do {

try testFunc1()

} catch {

print("Exception!!!! \(error)")

}

Switch Case

O Switch case não precisam de break

switch enum1{

case .a:

print()

fallthrough //Verifica as próximas condições mesmo se está condição for verdadeira

case .b , .c: //Lógica Ou

print()

default:

break //Não faz nada

}

switch enum1{

case Tipo.a:

print()

fallthrough //Verifica as próximas condições mesmo se está condição for verdadeira

case Tipo.b , Tipo.c: //Lógica Ou

print()

default:

break //Não faz nada

}

## Defer

Defer é um bloco de código colocado no inicio do metodo para ser executado bem no final do metodo, antes de retornar valor. Este bloco é executado mesmo se a função throw exception

func fridgeContains(\_ food: String) throws -> Bool {

defer{

print (“closing...”) //Só vai mostrar closing depois de opening e de tudo

}

print (“opening...”)

throw Error

return true

}

# Modificadores de Acesso (Encapsulamento)

## open

Acesso de qualquer lugar e qualquer modulo (bibliotecas, targets..etc)

## public

Acesso de qualquer lugar, menos dentro de subclasse(Hernaça) de outros módulos(bibliotecas, targets..etc)

## internal (Padrão Xcode)

acesso qualquer somente dentro do modulo. Outros modulos(targets, biliotecas) não conseguem acessar.

## private

Acesso somente dentro da classe

## fileprivate

Permite o acesso somente dentro do arquivo .swift de origem definindo.

## final

Usado para evitar um método, propriedade, subscript, classe seja sobre escrito, ou uma classe seja herdada.

## variavel computada somente leitura

var invoice : double{

return invoiceValue; //Outra Variável que estamos protegendo

}

## Encapsulamento set e get Separado

public private(set) var variable = 1 //Variavel public para leitura e private para escrita

# Generics

Generics é um tipo qualquer não definido em tempo de compilação e só é definido em tempo de execução. Usado em Metodos genericos, Tipo de variável, Tipos de retornos...etc. exemplo:

func funcao <T> (param1 : T, param2 : T){...}

funcao(param1: 1, param2: 2)

var variavel : [T]

func remove() -> T {

return items.removeLast()

}

## Protocolos de Genéricos

Strideable: Type de 1 dimensão só

### Numeric

Um variavel generica númerica. Pode fazer operações + - \* /, Não pode ser array (1 dimensão só)

func testFunc <T : Numeric> (\_ number : T) -> T{

return number + 1 \* 3

}

### SignedNumeric

Um variavel generica númerica e com sinal. Pode fazer operações + - \* /, Não pode ser array (1 dimensão só)

### Comparable

Qualquer variavel que possa comparar com == != de texto ou númerica.

# Subscripts

Os subscript são funções getter e setter de um objetos, podem ser chamados com []. Exemplo objeto[1]

struct TestClass {

var name : String

var number : Int

//Primeiro Jeito

/\* subscript(index: Int) -> Int {

return number \* index

}\*/

//Segundo

subscript(index: Int) -> String{

get{

var nameCharacters = Array(name)

return String(nameCharacters[index])

}

set{

var nameCharacters = Array(name)

nameCharacters[index] = Character(newValue)

self.name = String(nameCharacters)

}

}

}

var a = TestClass(name: "Classe", number: 123)

print(a[2])

a[2] = "8"

print(a)

# Anotation

## @Available

classe / Metodo disponível para certas plataformas, certas versões de sistemas operacionais.

@available(iOS 10.0, macOS 10.12, \*)

class MyClass {

// class definition

}

## @Deprecated

Função/classe discontinuado

deprecated: version number

## @Renamed

Indicar que a classe/função mudou de nome

renamed: new name

# Casting

## Verificação de Tipo

Is : verificar se um objeto é de uma instância

if object is Class {

....

}

## DownCasting

Existem 2 tipos de casting, forçada(as!) e condicional(as?)

**Casting forçada (as!):** é usada com verificação do tipo para fazer casting. **IMPORTANTE se casting falir, vai dar Crash**

if object is Class {

var object2 = object as! Class

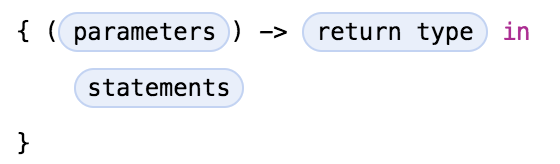
}

**Casting Condicional (as?):** Sempre retorna um variável opcional do tipo que está tentando **converter se é possível ou nil se não.**

var object2 = object as? Class

# Clousers

Clousers são blocos de código executavel que podem serem passados como argumentos, eles são a otimização das funções já preparados para receber como paramêtro, usados nos callbacks



**Tipo da Função: (Int, Int) -> Int**

func sum(a : Int, b: Int) -> Int{

return a + b

}

**//Tipo da Função: (Int, Int) -> Int**

func sub(a : Int, b: Int) -> Int{

return a - b

}

func doMath (a: Int, b: Int, operation: (Int, Int)->Int) -> Int{

return operation(a,b)

}

print(doMath(a: 10, b: 20, operation: sum(a:b:))) //passar função como parametro

**//Fazer como Clouser**

doMath(a: 10, b: 20 ,operation: {(a: Int, b: Int) -> Int in return a + b} )

**//Otimização 1: Inferring Type (otimir o Tipo)**

doMath(a: 10, b: 20 ,operation: { a, b in return a + b} )

**//Otimização 2: Remover a palavra return**

doMath(a: 10, b: 20 ,operation: { a, b in a + b} )

**//Otimização 3: Encurtar os nomes dos parametros**

doMath(a: 10, b: 20 ,operation: { $0 + $1} ) //$0 = Primeiro parametro, $1 = Segundo Parametro

**//Otimização 4: Trailing Clouser: Junção dos corpos dos metodos**

doMath(a: 10, b: 20){$0 + $1}

**//Exemplo: String Array Sort**

let names = ["John", "Luke", "Lucas", "Gabriel"]

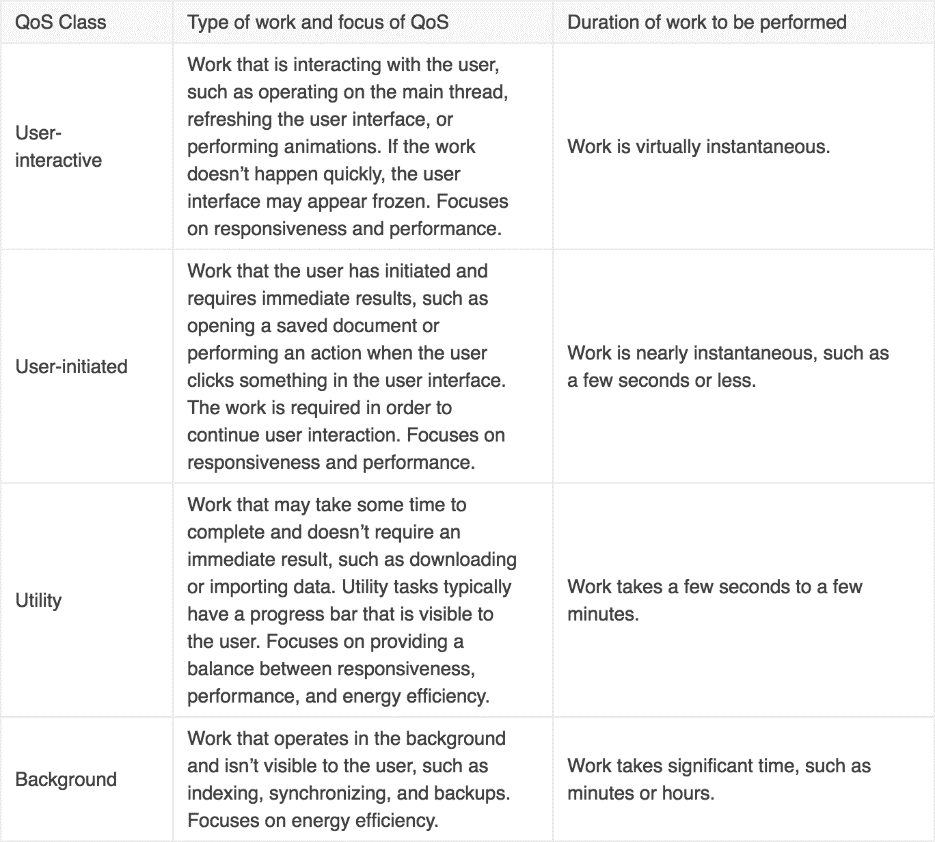
let sorted = names.sorted{$0 > $1} //Metodo com clouser para definir a politica para ordenar, onde a função iterara a cada 2 itens do array comparando usando o clouser

print(sorted)

# MultiThread

Prioridade de cada Task

DispatchQueue.global(qos: .userInitiated).



## Run on UI Thread

DispatchQueue.main.async() {

// your UI update code

}

**OU**

dispatch\_async(dispatch\_get\_main\_queue()) {

label.stringValue = result

}

## Run on UI Thread After X Seconds

let delay = DispatchTime.now() + .seconds(60) DispatchQueue.main.asyncAfter(deadline: delay) {

// Update UI!

}

## Run on Background

DispatchQueue.global(qos: .userInitiated).async {

// Download file or perform expensive task

DispatchQueue.main.async {

// Update the UI

}

}

**OU**

let queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_HIGH, 0)

dispatch\_async(queue) {

...

}

## Run Background Loop

DispatchQueue.global(qos: .background).async {

for i in 0 ..< self.fetchResult.count {

GenericCoreData.saveEntity("Details")

}

}

## DispatchGroup MultiTask

DispatchGroup permite que roda varias tarefas em background (concurrently).

**- dispatchGroup.notify(queue: .main):** Único metodo que roda no main thread, é usado para notificar que todas as tarefas já finalizaram, então podemos esconder loading progress spinner, retornar resulto de todas as tarefas que rodaram...etc.

- Antes de cada tarefa (queue), tem que executar **group.enter(),** rodar a tarefa em uma clouser e no final da clouser rodar **group.leave() SOMENTE QUANDO A TAREFA ACABA.**

**- group.wait()**: Usado para travar a thread atual até as Tarefas/queues antriores deste comando acabam (não recomendado o uso em alguns casos)

**DEADLOCK WARNING: you should never call sync on the main queue, because it'll cause a deadlock and a crash**

let group = DispatchGroup()

let queueImage = DispatchQueue(label: "com.image")

let queueVideo = DispatchQueue(label: "com.video")

group.enter()

queueImage.async(group: group) {

sleep(2)

print("image")

group.leave()

}

group.enter()

queueVideo.async(group: group) {

sleep(3)

print("video")

group.leave()

}

group.notify(queue: .main) {

print("all finished.")

}

Exemplo Com Wait

func myFunction() -> Int? {

var a: Int?

let group = DispatchGroup()

group.enter()

// avoid deadlocks by not using .main queue here

DispatchQueue.global(attributes: .qosDefault).async {

a = 1

group.leave()

}

// wait ...

group.wait()

// ... and return as soon as "a" has a value

return a

}

Mais Um Exemplo

import Foundation

import PlaygroundSupport

let higherPriority = DispatchQueue.global(qos: .userInitiated)

let lowerPriority = DispatchQueue.global(qos: .utility)

func asyncPrint(queue: DispatchQueue, symbol: String) {

queue.async {

for i in 0...10 {

print(symbol, i)

}

}

}

asyncPrint(queue: higherPriority, symbol: "🔴")

asyncPrint(queue: lowerPriority, symbol: "🔵")

PlaygroundPage.current.needsIndefiniteExecution = true

## Lock Resource Using .barrier

Lock resource é necessário em multithread para evitar Race Conditions, assim o recurso seja um variavel ou objeto é travado para thread specifico até acabar de usar e liberar para outros threads. IMPORTANTE: exagerar o uso de locks pode levar alguns thread em Starvation (nunca ganhar acesso ao recurso devido alta perioridades de outros threads).

// .barrier flag ensures that within the queue all reading is done

// before the below writing is performed and

// pending readings start after below writing is performed

concurrentQueue.async(flags: .barrier) {

self.dictionary[key] = value

}

## Cancelar Tarefa DispatchWorkItem

Jeito Não perfeito: porque ele tem que esperar até chegar na linha de verificação booleana

//JEITO RUIMMMMMMMMMMMMMMM

DispatchQueue(label: "Data Refresh Queue").async(after: .now() + 2) {

// shouldContinueDataRefresh() retorna booleano

guard shouldContinueDataRefresh() else { return }

// Refresh data

}

Por isso é Usado DispatchWorkItem que consegue cancelar, pausar, resumir...etc

**let** workItem **=** **DispatchWorkItem** { **doExcitingThings**() }

**DispatchQueue.**main**.asyncAfter**(deadline: **.now**() **+** 10, execute: workItem)

workItem**.cancel**()

## Pausar e Resumir Tarefa

**let** queue **=** **DispatchQueue**(label: "Network Queue")

queue**.suspend**()

queue**.resume**()

## DispatchQueue e DispatchWorkItem

Por isso é Usado DispatchWorkItem que consegue cancelar, pausar, resumir...etc

let group = DispatchGroup()

let queue = DispatchQueue(label: "com.theswiftdev.queues.serial")

let workItem = DispatchWorkItem {

print("start")

sleep(1)

print("end")

}

queue.async(group: group) {

print("group start")

sleep(2)

print("group end")

}

DispatchQueue.global().async(group: group, execute: workItem)

// you can block your current queue and wait until the group is ready

// a better way is to use a notification block instead of blocking

//group.wait(timeout: .now() + .seconds(3))

//print("done")

group.notify(queue: .main) {

print("done")

}

# Task

# Timer(Pasta Processamento)

# Thread

# Background Fetch (Serviço Background)

Background Fetch é um serviço background que ser despara automáticamente, ele pega uma padrão de acesso ao aplicativo e se despara antes que o usuário abrir o app.

- Da para definir o tempo minimo que precisa para executar o background Fetch. (não é executado depois do tempo minimo imediatamente).

- Tem opção para desabilitar/habilitar o background fetch

# Notifications