Swift

Laisa Martins
Lucas Lopes
Nathânia Queiroz

Introdução

História

- Uma linguagem recente, introduzida ao mundo em 2014
- Criada pela Apple, veio da necessidade de adequar o desenvolvimento para iOS e OS X aos padrões, tendências e necessidades atuais
- Objective-C era a linguagem utilizada para desenvolver para iOS e OS X, mas ela ainda era uma camada “fina” que cobria a linguagem C. Em comparação aos paradigmas atuais da programação, Objective-C, apesar de moderna e inovadora na época da sua criação, apresenta algumas lacunas
- Swift nasce prometendo substituir Objective-C oferecendo mais eficiência, clareza para facilitar o aprendizado, portabilidade, segurança…

História

- No seu lançamento em 2014, a Apple prometeu fazer da linguagem uma plataforma de código aberto em breve
- A promessa foi cumprida em 2015. Além do código, a empresa lançou um blog sobre a linguagem e um manual completo disponível online e para download
- Swift se mostra uma das linguagens de mais rápida disseminação da história
- Swift se manteve no top 4 linguagens de programação mais amadas na pesquisa anual do StackOverflow desde sua criação

Visão Geral e Objetivos

- A criação de Swift busca trazer modernidade para a programação para iOS e OS X
- A Apple espera que Swift se torne uma das principais linguagens de programação nos próximos 20 anos
- Swift se propõe ser uma linguagem mais fácil, segura e completa. Ou seja: uma linguagem que atraia mais público e atenda as necessidades do mundo corporativo
- Swift é uma linguagem multiparadigma: OO, imperativa, funcional
- Swift opera sob a licença Apache 2.0 desde a versão 2.2
- A linguagem é compilada

Visão Geral e Objetivos

- Linguagem possui clareza na sintaxe; uma escrita limpa, que facilita o aprendizado
- Swift é uma linguagem fortemente tipada e com inferência de tipos
- “Hello World!” em C:

```
#include <stdio.h>
int main() {
        printf(“Hello World!\n”);
        return 0;
}
```

“Hello World!” em Swift:

```
print(“Hello World!”)
```

Começando a Programar

- Possui uma plataforma online disponibilizada pela IBM
  - https://swift.sandbox.bluemix.net/
- A plataforma será descontinuada em janeiro de 2018 devido aos avanços da computação em nuvem, agora é mais fácil experimentar diretamente nesses ambientes
- A Apple oferece binários para OS X e Linux que podem compilar código para iOS, OS X, watchOS, tvOS e Linux
- Podemos usar a interface interativa pelo terminal digitando ***swift,*** ou podemos criar um arquivo com extensão .swift e executar no terminal ***swift arquivo.swift***

Curiosidades

Um algoritmo comum de busca, por exemplo, obtém o resultado muito mais rápido com Swift.

Até **2,6x** mais rápido que Objective-C

Até **8,4x** mais rápido que Python 2.7

Amarrações

Identificadores

- Podem começar com: caracteres maiúsculos e minúsculos de A – Z, underscore (_), caracteres alfanuméricos Unicode não combinados do Plano Básico
- Identificadores válidos: Ola, estou$, _testando
- Identificadores inválidos: $isso, ?nao, 1pode
- Swift é case sensitive!

Identificadores

- Dentro de uma Closure sem nomes explícitos de parâmetros, estes são nomeados por default como $0, $1, etc. Portanto dentro do escopo da Closure, é permitido iniciar com o caractere $
- É possível usar palavras reservadas como identificadores, usando o caractere ` antes e depois.

Exemplo:

class = palavra reservada, identificador inválido

`class` = identificador válido

-> Note que o caractere ` não pertence ao identificador, portanto X e `X` tem o mesmo significado.

Palavras-chave

Usadas em declarações:

associatedtype	class	deinit	enum
fileprivate	func	import	init
inout	internal	let	open
operator	private	protocol	public
scatic	struct	subscript	typealias
var			

Palavras-chave

Usadas em statements:

break	case	continue	default
defer	do	if	else
fallthrough	for	while	repeat
switch	return	where	in
guard			

Palavras-chave

Usadas em tipos e expressões:

as	Any	try	catch
throw	throws	rethrows	nil
super	self	Self	is
true	false		

Palavras-chave

Iniciadas com símbolos:

#avaliable	#colorLiteral	#if	#else
#elseif	#endif	#file	#fileLiteral
#function	#column	#imageLiteral	#line
#selector	#sourceLocation	_	

Palavras-chave

Reservadas em contextos particulares:

associativity	convenience	dynamic	didSet
final	get	infix	indirect
lazy	left	mutating	none
nonmutating	optional	override	postfix
precedence	prefix	Protocol	required
right	set	Type	unowed
weak	willSet		

Valores e Tipos de dados

Sistemas e Tipos

- Possui Tipagem Estática.
- Fortemente tipada.
- Inferência de tipo.
- Possui Case-sensitive.
- Aceita Emoticons.
- Suporta Unicode, os nomes podem ter acento.

Variáveis e Constantes

- variáveis são representadas por ***var*** e constantes por ***let***
- variáveis são mutáveis, constantes não

```
var qtdVariavel = 1 // Esse valor pode ser modificado
qtdVariavel = 2

let qtdConstante = 1
qtdConstante = 2 // Erro de compilação
```

Sistemas e Tipos

```
var sea = "abc"
var Sea = true
print(sea, Sea)
var chão = “de terra”
//aã2 = 1010   Erro de compilação
let sparklingHeart = "\u{1F496}"
// sparklingHeart = "\u{2665}"
print(sparklingHeart)
let □ = "👦□👩👶"
print(□)
var str¨12 = 1010.2
str¨12 = 1
print(str¨12)
```

```
abc true
💖
👦□👩👶
1.0
```

Sistemas e Tipos

- Os nomes constantes e variáveis não podem conter caracteres de espaço em branco, símbolos matemáticos, setas, pontos de código Unicode de uso privado (ou inválidos), ou caracteres de desenho de linha e caixa(╟). Nem podem começar com um número, embora os números possam ser incluídos em outro lugar dentro do nome.

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Principais Tipos de Dados

- Inteiros (Int)
- Ponto Flutuante (Double e Float)
- Carácter (Character)
- String
- Lógico (Bool)
- Nulo (nil/Optionals)
- Coleções de Tipos(Arrays, Tuplas, Conjuntos, Dicionários, Enumerado)

Tipo Inteiro

- A linguagem provê inteiros com e sem sinal, nas formas de 8, 16, 32 e 64 bits.
- Podemos acessar os valores mínimos e máximos de cada tipo inteiro com ***min*** e ***max***

```
let minValue = UInt8.min  // minValue equivale a 0, e seu tipo é UInt8
let maxValue = UInt8.max  // maxValue equivale a 255, e seu tipo é UInt8
let valor:Int32 = 10 // seu tipo é Int32
```

Tipo Inteiro

- Swift possui um tipo padrão que varia o tamanho dinamicamente de acordo com a plataforma atual
- Plataformas 32 bits, ***Int*** possui o mesmo tamanho que ***Int32***, assim como ***UInt*** possui o mesmo que ***UInt32***.
- Idem para plataformas 64 bits.

Ponto Flutuante

- *Double* representa um número de ponto flutuante de 64 bits, e *Float* representa um de 32 bits.
- Por padrão, na inferência é atribuído sempre *Double*.

```
var valor = 7.5
print(valor is Float)
print(valor is Double)
```

```
false
true
```

Literais Numéricos

Inteiros podem ser escritos como ***decimal***, ***binário***, ***octal*** e ***hexadecimal***;

- decimal, sem prefixo
- binário, com um prefixo ***0b***
- octal, com um prefixo ***0o***
- hexadecimal, com um prefixo ***0x***

Literais Numéricos

```
let decimalInteger = 17
let binaryInteger = 0b10001 // 17 em
binário
let octalInteger = 0o21  // 17 em octal
let hexadecimalInteger = 0x11 // 17 em
hexa

print(binaryInteger)
print(hexadecimalInteger)
```

```
17
17
```

Literais Numéricos

- Pontos Flutuantes podem ser decimais(sem prefix) ou hexadecimais(com prefixo ***0x***)
- Floats decimais podem ter um *expoente* opcional, indicado por um ***e*** maiúsculo ou minúsculo. Floats hexadecimais utilizam um ***p***
  - 1.25e2 significa 1.25 x 10^2, ou 125.0. (***e*** é base 10)
  - 1.25e-2 significa 1.25 x 10^{-2}, ou 0.0125.
  - 0xFp2 significa 15 x 2^2, ou 60.0. (***p*** é base 2)
  - 0xFp-2 significa 15 x 2^{-2}, ou 3.75.

Literais Numéricos

- Possuem formatação extra para melhor redigibilidade (uso de _).

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
let umMilhão = 1_000_000
let maisQueUmMilhão = 1_000_000.000_000_1
print(decimalDouble, exponentDouble,
hexadecimalDouble)
print(umMilhão)
print(maisQueUmMilhão)
```

```
12.1875 12.1875 12.1875
1000000
1000000.0000001
```

Booleans

- Possui valores ***true*** e ***false***

```
let verdadeiro = true
if verdadeiro {
   print ("verdadeiro = \(verdadeiro)")
} else {
   print ("falso \(!verdadeiro)")
}
```

```
verdadeiro = true
```

Strings e Caracteres

- Uso de \ para alguns caracteres especiais na string

```
var barraInvertida = String()
barraInvertida = "Swift\\ObjetiveC"
let tab = "Nome\tNath"
let quebraLinha = "Nome\nNath"
let aspas = "\"entre aspas \""

print(barraInvertida+"\n", tab+"\n",
quebraLinha+"\n", aspas)
```

```
Swift\ObjetiveC
 Nome    Nath
 Nome
Nath
 "entre aspas "
```

Strings e Caracteres

- Strings Multilines(“ ” ”)
- Strings como vetor de caracteres
- Concatenação (***+, +=***) e comparação(***==***) de Strings
- Escapes (***\\, \t, \0, \n, \r, \”, \’***)
- Unicode ***(\u{n})***, n possui de 1 a 8 dígitos em Hexadecimal
- Principais métodos: ***append(), count, isEmpty, startIndex, endIndex, index(before:), index(after:), index(_:offsetBy:), insert(_:at:), remove(at:)***
- Iteração com laço ***for***

Strings e Caracteres

```
let multilineString = """
    There is a house \
    in New Orleans
    They call the Rising Sun
    """
print(multilineString)

let catCharacters: [Character] = ["C", "a",
"t", "!", "🐱"]
let catString = String(catCharacters)
print(catString)
var lp = "Linguagem "
lp += "de Programação"
print(lp)
```

```
There is a house in New Orleans
They call the Rising Sun
Cat!🐱
Linguagem de Programação
```

Strings e Caracteres

```
let caféDeUmJeito = "caf\u{E9} é bom"
let caféDeOutroJeito = "caf\u{65}\u{301}
\u{65}\u{301} bom"
if(caféDeUmJeito == caféDeOutroJeito) {
   print("são iguais")
} else {
   print("são diferentes")
}
let latimLetraA: Character = "\u{41}"
let cirilicoLetraA: Character = "\u{0410}"
if latimLetraA != cirilicoLetraA {
   print("são diferentes")
}
```

```
são iguais
são diferentes
```

Strings e Caracteres

```
var dog = "Dogs"
for character in dog {
   print(character)
}

dog.append("🐶")
print(dog)

//dog.insert(" are better than cats", at:
dog.endIndex)
dog.insert(contentsOf: " are better than cats",
at: dog.endIndex)
print(dog)
```

```
D
o
g
s
Dogs🐶
Dogs🐶 are better than cats
```

Optionals

- É um tipo que representa ***nil*** ou um ***valor empacotado***
- Podemos declarar uma variável com um sinal de interrogação ***(?)*** após o tipo para dizer ao compilador que ela aceitará o valor ***nil*** além de um valor do tipo especificado.
- Importante para lidar com retorno de funções
- Optional é uma enumeração de dois casos, ***Optional.none*** que é equivalente ao ***nil***, e ***Optional.some(Wrapped)*** que armazena o ***valor empacotado***
- Desempacotamento feito através do operador de exclamação ***(!)***, ou por controle de fluxo

Optionals

```swift
var inteiro: Int? = 1
inteiro = nil
var Inteiro: Optional<Int> = 1
Inteiro = nil // Erro!

let stringNumero1 = "1"
let numero1 = Int(stringNumero1)
print(numero1!, type(of:numero1))

let stringNumero2 = "dois"
let numero2 = Int(stringNumero2)
//print(numero2!, type(of:numero2))
```

```
1 Optional<Int>
```

Optionals

```
let stringNumero1 = "1"
let numeroInteiro1 = Int(stringNumero1)
let soma = numeroInteiro1! + 1
print(soma)

let stringNumero2 = "Dois"
let numeroInteiro2 = Int(stringNumero2)
if let a = numeroInteiro2 {
   print(a * 2)
}
```

```
2
```

Arrays

- Coleção de dados indexados por inteiros de 0 a N- 1, onde N é o tamanho da coleção
- Arrays são fortemente tipados, ou seja, só podem conter elementos de mesmo tipo
- Possui iteração com laço ***for***
- Principais métodos: ***isEmpty, append(), insert(_:at:), remove(at:), removeLast(), count, sort(),***

Arrays

```
var vetorPar = [2, 4, 6]
print(vetorPar.count)
vetorPar.insert(6, at: 0)
vetorPar.remove(at:3)
print(vetorPar)
for item in vetorPar {
    print(item)
}
var vetor:[Int] = [1, 3, 5]
vetor += vetorPar
vetor.sort()
print(vetor)
```

```
3
[6, 2, 4]
6
2
4
[1, 2, 3, 4, 5, 6]
```

Arrays

```swift
var frase = [String]()
frase = ["My", "mother", "was", "a", "tailor"]

for (index, value) in frase.enumerated() {
   print("Item \(index): \(value)")
}
```

```
Item 1: My
Item 2: mother
Item 3: was
Item 4: a
Item 5: tailor
```

Dicionários

- São vetores associativos
- Armazena pares com chave e valor ***(key : value)***
- No momento da indexação, retornam o tipo da chave , ***opcional***, porque pode ser que a chave não exista. Por isso para acessar o valor de uma chave precisamos desempacotar a entrada com ***“if let”***
- Possui iteração com laço for
- Principais métodos: ***count, isEmpty, updateValue(_:forKey:), removeValue(forKey:), values, keys***

Dicionários

```
var preços = Dictionary<String,Double>()
preços = ["café":5.00, "açúcar":3.00,
"filtro":2.50 ]
print(preços)

preços["cafeteira"] = 89.90
preços["açúcar"] = 4.00
preços.updateValue(5.50, forKey: "café")
print(preços)

preços["filtro"] = nil
preços.removeValue(forKey:"cafeteira")
print(preços)
```

```
["café": 5.0, "açúcar": 3.0, "filtro": 2.5]
["café": 5.5, "filtro": 2.5, "açúcar": 4.0,
"cafeteira": 89.900000000000006]
["café": 5.5, "açúcar": 4.0]
```

Dicionários

```
for (chave, valor) in preços {
    print("preço do \(chave) é: R$\(valor) ")
}

let preçoDoCafé = preços["café"]
print(preçoDoCafé!)
if let preçoDoAçúcar = preços["açúcar"] {
    print("O preço do açúcar é: R$ \(preçoDoAçúcar)")
}
```

```
preço do café é: R$5.5
preço do açúcar é: R$4.0

5.5
O preço do açúcar é: R$ 4.0
```

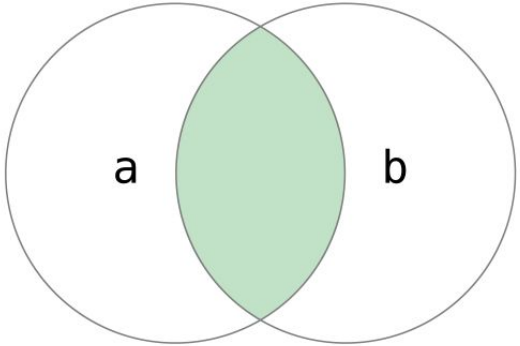
Conjuntos

- Usado para armazenar informações onde a ordem não importa
- Armazena valores distintos de um mesmo tipo
- Possui os fundamentos básicos de Conjuntos ***(União, Interseção, Diferença Simétrica, Subtração)***
- Principais métodos: ***isEmpty, count, insert(), remove(), contains(), intersection(), symmetricDifference(), union(), subtracting(), sorted()***

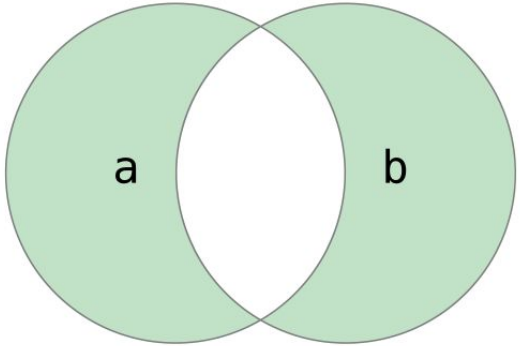
Conjuntos

- ***intersection(_:)*** cria um novo conjunto com os valores em comum entre os dois outros
- ***symmetricDifference(_:)*** cria um novo conjunto com os valores dos dois, exceto valores em comum
- ***union(_:)*** cria um conjunto com todos os valores dos dois conjuntos, mas não repete
- ***subtracting(_:)*** cria um conjunto com valores que não pertence ao outro conjunto

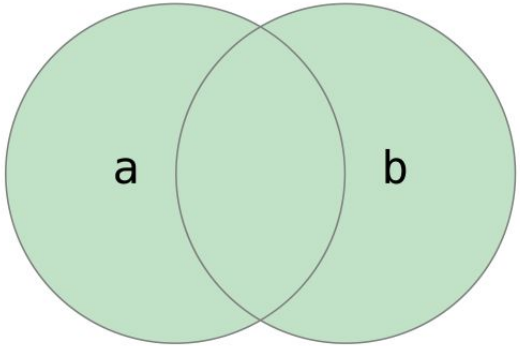
`a.intersection(b)`



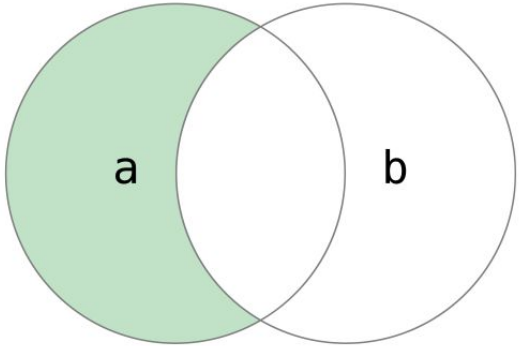
`a.symmetricDifference(b)`



`a.union(b)`



`a.subtracting(b)`



Conjuntos

```
var conjunto1 = Set<Int>()
conjunto1.insert(1)
conjunto1.insert(7)
var conjunto2: Set = [1, 2, 3]
print(conjunto1.union(conjunto2))
print(conjunto1.intersection(conjunto2))
print(conjunto2.subtracting(conjunto1))
print(conjunto1.symmetricDifference(conjunto2))

for value in conjunto2.sorted() {
    print(value)
}
```

```
[7, 2, 3, 1]
[1]
[2, 3]
[7, 2, 3]
1
2
3
```

Enumerados

- Possui uma sintaxe mais completa do que nas outras linguagens
- Podem armazenar valores de tipos diferentes

```
enum Bussola {
   case Norte, Sul, Leste, Oeste
}
var direção = Bussola.Sul // inferência do tipo Bussola
direção = .Norte

enum horarioAula {
    case NoHorario
    case Atrasado(Int) // atrasado alguns minutos
}
```

Enumerados

```
func descrição(status: horarioAula) {
    switch status {
        case .NoHorario:
            print("A aula começou no horário certo ")
        case .Atrasado(let min):
            print("A aula está atrasada \(min) minutos")
    }
}
var status = horarioAula.NoHorario
descrição(status: status)
status = .Atrasado(10)
descrição(status: status)
```

```
A aula começou no horário certo
A aula está atrasada 10 minutos
```

(Des)Alocação de memória

- A alocação de tipos primitivos é feita na pilha, os tipos compostos ou de coleções são alocados no monte mas o compilador pode alocá-los na pilha para otimizar o processo em alguns casos
- A desalocação pode ser feita pelo programador para classes utilizando o método ***deinit***, ou ela será feita pelo coletor de lixos ARC(Automatic Reference Counting)
- ARC é um contador de referências

Entrada e Saída

- Linha de comando
  - print() para output
  - readLine() para input
- Arquivos, utilizando FileManager do Foundation:

```
let response = readLine()
let arquivo: FileHandle? = FileHandle(forReadingAtPath: response!)

//LEITURA DO ARQUIVO:
if arquivo != nil {
let dados = arquivo?.readDataToEndOfFile()
}
```

Entrada e Saída

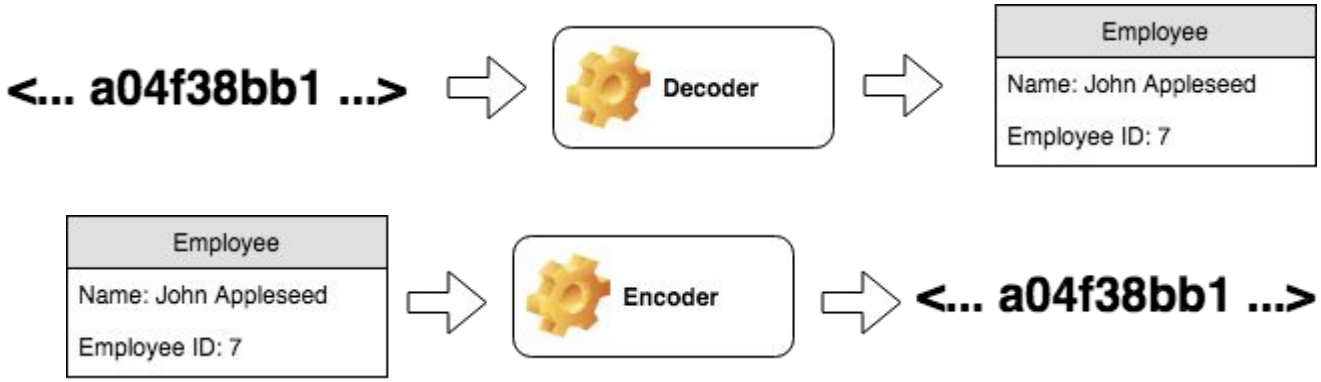
```
let componentes = NSString(data: dados!, encoding:
String.Encoding.utf8.rawValue)

//Separacao das linhas

let linhas = componentes!.components(separatedBy: "\n")
for linha in linhas {
}
```

Serialização

- É possível utilizando um protocolo chamado ***Codable***
- Utiliza dois protocolos chamados Encodable e Decodable que fazem o encode e decode do dado para/de uma representação externa.



Expressões e Comandos

Operadores

- Podem ser unários, binários ou ternários
- **Operadores de atribuição:**

```
let b = 10
let (x, y) = (1, 2) // x vale 1 e y vale 2
var a = 5
a = b   // a = 10
a += b  // a = 20
```

Operadores

- O operador de atribuição não retorna um valor. Evita efeitos colaterais!

```
if x = y {
        // operação inválida!!!
}
```

Operadores

- **Operadores aritméticos:**
    - não suportam overflow por default. É possível suportar overflow através de operadores específicos de overflow

- Adição (+)
- Subtração (-)
- Multiplicação (*)
- Divisão (/)

- Adição com overflow (&+)
- Subtração com overflow (&-)
- Multiplicação com overflow (&*)

Operadores

```
1 + 2   // 3
10.0 / 2.5  // 4.0
"hello, " + "world"  //  "hello, world"
9 % 4    //  1
-9 % 4   //  -1
2 + 3 % 4 * 5 // 17 (precedência de operadores!)
```

Operadores

- operador unário de menos

```
let tres = 3
let menosTres = -tres // -3
```

- operador unário de mais

```
let menosSeis = -6
let tambemMenosSeis = +menosSeis  // -6
```

Operadores

- **Operadores de comparação:**

- Igual a (a == b)
- Nâo igual a (a != b)
- Maior que (a > b)
- Menor que (a < b)
- Maior ou igual que (a >= b)
- Menor ou igual que (a <= b)

Comparação de tuplas:

```
(2, "zebra") < (2, "amor") // false
```

Usados para objetos que possam referenciar a mesma instância de uma classe:

- Idêntico a (===)
- Não idêntico a (!==)

Operadores

- Checagem de tipo **is**
- Type casting **as, as? e as!**

```
let x = “Sou uma string!”
if x is String {
        let y  = x as! String
print(“x e y são Strings!”)
}
if let z = x as? Int {
    print(“x é um Int!”)
} else if let z = x as? String {
    print(“x é uma String!”)
}
```

Operadores

- **Operadores bitwise:**

- not ~
- and &
- or |
- xor ^
- shift left <<
- shift right <<

Operadores

- **Curto-circuito:**

```
c = a != nil ? a! : b // b não é avaliado caso a seja diferente de nil
//outra forma de escrever:
c = (a ?? b)
    if forVerdade && tambemForVerdade { // se forVerdade == false, tambemForVerdade não é avaliado
       print("Aeee!")
}
if forVerdade || tambemForVerdade { // se forVerdade == true, tambemForVerdade não é avaliado
       print("Aeee!")
}
```

Fluxo de controle

- **For-in:**

```
let names = ["Ana", "José", "João", "Clotilde"]
for name in names {
print("Olá, \(name)!")
}

let numeroDePernas = ["aranha": 8, "formiga": 6, "gato": 4]
for (nomeAnimal, contagemPernas) in numeroDePernas {
print("\(nomeAnimal)s têm \(contagemPernas) pernas")
}
```

Fluxo de controle

- For-in:

```
for index in 1...5 {
print("\(index) vezes 5 é \(index * 5)")
}
let base = 3
let potencia = 10
var resposta = 1
for _ in 1...potencia {
  resposta *= base
}
```

Fluxo de controle

- **While:**

```
while semestre == true {
    print(“socorro”)
}
```

- **Repeat-while:**

```
repeat {
    print(“socorro”)
} while semestre == true
```

Fluxo de controle

- **if e else:**

```
if 1 < 2 {
   print(“que bom”)
} else {
   print(“como assim?”)
```

Fluxo de controle

- **switch:**

```
let umCaractere: Character = "z"
switch umCaractere {
   case "a":
       print("inicio do alfabeto")
   case "g", "h", "i":
       print("meio do alfabeto")
       fallthrough
   case let y where y == “k”:
       print(“kkk”)
   case "z":
       print("final do alfabeto")
   default:
       print("nao sei")
}
```

Fluxo de controle

- **continue:**

```
var presente = 10
while presente > 1 {
   if presente == 5 {
       continue
   }
   presente -= 1
}
```

Fluxo de controle

- **break:**

```
var presente = 10
while presente > 1 {
   if presente == 5 {
       break
   }
   presente -= 1
}
```

Funções

- Estrutura básica:

```
func cumprimenta(pessoa: String) -> String {
        let cumprimento = "Olá, " + pessoa + "!"
return cumprimento
}
```

Funções

- Manipulação de retornos - tuplas:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
  if array.isEmpty { return nil }
  var currentMin = array[0]
  var currentMax = array[0]
  for value in array[1..<array.count] {
      if value < currentMin {
          currentMin = value
      } else if value > currentMax {
          currentMax = value
      }
  }
  return (currentMin, currentMax)
}
```

Funções

```
if let extremos = minMax(array: [8, -6, 2, 109, 3, 71]) {
   print("min é \(extremos.min) e max é \(extremos.max)") //
Imprime “min é -6 e max é 109”
}
```

Funções

- **Rótulo de argumento vs Nome de parâmetro**
  - Rótulo de argumento: usado na chamada da função
  - Nome de parâmetro: usado na implementação da função
  - Por default, quando não especificado na definição, o rótulo do argumento é o nome do parâmetro

```
func qualquerCoisa(rotuloArgumentoUm nomeParametroUm: Int, rotuloArgumentoDois
nomeParametroDois: Int) -> Int {
   let soma = nomeParametroUm + nomeParametroDois
   return soma
}
let soma = qualquerCoisa(rotuloArgumentoUm: 1, rotuloArgumentoDois: 2)
```

Funções

- Omitindo o rótulo dos argumentos
  - para omitir o rótulo e não precisar explicitá-lo na chamada da função, usar o underscore (_)

```
func qualquerCoisa(_ nomeParametroUm: Int, nomeParametroDois: Int) -> Int {
    let soma = nomeParametroUm + nomeParametroDois
    return soma
}
let soma = qualquerCoisa(1, nomeParametroDois: 2)
```

Funções

- **Parâmetros default**

```
func qualquerCoisa(nomeParametroUm: Int, nomeParametroDois: Int = 13) -> Int
{
   let soma = nomeParametroUm + nomeParametroDois
    return soma
}
let soma = qualquerCoisa(nomeParametroUm: 1)
```

Funções

- **Parâmetros default**
  - A função print() estipula um parâmetro default \n (chamado **terminator**), imprimindo as informações com quebra de linha
  - Para alterar este valor, é preciso chamar a função como print(“olá!”, terminator: “”)

Funções

- **Parâmetros variantes**
  - especificados com reticências (...)
  - para a passagem de um número variado de parâmetros
  - funções devem ter no máximo um parâmetro variante

```
func totalAritmetico(_ numbers: Double...) -> Double {
  var total: Double = 0
  for number in numbers {
      total += number
  }
  return total / Double(numbers.count)
}
let x = totalAritmetico(1, 2, 3, 4, 5) // x recebe 3.0
let y = totalAritmetico(3, 8.25, 18.75) // y recebe 10.0
```

Funções

- **Parâmetros In-Out**
  - Swift define parâmetros como constantes, imutáveis
  - tentar mudar o valor de um parâmetro no escopo de uma função gera erro de compilção
  - para modificar parâmetros (e fazer com que a mudança permaneça fora da função), definir eles como inout
    - OBS: parâmetros inout devem ser variáveis, não podem ser variantes e não aceitam valores default

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
  let temporaryA = a
  a = b
  b = temporaryA
}
var umInt = 3
var outroInt = 107
swapTwoInts(&umInt, &outroInt)
print("umInt agora é \(umInt), e outroInt agora é \(outroInt)") // imprime “umInt agora é 107, e
outroInt agora é 3”
```

Funções

- **Tipos função**
  - Cada função possui um tipo formado pelos tipos dos parâmetros e do retorno

```
func addTwoInts(_ a: Int, _ b: Int) -> Int { // tipo: (Int, Int) -> Int
   return a + b
}
```

  - O tipo função pode ser usado como qualquer outro. Por exemplo, definindo variáveis:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
print("Resultado: \(mathFunction(2, 3))") // Imprime “Resultado: 5”
```

Funções

- **Tipos função**
  - Estes tipos também podem ser passados como parâmetros de funções

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
   return a + b
}

func imprimeResultado(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
  print("Resultado: \(mathFunction(a, b))")
}
imprimeResultado(addTwoInts, 3, 5) // imprime “Resultado: 8”
```

Funções

- **Funções aninhadas**
  - Swift permite funções aninhadas, definidas dentro de outras funções. Por default, as funções de dentro ficam escondidas do escopo global, mas podem ser chamadas e usadas pela função de fora
  - As funções de fora podem retornar uma de suas funções aninhadas e assim esta pode ser usada fora do seu escopo original

Funções

- **Funções aninhadas**

```
func opera(com simbolo:String) -> (Int, Int) -> Int {
    func adicao(num1:Int, num2:Int) -> Int {
        return num1 + num2
    }
    func subtracao(num1:Int, num2:Int) -> Int {
        return num1 - num2
    }
    let operacao = (simbolo ==  "+") ?  adicao : subtracao
    return operacao
}

let operacao = opera(com:  "+")
let resultado = operacao( 2, 3)
print(resultado)     // imprime 5
```

Modularização

Classes e Estruturas

Classes possuem capacidades adicionais que Structs não possuem:

- Herança
- TypeCast permite a identificação do tipo de uma classe
- Deinitializers
- Contadores de referência permitem que mais de uma referência seja feita a uma instância

Estruturas

```
struct Pessoa {
    private  var nome: String?
    public var telefone: Int
    internal let endereço : String

    init (nome: String, telefone: Int,
endereço: String){
        self.nome = nome
        self.telefone = telefone
        self.endereço = "Serra"
    }

    func getNome() -> String? {
        return nome
    }
```

```
    func info() -> [String] {
      var str: [String] = []

       if let nome = self.nome {
           str.append(nome)
       }

       let telefone = self.telefone
       str.append(String(telefone))

      return str
   }
}
```

Classes

```
class Pessoa {

    private var nome: String?
    public var telefone: Int
    internal let endereço = "Vitória"

    init (telefone: Int, nome: String) {
        self.telefone = telefone
        self.nome = nome
    }

    func getNome() -> String? {
        return nome
    }

    func setNome(nome: String) {
        self.nome = nome
    }
    func info() -> [String] {
        var str: [String] = []
        //Pode ser nula:
        if let nome = self.nome {
            str.append(nome)
        }
        let telefone = self.telefone
        str.append(String(telefone))
        return str
    }
}
```

Particularidades das classes

```
//Classe Pessoa possui atributo qtdCafé = "Moderada" e atributo nome é public
class Estudande: Pessoa {
   let universidade = "UFES"
   init (nome: String, telefone: Int, endereço: String, qtdCafé: String) {
      super.init(nome:nome,telefone:telefone,endereço:endereço)
      super.qtdCafé = "Alta!"
   }
   override func info () -> [String] {
       var str: [String] = []
       if let nome = self.nome {
           str.append(nome)
       }
       let telefone = self.telefone     str.append(String(telefone))
       str.append (universidade)
      return str
   }
}
```

Particularidades das classes

```swift
class Banco {
    static var moedasBanco = 10_000
    static func receber(moedas: Int) {
        self.moedasBanco += moedas
    }
}

class Jogador {
    var moedas: Int
    init(moedas: Int) {
        self.moedas = moedas
    }
    deinit {
        Banco.receber(moedas: moedas)
    }
}
```

```swift
var jogador: Jogador? =
Jogador(moedas: 100)

print("O novo jogador possui:
\(jogador!.moedas) moedas")

print("O banco possui
\(Banco.moedasBanco) moedas")

jogador = nil
print("Jogador saiu do jogo")

print("O banco possui
\(Banco.moedasBanco) moedas")
```

```
O novo jogador
possui: 100 moedas

O banco possui
10000 moedas


Jogador saiu do
jogo

O banco possui
10100 moedas
```

Uso de estruturas

Parte do desenvolvedores prefere usar estruturas, ao invés de classe, devido a alguns fatores como:

- São mais confiáveis para dados pequenos, pois não é referenciado e sim copiado. É mais seguro criar cópias do que fazer múltiplas referências a uma instância.
- Menor preocupação com acesso ilegal da memória

Extensões

Servem para adicionar funcionalidades de uma forma organizada a classes, enumeradores, estruturas ou protocolos.

```
extension Pessoa{
   var saudacao: String {return "Ei " + self.getNome()!}
}

var pessoa = Pessoa (nome:"Nat",telefone:12345678,endereço:"Vitória")
print(pessoa.saudacao)
```

Protocolos

Os protocolos prometem que uma classe particular implemente um conjunto de métodos.

```
protocol  AddStrings{
   func toString() -> String
}

extension String:
AddStrings{
   func toString() ->
String{
        return  self
   }
}

var aux: AddStrings
print (aux.toString())
```

Nat

Módulos

Existem quatro formas primárias de organização de um código em Swift:

- Módulos
- Arquivos Fonte
- Classes
- Blocos de Código

Módulos

Ao importar um módulo ele especificará:

- Namespace
- Controle de acesso

O controle de acesso se divide em:

- Open (fora do módulo)
- Public (fora do módulo, mas subclass e o override são apenas no módulo de origem.)
- Internal (somente no módulo)
- File-private (dentro do arquivo)
- Private (apenas a partir da declaração de inclusão)

```
//Importa todo o módulo
import Foundation

//Importa apenas o tipo
(typealias, struct, class,
enum, protocol, var, func) de
um módulo
import class
Foundation.NSString
```

Pacotes

O Framework ‘Gerenciador de pacotes’ oferece um sistema convencional para criar bibliotecas e executáveis, e compartilhar código entre projetos diferentes.

Comandos: *swift package*, *swift build* e *swift test*.

```
import PackageDescription

let package = Package(
    name: "ProjetoSwift"
)
```

```
nat@nat-queiroz:~/Documentos/ProjetoSwift$ swift build -c release
Compile Swift Module 'ProjetoSwift' (5 sources)
Linking ./.build/release/ProjetoSwift
nat@nat-queiroz:~/Documentos/ProjetoSwift$
```

Polimorfismo

Ad-hoc

- **Coerção:**

```
func recebeOpcional(value: Int?) { }
let x: Int = 1
recebeOpcional(value: x) // converte um tipo não opcional para um tipo opcional
// ---------------//
func recebeDouble(value: Double) { }
recebeDouble(value: 2) // converte um Int para um Double (tipo mais amplo); não faz estreitamento

// ---------------//

func recebeDouble(value: Double) { }
let x: Int = 2
recebeDouble(value: x) // ERRO! tipo já foi inferido em x e não é possível convertê-lo implicitamente
```

Ad-hoc

- **Sobrecarga:**

```
let umInteiro = 1 + 2
let umaString = "hello, " + "world"
// ---------------//
struct Vector2D {
   var x = 0.0, y = 0.0
}
extension Vector2D {
   static func + (left: Vector2D, right: Vector2D) -> Vector2D {
       return Vector2D(x: left.x + right.x, y: left.y + right.y)
       }
}
let vector = Vector2D(x: 3.0, y: 1.0)
let outroVector = Vector2D(x: 2.0, y: 4.0)
let somaVectors = vector + outroVector
print("soma = \(somaVectors)") // imprime “”soma = Vector2D(x: 5.0, y: 5.0)”
```

Ad-hoc

- **Sobrecarga:**

```
func informaValor(_ valor: Int) {
            print("Valor = \(valor)")
}

func informaValor(_ valor: Double) {
   print("Valor = \(valor)")
}

let a = 1
let b = 1.0

informaValor(a) // imprime "Valor = 1"
informaValor(b)     // imprime "Valor = 1.0"
```

Universal

- **Paramétrico:**
  - Parametrização de funções:

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
       let temporaryA = a
       a = b
        b = temporaryA
}
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)


var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
```

Universal

- **Paramétrico:**
  - Parametrização de estruturas:

```
struct Stack<Element> {
        var items = [Element]()
        mutating func push(_ item: Element) {
                items.append(item)
        }
        mutating func pop() -> Element {
                return items.removeLast()
        }
}
var stackOfStrings = Stack<String>()
var stackOfInts = Stack<Int>()
```

Universal

- **Inclusão:**

```
class Veiculo {
          var velocidadeAtual = 0.0
          var descricao: String {
               return "Andando a \(velocidadeAtual) km/h"
          }
     func fazerBarulho() {  }
}
class Carro: Veiculo {
   var possuiCarroceria = false
}
class Taxi: Carro {
   var quantidadeAtualPassageiros = 0
}
```

Exceções

Tratamento de erros

- Swift não suporta exceções não checadas
- Os erros são representados com valores que conformam com o protocolo Error
- O tipo enumerado é o mais adequado para se definir erros simples

```
enum VendingMachineError: Error {
  case invalidSelection
  case outOfStock
}
```

Tratamento de erros

- Há quatro maneiras de tratar exceções:
  - propagar o erro - funções, métodos ou inicializações “throwing”

```
func canThrowErrors() throws -> String
```

    - dentro da função:

```
guard item.count > 0 else {
   throw VendingMachineError.outOfStock
}
```

    - o método que chama funções *throwing* precisa obrigatoriamente tratar ou continuar a propagar as exceções recebidas

Tratamento de erros

- tratar o erro com do-catch:
  - uso do try

```
do {
   try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
} catch VendingMachineError.outOfStock {
   print("Out of Stock.")
}
```

Tratamento de erros

- converter o erro para um valor opcional:
  - uso do try?

```
func buscaDado() -> Dado? {
  if let dado = try? buscaDadoNoDisco() { return dado }
  if let dado = try? buscaDadoNoServidor() { return dado }
  return nil
}
```

Tratamento de erros

- descartar a possibilidade de erro, parando a propagação
  - útil quando se tem certeza que o método não causará uma exceção
  - uso do try!

```
func loadImage (atPath: String) throws -> UIImage
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Concorrência

Concorrência - a classe Thread

- O framework Foundation oferece uma classe Thread, baseada internamente na pthread

```
var t = Thread {
   print("Comecei!")
}
t.start()
```

Concorrência - primitivas de sincronização

- As funcionalidades básicas usadas para sincronizar threads são travas e semáforos
- **NSLock**
  - Quando requisitado, adquire o lock ou entra em espera (lock indisponível)
  - A ordem de aquisição dos locks é “unfair”: não é possível garantir que o primeiro a requisitar será o primeiro a receber

Concorrência - primitivas de sincronização

```
let lock = NSLock()
class LThread : Thread {
   var id:Int = 0
   convenience init(id:Int){
       self.init()
       self.id = id
   }
   override func main(){
       lock.lock()
       print(String(id)+" acquired lock.")
       lock.unlock()
       if lock.try() {
           print(String(id)+" acquired lock again.")
           lock.unlock()
       }else{  // If already locked move along.
           print(String(id)+" couldn't acquire lock.")
       }
       print(String(id)+" exiting.")
   }
}
```

Concorrência - primitivas de sincronização

```
var t1 = LThread(id:1)
var t2 = LThread(id:2)
t1.start()
t2.start()
```

Concorrência - primitivas de sincronização

- **NSContidionLock**

```
let cond = NSCondition()
var available = false
var SharedString = ""
class WriterThread : Thread {
   override func main(){
       for _ in 0..<5 {
           cond.lock()
           SharedString = "💩"
           available = true
           cond.signal()
           cond.unlock()
       }
   }
}
```

Concorrência - primitivas de sincronização

```
class PrinterThread : Thread {
   override func main(){
       for _ in 0..<5 {
           cond.lock()
           while(!available){
               cond.wait()
           }
           print(SharedString)
           SharedString = ""
           available = false
           cond.unlock()
       }
   }
}
```

Concorrência - primitivas de sincronização

```
let writet = WriterThread()
let printt = PrinterThread()
printt.start()
writet.start()
```

Concorrência - Grand Central Dispatch (GCD)

- API que permite executar closures em gurpos de trabalho
- Funciona através de filas de despacho
- Utiliza a classe DispatchQueue
- Cada item submetido a uma queue é processado em um conjunto de threads administradas pelo sistema

Concorrência - Grand Central Dispatch (GCD)

- Criando uma queue:

```
let fila1 = DispatchQueue(label: "com.lpclass.myfirstqueue")
let fila2 = DispatchQueue(label: "com.lpclass.mysecondqueue")
```

Concorrência - Grand Central Dispatch (GCD)

- Modos sync e async:
  - Executa todo o bloco de uma vez

```
fila1.sync {
   for i in 0..<10 {
       print(i)
   }
}
```

  - Execução do programa não espera pelo fim da tarefa do bloco

```
fila1.async {
   for i in 0..<10 {
       print(i)
   }
}
```

Concorrência - Grand Central Dispatch (GCD)

- Classes definem a prioridade das tarefas. Por ordem mais importante - menos importante:
  - userInteractive, userInitiated, default, utility, background, unspecified

```
let fila1 = DispatchQueue(label: "com.lpclass.myfirstqueue", qos: DispatchQoS.userInitiated)
let fila2 = DispatchQueue(label: "com.lpclass.mysecondqueue", qos: DispatchQoS.userInitiated)
```

Concorrência - Grand Central Dispatch (GCD)

- Atributo define concorrência e situação inicial

```
let fila1 = DispatchQueue(label: "com.lpclass.myfirstqueue", qos: .userInitiated,
attributes: [.concurrent, .initiallyInactive])
let fila2 = DispatchQueue(label: "com.lpclass.mysecondqueue", qos: .userInitiated,
attributes: .concurrent)
```

Concorrência - Grand Central Dispatch (GCD)

- Exemplo geral

```
let concurrentQueue = DispatchQueue(label: "com.lpclass.Concurrent", attributes:
.concurrent)
concurrentQueue.async {
   DispatchQueue.concurrentPerform(iterations: 5) {
     sleep(1)
     print("Dormi 1 segundo...")
   }
}
concurrentQueue.async (flags: .barrier) {
       print("As filas já dormiram o suficiente!")
}
```

Avaliação de linguagens

Critérios Gerais	C	C++	Java	Swift
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Não
Método de Projeto	Estruturado	Estruturado e OO	OO	Multiparadigma (OO, Imperativa, Funcional, O. Protocolo, estruturada em blocos)

Critérios Gerais	C	C++	Java	Swift
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim	Sim
Tipos primitivos e Compostos	Sim	Sim	Sim	Sim

Critérios Gerais	C	C++	Java	Swift
Gerenciamento de memória	Programador	Programador	Sistema	Sistema
Persistência de dados	Biblioteca de Funções	Biblioteca de classes e Funções	JDBC, biblioteca de classes, serialização	biblioteca de classes, serialização
Passagem de parâmetros	Lista variável e por valor	Lista variável, default, por valor e por cópia de referência	Lista variável, por valor e por cópia de referência.	Lista variável, default, por valor e por cópia de referência
Encapsulamento e proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática/Dinâmica	Estática/Dinâmica	Estática/Dinâmica

Critérios Gerais	C	C++	Java	Swift
Polimorfismo	Coerção e Sobrecarga	Todos	Todos	Todos
Exceções	Não	Parcial	Sim	Parcial
Concorrência	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim	Não (classe de funções)

Referências

- https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html
- https://developer.apple.com/swift/blog/
- https://www.uraimo.com/2017/05/07/all-about-concurrency-in-swift-1-the-present/
- https://insights.stackoverflow.com/survey/2017
- https://github.com/apple/swift-evolution/blob/master/proposals/0123-disallow-value-to-optional-coercion-in-operator-arguments.md
- https://swift.org/about/