
Swift

Linguagem de Programação 2018/2
Prof. Vítor E. Souza

Dyego Cavalieri Pansiere
Gabriel Dalla Bernardina Fagundes
Luiz Felipe Ribeiro
Melina Schneider Campo Dall'Orto
Sandor Ferreira

História e Curiosidades

História

- Linguagem recente: criada em 2014, tornou-se open source em 2015
- Criada para substituir Objective-C no desenvolvimento iOS e OS X para atender às novas demandas de programação
- Criada por Chris Lattner e Apple Inc.
- 2015 surge a Swift Community

Curiosidades

- 6º linguagem mais amada por desenvolvedores (2018)
- Algoritmos de Busca:

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

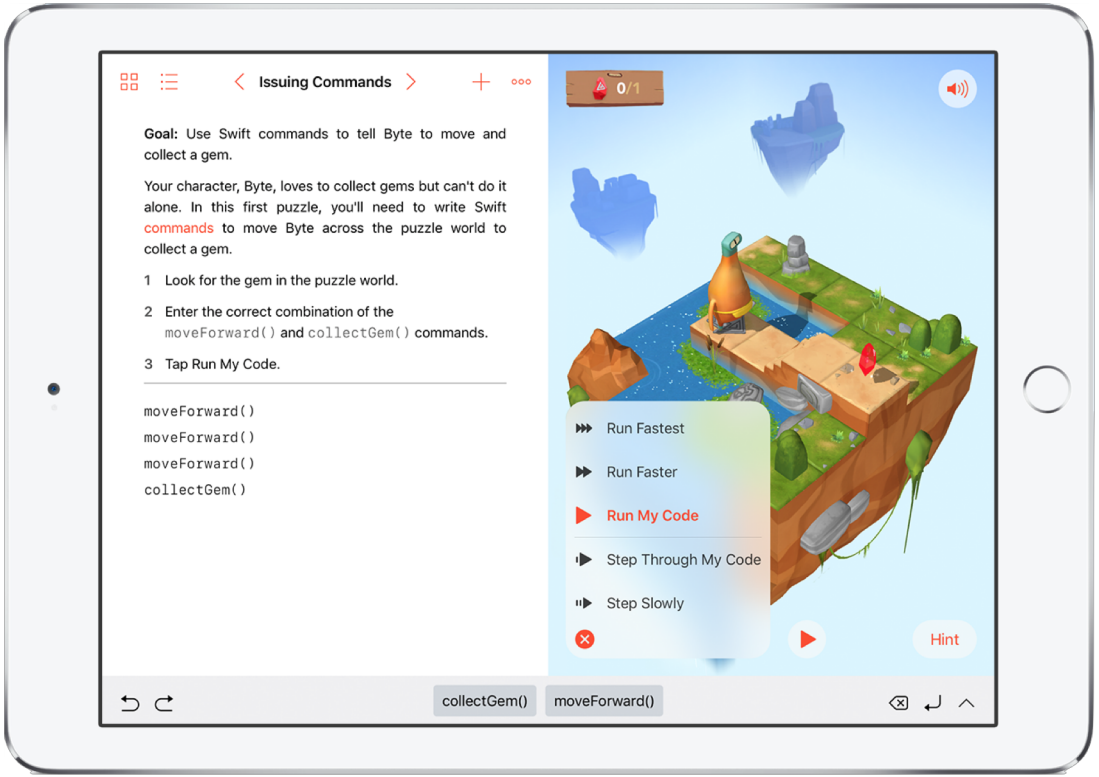
    Até **8,4**x mais rápido que Python 2.7

- Padronizada por Apple Inc.
- Licença open source *Apache License 2.0 em 3 de Dezembro, 2015*

Mini Tutorial

Playgrounds

- Interativo;
- Fácil de usar;
- Roda o código enquanto escreve;
- Compatível com robôs e drones.



Pré-Requisitos

- **Compilador** Swift para MacOS ou Ubuntu
- IDE ou Linhas de Código

XCode

Swift

MacOS

Ubuntu

Hello World!

```
1. print("Hello, world!")
2. // Printa "Hello, world!"
```

- Todo código em escopo global é um programa
- **Comentários:** //
- Não é necessário ponto e vírgula

Hello World!

É um programa completo em Swift!

```
1. print("Hello, world!")
2. // Printa "Hello, world!"
```

- Todo código em escopo global é um programa
- **Comentários:** //
- Não é necessário ponto e vírgula

Sintaxe

Variáveis e Constantes

```
// Variáveis
var minhaVariavel = 42
minhaVariavel = 50

// Constantes
let meuNome = "Meu nome"
```

```
// Variáveis
var minhaVariavel: Double = 42
minhaVariavel = 50.0

// Constantes
let meuNome: String = "Meu nome"
```

Sintaxe

Variáveis e Constantes

Declaração Implícita

```
// Variáveis
var minhaVariavel = 42
minhaVariavel = 50

// Constantes
let meuNome = "Meu nome"
```

Declaração Explícita

```
// Variáveis
var minhaVariavel: Double = 42
minhaVariavel = 50.0

// Constantes
let meuNome: String = "Meu nome"
```

Sintaxe

Expressões Condicionais

```
let a = 15
let b = 20
if b > a {
        print("\(b) é maior!")
}
print("\(b > a ? b : a) é maior!")
```

- Sem necessidade de parênteses
- Sem necessidade de ponto e vírgula
- Operador \() que chama a função "toString()" de qualquer objeto

Sintaxe

Expressões Condicionais

```
let a = 15
let b = 20
if b > a {
        print("\(b) é maior!")
}
print("\(b > a ? b : a) é maior!")
```

Saída:
20 é maior!

- Sem necessidade de parênteses
- Sem necessidade de ponto e vírgula
- Operador \() que chama a função "toString()" de qualquer objeto

Sintaxe

Funções

```
func oi(pessoa: String, dia: String) -> String {
        return "Oi \(pessoa), hoje é \(dia)."
}
oi(pessoa: "João", dia: "Terça-Feira")
```

```
func oi(_ pessoa: String,no dia : String) -> String {
        return "Oi \(pessoa), hoje é \(dia)."
}
oi("João", no: "Terça-Feira")
```

- Funções começam com "func"
- Retorno é indicado por "->"
- Parameter Labels: nomes auxiliares dos parâmetros

Sintaxe

Classes

```
class Forma {
    var lados: Int = 0
    var nome: String
    init(nome: String) {
        self.nome = nome
    }
    func descrição() -> String {
        return "A forma tem \(lados) lados."
    }
}
```

- "class" inicia uma classe
- "self" é equivalento ao "this" de Java
- Construtores são declarados com "init"
- Vários construtores são permitidos desde que possuam diferentes parâmetros

Sintaxe

Loops e Controladores de Fluxo

```
let scores = [12,13,14,5]
var max = scores[0]
for score in scores {
        if score > max {
            max = score
        }
}
print("Maior score é \(max)")
```

- for-in: cria e utiliza uma variável para percorrer
- Não é necessário parênteses assim como if

Sintaxe

Loops e Controladores de Fluxo

```
var n = 2
while n < 100 {
   n *= 2
}
print(n)
// Printa "128"
```

```
var m = 2
repeat {
   m *= 2
} while m < 100
print(m)
// Printa "128"
```

- while: avalia a expressão e entra no loop
- repeat-while: executa o bloco e continua caso avalie *true*

Sintaxe

Loops e Controladores de Fluxo

```
var n = 2
while n < 100 {
   n *= 2
}
print(n)
// Printa "128"
```

```
var m = 2
repeat {
   m *= 2
} while m < 100
print(m)
// Printa "128"
```

Saída:
128

- while: avalia a expressão e entra no loop
- repeat-while: executa o bloco e continua caso avalie *true*

Sintaxe

Loops e Controladores de Fluxo

```
let vegetal = "pimenta vermelha"
switch vegetal {
case "batata":
   print("Essencial!")
case "pepino", "agrião":
   print("Prefiro que seja numa salada.")
case let x where x.hasPrefix("pimenta"):
print("É forte essa \(x)?")
default:
   print("Tudo fica uma delícia na sopa.")
}
```

- switch: qualquer variável/constante de qualquer tipo pode ser avaliada
- Permite declarações dentro dos cases
- breaks não são necessários

Sintaxe

Loops e Controladores de Fluxo

```
let vegetal = "pimenta vermelha"
switch vegetal {
case "batata":
   print("Essencial!")
case "pepino", "agrião":
   print("Prefiro que seja numa salada.")
case let x where x.hasPrefix("pimenta"):
print("É forte essa \(x)?")
default:
   print("Tudo fica uma delícia na sopa.")
}
```

Saída:
É forte essa pimenta vermelha?

- switch: qualquer variável/constante de qualquer tipo pode ser avaliada
- Permite declarações dentro dos cases
- breaks não são necessários

Guia da Linguagem

Paradigmas

Multiparadigmal

- Imperativo (principal)

```
let numbers = [1, 2, 3, 4, 5]
var evenNumbers = [Int]()
for i in 0..<numbers.count {
    let number = numbers[i]
    if number % 2 == 0 {
        evenNumbers.append(number)
    }
}
```

- Funcional (poderoso)

```
let evenNumbers = [1, 2, 3, 4, 5].filter {
(number) -> Bool in
    if number % 2 == 0 {
        return true
    } else {
        return false
    }
}
```

Execução

- Interoperável apenas com C e Objective-C
- É uma linguagem otimizada e compilada;
- Faz sobrescrita de todos os operadores e tipos básicos na Swift

  Standard Library para dizer as operações ao compilador

Amarrações

- Amarração estática e escopo estático aninhado

```
func add(a: Int, b: Int) -> Int {
    let c = 10
    print("adicionei \(a) e \(b) com \(c) 😜");
    return a + b + c;
}

let c = add(a: 10, b: 15)
print(c)
```

Amarrações

- Amarração estática e escopo estático aninhado

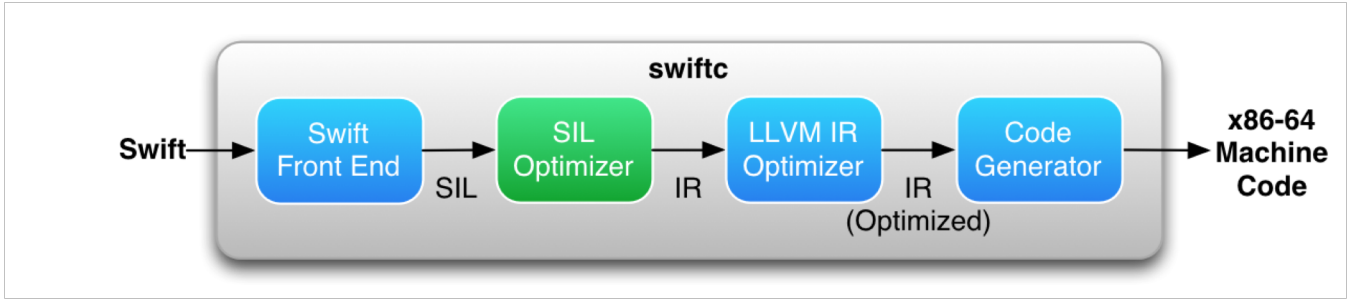
```
func add(a: Int, b: Int) -> Int {
    let c = 10
    print("adicionei \(a) e \(b) com \(c) 😜");
    return a + b + c;
}

let c = add(a: 10, b: 15)
print(c)
```

Saída:
adicionei 10 e 15 com 10 😜
35

Amarrações

- Inferência de tipos em tempo de compilação;
- Otimizada com Swift Intermediate Language:



Identificadores

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

- Podem conter quase todos os caracteres incluindo Unicode
- Não podem **começar com números**, não pode conter **espaços**, **símbolos matemáticos**, **setas**, **traços** e **box-drawing**.
- Num closure sem nomes de parâmetros, é possível um identificador iniciar com ($), pois estes já declaram variáveis como $0 ou $01 por default

Identificadores

```
var `class` = 45
`class` = 20

let `Int` = "Sou um inteiro sqn"
```

- Palavras reservadas podem ser usadas como identificadores se acompanhadas de backticks (`)
- Esse uso não é recomendado a não ser que seja estritamente necessário

Identificadores

Keywords para Declaração:

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

Identificadores

Keywords para Statements:

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

Identificadores

Keywords para Expressões:

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

Identificadores

Keywords iniciadas com símbolos:

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

Identificadores

Keywords para 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

- Possui Tipagem Estática.
- Fortemente tipada.
- Possui Inferência de tipo.
- É Case-sensitive.
- Apesar da inferência de tipo, é possível fazer “type annotations”

```
var red, green, blue: Double
```

Valores e tipos de dados

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 - Int

- 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 is equal to 0, and is of type UInt8

let maxValue = UInt8.max  // maxValue is equal to 255, and is of type UInt8
```

Tipo Inteiro - Int

- Inteiros seguem a convenção de nomes similar a C, desse modo um 8-bit “unsigned integer” é do tipo UInt8, e um 32-bit “signed integer” é do tipo Int32.
- Em plataformas 32-bit “Int” tem o mesmo tamanho de Int32, já para plataformas 64-bit , Int tem o mesmo tamanho de Int64. O mesmo vale para UInt.

Tipo 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)
```

Saída:
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

```
let decimalInteger = 17

let binaryInteger = 0b10001 //17 in binary

let octalInteger = 0o21 // 17 in octal

let hexadecimalInteger = 0x11 //17 in

hexadecimal
```

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: Exemplos

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

Caracteres e Strings

- Strings Multilines(“ ” ”)
- Strings como vetor de caracteres
- Concatenação (+, +=) e comparação(==) de Strings
- Escapes (\\, \t, \0, \n, \r, \”, \’)
- Principais métodos: append(), count, isEmpty, startIndex, endIndex, index(before:), index(after:), index(_:offsetBy:), insert(_:at:), remove(at:)
- Uso de \ para alguns caracteres especiais na string

Caracteres e Strings: Exemplos

```
let someString = "Some string literal value"
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.
"""
let wiseWords = "\"Imagination is more important than
knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
```

Caracteres e Strings: Exemplos

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
// message is "3 times 2.5 is 7.5"
```

```
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
// G
var welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
// welcome now equals "hello!"
welcome.remove(at: welcome.index(before: welcome.endIndex))
// welcome now equals "hello"
```

Booleans

- Swift tem um tipo Booleano básico chamado Bool. Swift provém dois valores Boolean , true e false.

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

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: Exemplos

```
let shortForm: Int? = Int("42")
let longForm: Optional<Int> = Int("42")

let number: Int? = Optional.some(42)
let noNumber: Int? = Optional.none
print(noNumber == nil)
// Prints "true"

let number = Int("42")!
print(number)
// Prints "42”
```

Coleções: 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: Exemplos

```
var someInts = [Int]()

print("someInts is of type [Int] with \(someInts.count) items.")

// Prints "someInts is of type [Int] with 0 items."

someInts.append(3)
```

```
var frase = [String]()

frase = ["My", "mother", "was", "a", "tailor"]

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

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

Arrays: Exemplos

```
var threeDoubles = Array(repeating: 0.0, count:3)
// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]

var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]

var shoppingList: [String] = ["Eggs", "Milk"]
```

Coleções: 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”
- Principais métodos: count, isEmpty, updateValue(_:forKey:), removeValue(forKey:), values, keys

Dicionários: Exemplos

```
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)
```

Saída:
["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]

Coleções: 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()

Coleções: 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

Conjuntos: Exemplos

```
let oddDigits: Set = [1, 3, 5, 7, 9]

let evenDigits: Set = [0, 2, 4, 6, 8]

let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted() // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

oddDigits.intersection(evenDigits).sorted() // []

oddDigits.subtracting(singleDigitPrimeNumbers).sorted() // [1, 9]

oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted() // [1, 2, 9]
```

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
```

Enumerados: Exemplos

```
directionToHead = .Sul
switch directionToHead {
case .Norte:
    print("Lots of planets have a north")
case .Sul:
    print("Watch out for penguins")
case .Leste:
    print("Where the sun rises")
case .Oeste:
    print("Where the skies are blue")
} // Prints "Watch out for penguins"
```

```
enum Beverage: CaseIterable {
    case coffee, tea, juice
}
for beverage in Beverage.allCases {
    print(beverage)
}
// coffee
// tea
// juice
```

Variáveis e constantes

- Swift faz uso extensivo de constantes, e aqui elas são muito mais poderosas do que na linguagem C.
- Variáveis são representadas por var e constantes por let
- Variáveis são mutáveis, constantes não

```
let    pi    =    3.14

var    delta    =    7
```

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

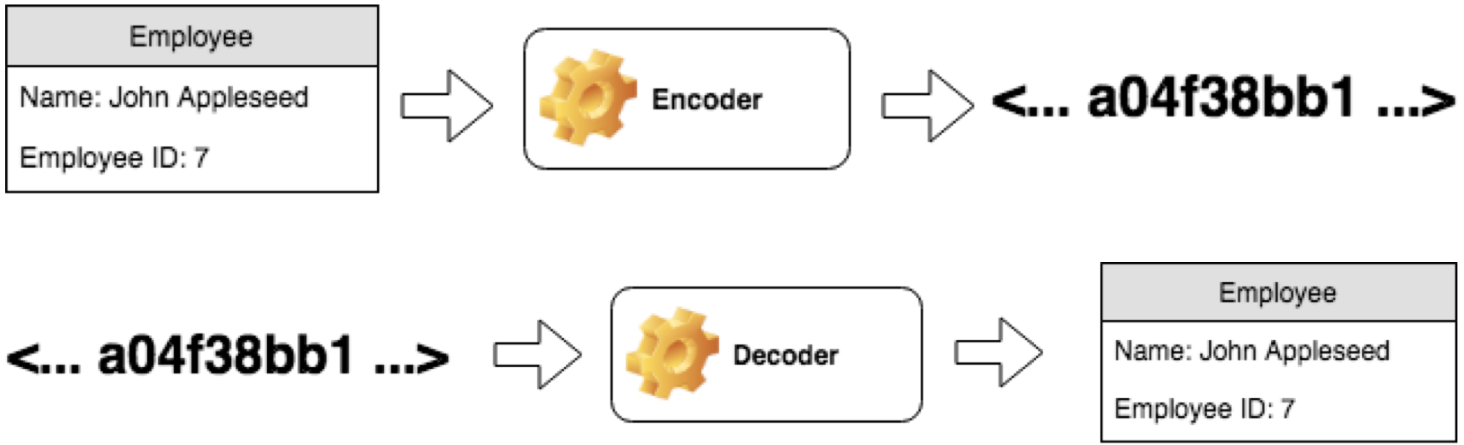
- Linha de comando
  - print() para output
  - readLine() para input
- Arquivos, utilizando FileManager do Foundation:

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

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

- Swift suporta a maioria das operações básicas de C e melhora vários recursos com o objetivo de eliminar erros comuns de codificação.

Em Swift, os operadores podem ser :

●Unitário (Pré-fixado ou Pós-fixado):
C! ou !B

●Binários: 1 + 2, são fixados entre dois alvos com espaço em branco.

● Ternário: Como em C, Swift tem apenas um operador ternário.
Condição de Operação: A ? B : C

Operador (=)

Inicializa ou atribui os valores dos operandos:

1. let b = 10
2. var a = 5
3. a = b
4. // a is now equal to 10

Como o operador (=) não retorna um valor, deve-se utilizar o operador de igualdade (==). Exemplo: if x == y.
Swift tem a facilidade de auxiliar nesse tipo de erro durante a confecção do programa.

Operadores Aritméticos

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

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

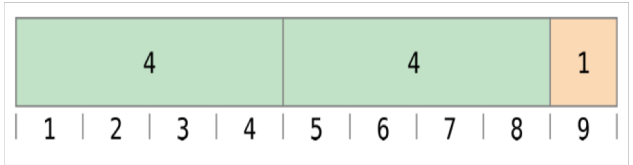
●Swift não suporta overflow por default. É possível suportar overflow através de operadores específicos de overflow.

Operadores Aritméticos

```
1. let b = 10
2. var a = 5
3. a = b
4. // a is now equal to 10
5. "hello, " + "world" // equals "hello, world"
```

Operador de Resto (%)

```
1. a % b
2. 9 % 4 // equals 1
```



- Retorna a quantidade de múltiplos, ou seja, a quantidade de “b” que cabem dentro de “a”, retornando o resto.
- Em outras linguagens esse operador, também e conhecido como modulo, porém em Swift ele não possui essa função.

Operador Menos (-)

```
1. let three = 3
2. let minusThree = -three
3. let plusThree = -minusThree
```

- Retorna a quantidade de múltiplos, ou seja, a quantidade de “b” que cabem dentro de “a”, retornando o resto.
- Em outras linguagens esse operador, também e conhecido como modulo, porém em Swift ele não possui essa função.

Operadores de Atribuição Composta

- Como em C, Swift permite combinar o operador (=) a outro operador.

1. `var a = 1`
2. `a += 2`

Como os operadores de atribuição composta NÃO RETORNAM UM VALOR, NÃO PODEMOS ESCREVER:

```
let b = a += 2
```

Operadores De Comparação

- Swift possui os seguintes operadores de comparação:

```
1. 1 == 1 // true because 1 is equal to 1
2. 2 != 1 // true because 2 is not equal to 1
3. 2 > 1 // true because 2 is greater than 1
4. 1 < 2 // true because 1 is less than 2
5. 1 >= 1 // true because 1 is greater than or equal to 1
6. 2 <= 1 // false because 2 is not less than or equal to 1
```

Operadores De Comparação

1. a === b **//São referências ao** mesmo **objeto.**
2. a !== b **//São referências a objetos** diferentes.

- Swift ainda possui os seguintes operadores de comparação.

Operadores De Comparação(Tuplas)

- Se a comparação entre os primeiros elementos da Tuplas for verdadeira, a comparação dos segundos elementos e ignorada.
- Swift, não compara tipos Boolean.

```
1. (1, "zebra") < (2, "apple") // true because 1 is less than 2; "zebra" and "apple" are not compared
```

Operador Ternário

- É um atalho para avaliar uma das duas expressões com base em se a pergunta é verdadeira ou falsa. Se a pergunta for verdadeira, ela avalia answer1 e retorna seu valor; caso contrário, ele avalia answer2 e retorna seu valor.

```
1. question ? answer1 : answer 2
2. let contentHeight = 40
3. let hasHeader = true
4. let rowHeight = contentHeight + (hasHeader ? 50 : 20) // rowHeight is equal to 90
```

Nil - Coalescing Operator

```
1. a != nil ? a! : b
2. //  a recebe a caso contenha
   valor. Se não, recebe b.
3. var nome: String?
4. nome != nil ? nome! : "Maria"
```

- O Nil Coalescing Operator "unwraps" um optional caso esta possua um valor ou retorna um valor padrão caso ela este não contenha.

Operadores de Intervalo (Range Operators)

- Swift possui vários operadores de intervalo, que são atalhos para expressar um intervalo de valores:

1. Operador de Intervalo Fechado;
2. Operador de Intervalo Meio - Aberto
3. Operador de Apenas um Lado.

Operadores de Intervalo Fechado (a ... b)

```
1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. let count = names.count
3. for i in 0..<count {
4. print("Person \(i + 1) is called \(names[i])")
5. }
6. // Person 1 is called Anna
7. // Person 2 is called Alex
8. // Person 3 is called Brian
9. // Person 4 is called Jack
```

Operadores de Apenas um Lado (a ...) ou (... b)

```
1.  for name in names[2...] {
2.      print(name)
3.  }
4.  // Brian
5.  // Jack
6.  for name in names[...2] {
7.      print(name)
8.  }
9.  // Anna
10. // Alex
11. // Brian
```

Operadores de Intervalo Fechado (a ..<b)

```
1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. let count = names.count
3. for i in 0..<count {
4. print("Person \(i + 1) is called \(names[i])")
5. }
6. // Person 1 is called Anna
7. // Person 2 is called Alex
8. // Person 3 is called Brian
9. // Person 4 is called Jack
```

Operadores Lógicos

- Logical NOT (!a)
- Logical AND (a && b)
- Logical OR (a || b)

- Os operadores lógicos modificam ou combinam os valores lógicos booleanos como verdadeiro e falso. O Swift suporta os três operadores lógicos padrão encontrados em linguagens baseadas em C.

Operadores Lógicos

- Os operadores lógicos Swift && e || são associativas à esquerda, o que significa que expressões compostas com múltiplos operadores lógicos avaliam primeiro a subexpressão mais à esquerda.

```
1. if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword
2. {
3. print("Welcome!")
4. }
5. else {
6. print("ACCESS DENIED")
7. }
8. // Prints "Welcome!"
```

Controle de Fluxo (looping)

- A maioria das linguagens de programação possui várias estruturas de controle de fluxo, que também são conhecidas como *loop*, *looping*, laço de repetição ou estrutura de repetição. Na Swift não seria diferente. Vamos conhecer então as principais estruturas de controle de fluxo.
    1. For - in
    2. While
    3. Repeat ~ While

Operador For - In

- Você usa o loop de entrada para iterar em uma sequência, como itens em uma matriz, intervalos de números ou caracteres em uma sequência.

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
print("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

Operador For - In com Tuplas

- Você também pode iterar em um dicionário para acessar seus pares de valores-chave. Cada item no dicionário é retornado como uma tupla (chave, valor) quando o dicionário é iterado e você pode decompor os membros da tupla (chave, valor) como constantes explicitamente nomeadas para uso dentro do corpo do loop de entrada.

```
1. let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
2. for (animalName, legCount) in numberOfLegs {
3. print("\(animalName)s have \(legCount) legs")
4. }
5. // ants have 6 legs
6. // cats have 4 legs
7. // spiders have 8 legs
```

Operador While

- Um loop While inicia avaliando uma única condição. Se a condição for verdadeira ele inicia até que se torne falsa.

```
1. while condition {
2. statements
3. }
```

Operador Repeat - While

- Analogo a do - while em C:

```
1. repeat {
2. statements
3. } while condition
```

Operadores If - else , Switch , Continue, Break

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

```
1. let umCaractere: Character = "z"
2. switch umCaractere {
3. case "z":
4. print("final do alfabeto")
5. default:
6. print("nao sei") }
```

```
1. var presente = 10
2. while presente > 1 {
3. if presente == 5 {
4. continue
5. }
6. presente -= 1
7. }
```

```
1. fallthrough
```

```
1. var presente = 10
2. while presente > 1 {
3. if presente == 5 {
4. break
5. }
6. presente -= 1
7. }
```

Funções

Funções

- Estrutura Básica:

```
1. func cumprimenta (pessoa: String) -> String {
2. let cumprimento = "Olá, " + pessoa + "!"
3. return cumprimento
4. }
```

Funções

- Manipulação de Retornos - Tuplas:

```
1.  func minMax(array: [Int]) -> (min: Int, max: Int)? {
2.  if array.isEmpty { return nil }
3.  var currentMin = array[0]
4.  var currentMax = array[0]
5.  for value in array[1..<array.count] {
6.  if value < currentMin {
7.  currentMin = value
8.  } else if value > currentMax {
9.  currentMax = value
10. }
11. }
12. return (currentMin, currentMax)
13. }
```

Funções

- Manipulação de Retornos::

```
1. if let extremos = minMax(array: [8, -6, 2, 109, 3, 71]) {
2. print("min é \(extremos.min) e max é \(extremos.max)") //
3. Imprime “min é -6 e max é 109”
4. }
```

Funções - Rótulo de argumento vs Nome de

- 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

```
1. func qualquerCoisa(rotuloArgumentoUm nomeParametroUm: Int,
   rotuloArgumentoDois
2. nomeParametroDois: Int) -> Int {
3. let soma = nomeParametroUm + nomeParametroDois
4. return soma
5. }
6. let soma = qualquerCoisa(rotuloArgumentoUm: 1, rotuloArgumentoDois: 2)
```

Funções - Rótulo de argumento vs Nome de

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

```
1. func qualquerCoisa(_ nomeParametroUm: Int, nomeParametroDois: Int) -> Int {
2. let soma = nomeParametroUm + nomeParametroDois
3. return soma
4. }
5. let soma = qualquerCoisa(1, nomeParametroDois: 2)
```

Funções - Parâmetros Default

```
1. func qualquerCoisa(nomeParametroUm: Int, nomeParametroDois: Int = 13) ->
   Int
2. {
3. let soma = nomeParametroUm + nomeParametroDois
4. return soma
5. }
6. 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

```
1. func totalAritmetico(_ numbers: Double...) -> Double {
2. var total: Double = 0
3. for number in numbers {
4. total += number
5. }
6. return total / Double(numbers.count)
7. }
8. let x = totalAritmetico(1, 2, 3, 4, 5) // x recebe 3.0
9. 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 compilação
- Tara modificar parâmetros (e fazer com que a mudança permaneça fora da função), definir eles como in-out
- Parâmetros in-out devem ser variáveis, não podem ser variantes e não aceitam valores default

```
1. func swapTwoInts(_ a: inout Int, _ b: inout Int) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }
```

Funções - Tipos

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

```
1. func addTwoInts(_ a: Int, _ b: Int) -> Int { // tipo: (Int, Int) -> Int
2. return a + b
3. }
```

Funções - Tipos

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

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

Funções - Tipos

- Estes tipos também podem ser passados como parâmetros de funções

```
1. func addTwoInts(_ a: Int, _ b: Int) -> Int {
2. return a + b
3. }
4. func imprimeResultado(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
5. print("Resultado: \(mathFunction(a, b))")
6. }
7. imprimeResultado(addTwoInts, 3, 5) // imprime “Resultado: 8”
```

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 Aninhadas

```
1.  func opera(com simbolo:String) -> (Int, Int) -> Int {
2.  func adicao(num1:Int, num2:Int) -> Int {
3.  return num1 + num2
4.  }
5.  func subtracao(num1:Int, num2:Int) -> Int {
6.  return num1 - num2
7.  }
8.  let operacao = (simbolo == "+") ? adicao : subtracao
9.  return operacao
10. }
11. let operacao = opera(com: "+")
12. let resultado = operacao( 2, 3)
13. print(resultado) // imprime 5
```

Modularização

Classes e Estruturas

Classes possuem capacidades adicionais que Structs não possuem :

- Herença;
- TypeCast permite a identificação do tipo de uma classe;
- 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
}
```

Estruturas

```
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
}
```

Classes

```
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
        }
    }
```

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,enumerados, 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
```

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)

Módulos

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")
```

Polimorfismo

Polimorfismo

1. Sistemas de tipos (verificação, inferência, conversão de tipos);
2. Sistemas de tipos monomórficos e polimórficos;
3. Tipos de polimorfismo: coerção, sobrecarga, paramétrico e inclusão;
4. Herança, sobrescrita, amarração tardia, classes e métodos abstratos;
5. Metaclasses;

Polimorfismo

- *Tipagem forte e estática*
- *Type-safe*
- *Inferência de tipo*
    - *Operador* ***is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

Polimorfismo

- *Tipagem forte e estática* ✓
- *Type-safe*
- *Inferência de tipo*
    - *Operador* ***is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

Polimorfismo

- *Tipagem forte e estática* ✓
- Type-safe
- *Inferência de tipo*
    - *Operador* ***is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

Polimorfismo

- *Tipagem forte e estática* ✓
- Type-safe ✓
- *Inferência de tipo*
    - *Operador* ***is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

Polimorfismo

- *Tipagem forte e estática* ✓
- *Type-safe* ✓
- *Inferência de tipo*
    - *Operador* ***is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

```
var a = 123
```

Polimorfismo

- *Tipagem forte e estática ✓*
- *Type-safe ✓*
- *Inferência de tipo*
    - *Operador **is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do **as**)*
        - *Type Casting para **Any** e **AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

```
var a = 123
```

Infere tipo com base no contexto no qual o elemento é definido

Polimorfismo

- *Tipagem forte e estática ✓*
- *Type-safe ✓*
- *Inferência de tipo*
    - *Operador **is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do **as**)*
        - *Type Casting para **Any** e **AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

```
var a = 123
var a: Int = 123
```

Tipo definido explícitamente

Polimorfismo

- *Tipagem forte e estática ✓*
- *Type-safe ✓*
- *Inferência de tipo*
    - *Operador **is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do **as**)*
        - *Type Casting para **Any** e **AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

```
var a = 123
```

```
var a: Int = 123
```

```
var a
```

error: type annotation missing in pattern

Polimorfismo

- *Tipagem forte e estática ✓*
- *Type-safe ✓*
- *Inferência de tipo*
    - *Operador **is***
- *Conversão explícita de tipo:*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do **as**)*
        - *Type Casting para **Any** e **AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

```
var a = 123
```

```
var a: Int = 123
```

```
var a: Int
```

Compila!

Polimorfismo

Exemplo:

```
class MediaItem {
	var name: String
	init(name: String) {
		self.name = name
	}
}
```

```
class Movie: MediaItem {
	var director: String
	init(name: String, director: String) {
		self.director = director
		super.init(name: name)
	}
}
class Song: MediaItem {
	var artist: String
	init(name: String, artist: String) {
		self.artist = artist
		super.init(name: name)
	}
}
```

Polimorfismo

Exemplo:

```
class MediaItem {
        var name: String
        init(name: String) {
                self.name = name
        }
}
```

Classe MediaItem

```
class Movie: MediaItem {
        var director: String
        init(name: String, director: String) {
                self.director = director
                super.init(name: name)
        }
}
class Song: MediaItem {
        var artist: String
        init(name: String, artist: String) {
                self.artist = artist
                super.init(name: name)
        }
}
```

Subclasses de MediaItem

Polimorfismo

Exemplo:

```
let library: [
        Movie(name: “Casablanca”, director: “Michael Curtiz”),
        Song(name: “Blue Suede Shoes”, artist: “Elvis Presley”),
        Movie(name: “Citizen Kane”, director: “Orson Welles”)
        ]
```

O tipo de “library” é inferido pela inicialização com o conteúdo

Polimorfismo

Exemplo:

```
let library: [
        Movie(name: "Casablanca", director: "Michael Curtiz"),
        Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
        Movie(name: "Citizen Kane", director: "Orson Welles")
        ]
```

O verificador de tipos de Swift é capaz de deduzir que "Movie" e "Song" tem uma superclasse comum de MediaItem

O tipo de "library" é inferido pela inicialização com o conteúdo

Polimorfismo

Exemplo:

```
let library: [
        Movie(name: "Casablanca", director: "Michael Curtiz"),
        Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
        Movie(name: "Citizen Kane", director: "Orson Welles")
        ]
```

O verificador de tipos de Swift é capaz de deduzir que "Movie" e "Song" tem uma superclasse comum de MediaItem

O tipo de "library" é inferido pela inicialização com o conteúdo

Para trabalhar com o tipo nativo, você precisa verificar seu tipo ou reduzi-los (downcast) para um tipo diferente.

Polimorfismo

Inferência de Tipos: Operador ***is***

```
var movieCount = 0
var songCount = 0

for item in library {
   if item is Movie {
       movieCount += 
1
   } else if item is 
Song {
       songCount += 1
   }
}
```

Use “is” para verificar se uma instância é de um determinado tipo de subclasse.

Retorna valores booleanos: true e false.

Polimorfismo

Inferência de Tipos: Operador ***is***

```
var movieCount = 0
var songCount = 0

for item in library {
   if item is Movie {
       movieCount +=
1
   } else if item is
Song {
       songCount += 1
   }
}
```

Use “is” para verificar se uma instância é de um determinado tipo de subclasse.

Retorna valores booleanos: true e false.

Mas ainda não consigo alterar ou acessar valores de item

Polimorfismo

- *Tipagem forte e estática* ✓
- *Type-safe* ✓
- *Inferência de tipo* ✓
    - *Operador* ***is*** ✓
- Conversão explícita de tipo:
    - Type Casting:
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores* ***as***, ***as?*** *e* ***as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores* ***as***, ***as?*** *e* ***as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

O upcasting (ou ampliação) sempre será legal.

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores* **as**, **as?** *e* **as!**

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

as:

quando você sabe que é uma instância de uma subclasse X.

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores **as**, **as?** e **as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

as:
quando você sabe que pode

as?
quando você **não** sabe se é uma instância de uma subclasse X. *

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores **as**, **as?** e **as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

as:
quando você sabe que pode

as?
quando você **não** sabe se é uma instância de uma subclasse X. *

Tenta o downcast. Se não conseguir, retorna ***nil***.

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores* ***as***, ***as?*** *e* ***as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

as:
quando você sabe que pode

as?
quando você **não** sabe se é uma

as!
Forma forçada do operador; somente quando tiver certeza. *

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores **as**, **as?** e **as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

as:
quando você sabe que pode

as?
quando você **não** sabe se é uma

as!
Forma forçada do operador; somente quando tiver certeza. *

Se o downcast falhar, dispara *runtime error.*

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores **as**, **as?** e **as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

as:
quando você sabe que pode

as?
quando você **não** sabe se é uma

as!
Forma forçada do operador; somente quando tiver certeza. *

O casting não modifica a instância nem altera seus valores!

Polimorfismo

Type Casting: *Downcasts e Upcasts*

- *Uso dos operadores* ***as***, ***as?*** *e* ***as!***

```
for item in library {
        if let movie = item as? Movie {
                print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
                print("Song: \(song.name), by \(song.artist)")
        }
}
```

```
//Movie: Casablanca, dir. Michael Curtiz
//Song: Blue Suede Shoes, by Elvis Presley
//Movie: Citizen Kane, dir. Orson Welles
```

as:
quando você sabe que pode

as?
quando você **não** sabe se é uma

as!
Forma forçada do operador; somente quando tiver certeza. *

Polimorfismo

Type Casting para **Any** e **AnyObject**

Swift fornece dois tipos especiais para trabalhar com tipos não específicos:

- **Any** pode representar uma instância de qualquer tipo, incluindo tipos de função;
- **AnyObject** pode representar uma instância de qualquer tipo de classe;

```
var things = [Any] ()
            things.append(0)
            things.append(0.0)
            things.append("hello")
            things.append(3.0, 5.0)
            things.append(Movie(name:"Ghostbusters", director: "Ivan Reitman"))
            things.append({(name: String) -> String in "Hello, \(name)"})
```

Polimorfismo

Type Casting para **Any** e **AnyObject**

Swift fornece dois tipos especiais para trabalhar com tipos não específicos:

- **Any** pode representar uma instância de qualquer tipo, incluindo tipos de função;
- **AnyObject** pode representar uma instância de qualquer tipo de classe;

```
var things = [Any] ()
things.append(0)
things.append(0.0)
things.append("hello")
things.append(3.0, 5.0)
things.append(Movie(name:"Ghostbusters", director: "Ivan Reitman"))
things.append({(name: String) -> String in "Hello, \(name)"})
```

Array só aceitam um mesmo tipo.

Polimorfismo

Type Casting para **Any** e **AnyObject**

Para saber o tipo específico de uma constante ou variável, podemos usar o **is** ou o **as** nos **cases** de switch.

```
for thing in things {
        switch thing {
                case 0 as Int:
                        //...
                case 0 as Double:
                        //...
                case someString as String:
                        //..
        ...
```

Any representa valores de qualquer tipo, incluindo tipo Optional.

Polimorfismo

Type Casting para **Any** e **AnyObject**

Para saber o tipo específico de uma constante ou variável, podemos usar o **is** ou o **as** nos **cases** de switch.

```
for thing in things {
        switch thing {
                case 0 as Int:
                        //...
                case 0 as Double:
                        //...
                case someString as String:
                        //..
        ...
```

Any representa valores de qualquer tipo, incluindo tipo Optional.

Swift dá um aviso quando você usar um valor Optional quando espera um valor Any.

Polimorfismo

Type Casting para **Any** e **AnyObject**

Para saber o tipo específico de uma constante ou variável, podemos usar o **is** ou o **as** nos **cases** de switch.

```
for thing in things {
        switch thing {
                case 0 as Int:
                        //...
                case 0 as Double:
                        //...
                case someString as String:
                        //..
        ...
```

Any representa valores de qualquer tipo, incluindo tipo Optional.

Swift dá um aviso quando você usar um valor Optional quando espera um valor Any.

Os tipos do Array são Any e não *int, double, string…*

Polimorfismo

Type Casting para **Any** e **AnyObject**

Se você realmente precisar de um valor Optional com Any, pode-se usar a conversão explícita com o operador **as**:

```
let optionalNumber: Int? = 3
        things.append(optionalNumber) // Warning
        things.append(optionalNumber as Any) // No Warning
```

Polimorfismo

- *Tipagem forte e estática* ✓
- *Type-safe* ✓
- *Inferência de tipo* ✓
    - *Operador* ***is***
- *Conversão explícita de tipo:* ✓
    - *Type Casting:*
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos*
- *Sobrescrita*
- *Herança*

Polimorfismo

Tipos de Polimorfismos

Coerção:

Sobrecarga:

Paramétrico:

Inclusão:

Polimorfismo

Tipos de Polimorfismos

Coerção:

Sobrecarga:

Paramétrico:

Inclusão:

Não!
Not!
Nein!
Non!

Polimorfismo

Tipos de Polimorfismos

Coerção:

Não!
Not!
Nein!
Non!

Sobrecarga:

Paramétrico:

```
let pi = 3.14 //Constante do tipo Double
pi = 3.1 // error: cannot assign to value: 'pi' is a 'let' constant
```

Inclusão:

```
var pi = 3.14
let hello = “hello” //Constante do tipo String
pi = hello // error: cannot assign value of type 'String' to type 'Double'
```

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga:

Sim!

Paramétrico:

Inclusão:

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga:

Paramétrico:

Inclusão:

O operador de adição aritmética é um operador binário e não sabe/sabia "somar vetores".

```
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)
      }
}
```

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico:

Inclusão:

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico:

Sim!

Inclusão:

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico:

Inclusão:

Generics

funções

```
func imprimeElementos<T>(a: [T]) {
    for elemento in a {
        println(elemento)
    }
}
```

Placeholders: seu uso indica que ele será substituído por um tipo real

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico:

Inclusão:

Generics

funções

```
func minhaFuncao<T, U>(a: T, b: U) {}
```

Mais de um Generic

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico:

Inclusão:

Generics

tipos

```
struct MinhaColecao<T> {
    let itens: [T]

    init(itens: [T]) {
        self.itens = itens
    }
            //...
}
```

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico: Sim!

Inclusão:

Sim!

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico: Sim!

Inclusão:

```
class Vehicle {
   var currentSpeed = 0.0
   var description: String {
      return "traveling at \(currentSpeed) miles per hour"
   }
            func makeNoise() {  //do nothing }
      //...
}

class Bicycle: Vehicle {    //defines a subclass called Bicycle,
   var hasBasket = false  // with a superclass of Vehicle
}

let bicycle = Bicycle() //
bicycle.hasBasket = true

bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

Polimorfismo

Tipos de Polimorfismos

Coerção: Não!

Sobrecarga: Sim!

Paramétrico: Sim!

Inclusão: Sim!

Polimorfismo

- *Tipagem forte e estática ✓*
- *Type-safe ✓*
- *Inferência de tipo ✓*
    - *Operador **is***
- *Conversão explícita de tipo: ✓*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do **as**)*
        - *Type Casting para **Any** e **AnyObject***
- *Tipos de Polimorfismos ✓*
- *Sobrescrita*
- *Herança*

Polimorfismo

- *Tipagem forte e estática ✓*
- *Type-safe ✓*
- *Inferência de tipo ✓*
    - *Operador **is***
- *Conversão explícita de tipo: ✓*
    - *Type Casting:*
        - *Downcasts e upcasts (uso do **as**)*
        - *Type Casting para **Any** e **AnyObject***
- *Tipos de Polimorfismos ✓*
- Sobrescrita
- *Herança*

Outra maneira de fazer polimorfismo de sobrecarga

Polimorfismo

Sobrescrita

Para substituir uma característica que seria herdada, você pré-fixará sua definição de substituição com a palavra-chave **override:**

```
class Train: Vehicle {
   override func makeNoise() {
      print("Choo Choo")
   }
}
```

Na superclasse, "makeNoise" não fazia nada.

Se você criar uma nova instância *Train* e chamar o método *makeNoise()*, verá que a versão da subclasse *Train* do método é chamada:

```
let train = Train()
train.makeNoise() // Prints "Choo Choo"
```

Polimorfismo

Sobrescrita

```
class Car: Vehicle {
        var gear = 1
        override func description: String {
                        return super.description + "in gear
\(gear)”
    }
}
```

Polimorfismo

Sobrescrita

```
class Car: Vehicle {
        var gear = 1
        override func description: String {
                        return super.description + "in gear
\(gear)”
    }
}
```

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3


print(“Car:
\(car.decription)”)
```

Polimorfismo

Sobrescrita

```
class Car: Vehicle {
        var gear = 1
        override func description: String {
                        return super.description + "in gear
\(gear)”
    }
}
```

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3


print(“Car:
\(car.decription)”)
```

```
// Car: travelling at 25.0 miles per hour in gear 3
```

Polimorfismo

Sobrescrita

- *Prevenindo* overrides: **final**

Polimorfismo

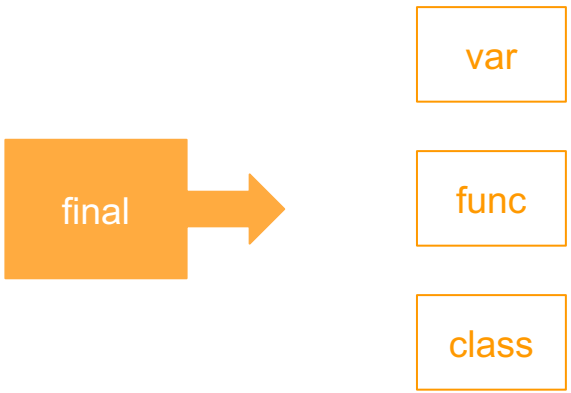
Sobrescrita

- *Prevenindo* overrides: **final**

Polimorfismo

Sobrescrita

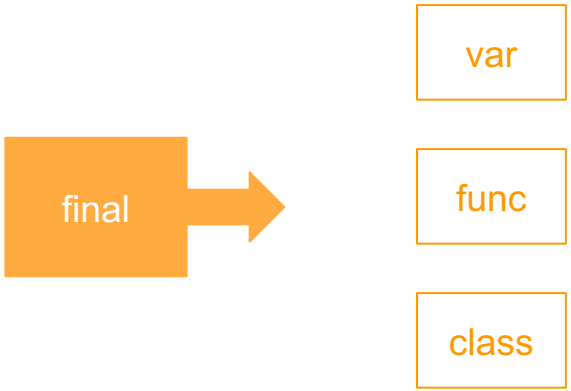
- *Prevenindo* overrides: **final**



Polimorfismo

Sobrescrita

- *Prevenindo* overrides: **final**



Qualquer tentativa de substituir um método/propriedade/… ***final***, gerará um *Compile-time Error.*

Polimorfismo

- *Tipagem forte e estática* ✓
- *Type-safe* ✓
- *Inferência de tipo* ✓
    - *Operador* ***is***
- *Conversão explícita de tipo:* ✓
    - *Type Casting:*
        - *Downcasts e upcasts (uso do* ***as****)*
        - *Type Casting para* ***Any*** *e* ***AnyObject***
- *Tipos de Polimorfismos* ✓
- *Sobrescrita* ✓
- *Herança*

Polimorfismo

Herança

- Não possui herança múltipla;

Polimorfismo

Herança

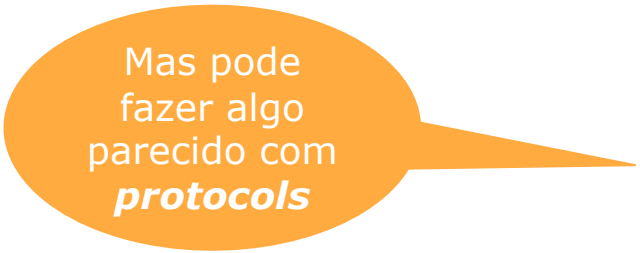
- Não possui herança múltipla;

Mas pode fazer algo parecido com ***protocols***

Polimorfismo

Herança

- Não possui herança múltipla;




```
protocol A {
    func aFunctionName()
}

protocol B {
    func bFunctionName()
}

class someClass: A, B {
                //...
}
```

Polimorfismo

* Metaclasse

Em Swift, uma classe é em si um objeto que existe em tempo de execução. O objeto de classe é uma instância de uma **metaclasse**

Polimorfismo

* ***Metaclasses***

Um "class object" é uma "type instance".

"Meta Class" é um Meta Type".

Polimorfismo

* ***Metaclasses***

Um "class object" é uma "type instance".

"Meta Class" é um Meta Type".

O tipo `Foo` possui o metatipo chamado `Foo.Type`.

Exceções

Exceções

* Objective-C e o ***NSError***

O tratamento de erros no Swift interopera com padrões de manipulação de erros que usam a classe ***NSError*** no Objective-C.

Exceções

* Objective-C e o ***NSError***

O tratamento de erros no Swift interopera com padrões de manipulação de erros que usam a classe ***NSError*** no Objective-C.

Error é um protocolo

Exceções

* Objective-C e o ***NSError***

O tratamento de erros no Swift interopera com padrões de manipulação de erros que usam a classe ***NSError*** no Objective-C.

Error é um protocolo

Um *protocolo* define um esquema de métodos, propriedades e outros requisitos que se adequam a uma determinada tarefa ou funcionalidade.

Exceções

Exemplo:

```
protocol  Error {

}
```

Protocolo vazio ***Error***.

Exceções

Exemplo:

```
enum FileError: Error {
        case notFound
        case insufficientLines(linesNeeded: Int)
}
```

Exceções

Exemplo:

```
enum FileError: Error {
        case notFound
        case insufficientLines(linesNeeded: Int)
}
```

- **Throw:**

Exceções

Exemplo:

```
enum FileError: Error {
        case notFound
        case insufficientLines(linesNeeded: Int)
}
```

- **Throw:**

```
throw FileError.insufficientLines(linesNeeded: 1000)
```

Exceções

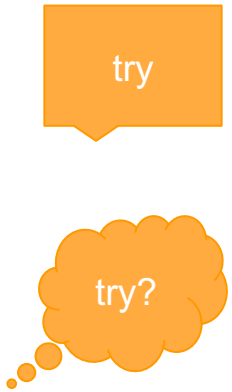
Manipulando Erros

Exceções

Manipulando Erros

Exceções

Manipulando Erros



Exceções

Manipulando Erros

Exceções

Manipulando Erros

Ao contrário do tratamento de exceções em muitas linguagens, o tratamento de erros no Swift não envolve o desenrolar da pilha de chamadas, um processo que pode ser computacionalmente caro.

Exceções

Manipulando Erros

Há quatro formas de lidar com erros lançados:

1. Propagando o erro;
2. Manipulando o erro usando uma instrução *do-catch*;
3. Manipulando o erro como um valor optional; ou
4. Afirmando que o erro não ocorrerá.

Exceções

Manipulando Erros

Há quatro formas de lidar com erros lançados:

1. Propagando o erro;
2. Manipulando o erro usando uma instrução *do-catch*;
3. Manipulando o erro como um valor optional; ou
4. Afirmando que o erro não ocorrerá.

Exceções

1. Propagando o erro:

- Uso do “throw”
- Propaga erros que são lançados dentro dele para o escopo do qual é chamado.

```
func canThrowErrors() throws -> String

func cannotThrowErros() -> String
```

Exceções

1. Propagando o erro:

- Uso do “throw”
- Propaga erros que são lançados dentro dele para o escopo do qual é chamado.

```
func canThrowErrors() throws -> String

func cannotThrowErros() -> String
```

Erros lançados dentro de uma função sem ***throw*** devem ser manipulados dentro da função.

Exceções

1. Propagando o erro:

- Instrução **guard**

```
enum VendingMachineError: Error {

    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock

}
```

Exceções

1. Propagando o erro:

- Instrução **guard**

```
struct Item {
  var price: Int
  var count: Int
}

class VendingMachine {
  var inventory = [
    "Candy Bar": Item(price: 12, count: 7),
    "Chips": Item(price: 10, count: 4),
    "Pretzels": Item(price: 7, count: 11)
  ]
  var coinsDeposited = 0
```

Exceções

1. Propagando o erro:

- Instrução **guard**

```
struct Item {
   var price: Int
   var count: Int
}

class VendingMachine {
   var inventory = [
      "Candy Bar": Item(price: 12, count: 7),
      "Chips": Item(price: 10, count: 4),
      "Pretzels": Item(price: 7, count: 11)
   ]
   var coinsDeposited = 0
```

```
func vend(itemNamed name: String) throws {
      guard let item = inventory[name] else {
         throw
            VendingMachineError.invalidSelection
      }

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

            //...

      print("Dispensing \(name)")
   }
}
```

Exceções

Manipulando Erros

Há quatro formas de lidar com erros lançados:

1. Propagando o erro; ✓
2. Manipulando o erro usando uma instrução *do-catch*;
3. Manipulando o erro como um valor optional; ou
4. Afirmando que o erro não ocorrerá.

Exceções

2. Manipulando o erro usando uma instrução do-catch:

```
do {
        try expression
            statements
        } catch pattern 1 {
            statements
} catch pattern 2 where condition {
        statements
} catch {
        statements
}
```

Exceções

2. Manipulando o erro usando uma instrução do-catch:

```
do {
        try expression
            statements
        } catch pattern 1 {
            statements
} catch pattern 2 where condition {
        statements
} catch {
        statements
}
```

Pattern para dizer quais erros a cláusula catch pode manipular

Exceções

2. Manipulando o erro usando uma instrução do-catch:

```
do {
        try expression
              statements
        } catch pattern 1 {
              statements
} catch pattern 2 where condition {
        statements
} catch {
        statements
}
```

Um *catch* sem um *pattern*, a cláusula corresponderá a qualquer erro...

Exceções

2. Manipulando o erro usando uma instrução do-catch:

```
do {
        try expression
            statements
        } catch pattern 1 {
            statements
} catch pattern 2 where condition {
        statements
} catch {
        statements
}
```

Um *catch* sem um *pattern*, a cláusula corresponderá a qualquer erro...

...e ligará o erro a uma constante local nomeada *error*.

Exceções

Manipulando Erros

Há quatro formas de lidar com erros lançados:

1. Propagando o erro; ✓
2. Manipulando o erro usando uma instrução *do-catch*; ✓
3. Manipulando o erro como um valor Optional; ou
4. Afirmando que o erro não ocorrerá.

Exceções

3. Manipulando o erro como um valor *Optional*:

```
func someThrowingFunction() throws -> Int {

    // ...
}
let x = try? someThrowingFunction()
let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

Exceções

Manipulando Erros

Há quatro formas de lidar com erros lançados:

1. Propagando o erro; ✓
2. Manipulando o erro usando uma instrução *do-catch*; ✓
3. Manipulando o erro como um valor optional; ou ✓
4. Afirmando que o erro não ocorrerá.

Exceções

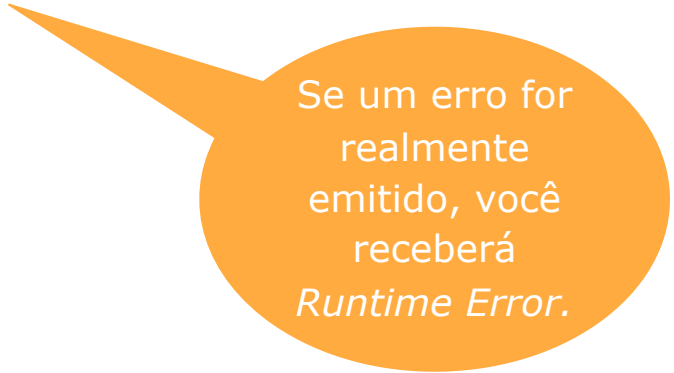
4. Afirmar que o erro não ocorrerá:

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Exceções

4. Afirmar que o erro não ocorrerá:

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```



Exceções

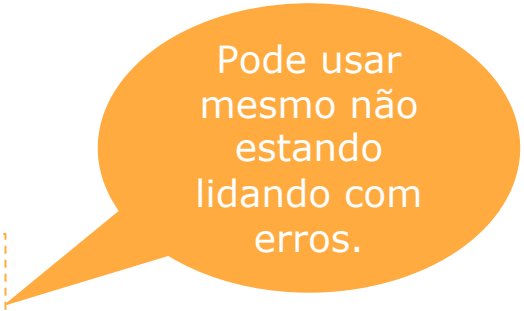
O uso do ***defer***:

```
func processFile(filename: String) throws {
	if exists(filename) {
		let file = open(filename)
		defer {
			close(file)
		}
		while let line = try file.readline() {
			// Work with the file.
		}
		// close(file) is called here, at the end of the
	scope.
	}
}
```

Exceções

O uso do ***defer***:

```
func processFile(filename: String) throws {
	if exists(filename) {
		let file = open(filename)
		defer {
			close(file)
		}
		while let line = try file.readline() {
			// Work with the file.
		}
		// close(file) is called here, at the end of the
	scope.
	}
}
```



Concorrência

Concorrência

- Swift 4.0 não oferece nenhuma funcionalidade nativa de concorrência, com isso é necessário utilizar de bibliotecas externas;
- Já foi anunciado para a próxima atualização da LP uma expansão de funcionalidades nativas que incluirão modelos de memória e concorrência.

Concorrência

Processos e threads

- A framework Foundation oferece uma classe Thread que é internamente baseada na pthread.
- As threads podem ser criadas declarando uma classe thread personalizada e então realizar um override na main();

```
class MyThread : Thread {
    override func main(){
        print("A Thread começou a dormir por 2 segundos...")
        Thread.sleep(forTimeInterval:2)

        print("Acordou!! ")
    }
}
```

Concorrência

- Entretanto, desde o iOS10 e macOS Sierra é possível, em todas as plataformas, realizar a criação de uma nova Thread usando o inicializador que permite especificar a closure que a thread irá executar.

```
var t = Thread {
    print("Começou!")
}

t.start()
```

Concorrência

Semáforos

- A framework Foundation fornece todas as funcionalidades básicas para a sincronização de threads.
- São eles: NSLock, NSRecursiveLock, NSConditionLock, NSCondition e NSDistributedLock

Concorrência

NSLock:

- Tipo básico de lock;
- Quando uma thread tenta bloquear um objeto, duas coisas podem acontecer: a thread adquirirá o bloqueio e prosseguirá se ainda não tiver sido adquirido por um encadeamento anterior ou, alternativamente, a thread irá esperar, bloqueando sua execução, até que o dono do lock desbloqueie-a.

Concorrência

```
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)+" adquiriu o lock.")
        lock.unlock()
        if lock.try() {
            print(String(id)+" adquiriu o lock
novamente.")
            lock.unlock()
        }
```

```
else{  // Se já está em lock, siga em frente.
        print(String(id)+" não pode adquirir o
lock.")
                }
         print(String(id)+" saindo.")
    }
}
var t1 = LThread(id:1)
var t2 = LThread(id:2)
t1.start()
t2.start()
```

Concorrência

NSRecursiveLock:

- Locks recursivos podem ser adquiridos múltiplas vezes pela thread que já possui aquele lock. Muito útil em funções recursivas ou quando são chamadas múltiplas funções que checam o mesmo lock em sequência.

```
let rlock = NSRecursiveLock()
class RThread : Thread {
    override func main(){
        rlock.lock()
        print("Thread adquiriu o lock")
        callMe()
        rlock.unlock()
        print("Saindo da main")
    }
    func callMe(){
        rlock.lock()
        print("Thread adquiriu o lock")
        rlock.unlock()
        print("Saindo do método callMe")
    }
}
var tr = RThread()
tr.start()
```

Concorrência

NSConditionLock:

- Locks de condição oferecem sub-locks adicionais que podem ser bloqueados/desbloqueados independentemente um do outro para apoiar configurações de locking mais complexas (por exemplo, cenário de produtor-consumidor).

Concorrência

```
// PRODUTOR:
let NO_DATA = 1
let GOT_DATA = 2
let clock = NSConditionLock(condition: NO_DATA)
var SharedInt = 0
class ProducerThread : Thread {
    override func main(){
        for i in 0..<5 {
            clock.lock(whenCondition: NO_DATA) //Adquire o lock quando: NO_DATA
            // Se não temos que esperar pelos consumidores, poderíamos ter acabado de
fazer clock.lock()
            SharedInt = i
            clock.unlock(withCondition: GOT_DATA) //Unlock e define como GOT_DATA
        }
    }
}
```

Concorrência

```
// CONSUMIDOR
class ConsumerThread : Thread {
    override func main(){
        for i in 0..<5 {
            clock.lock(whenCondition: GOT_DATA) //Adquire o lock quando: GOT_DATA
            print(i)
            clock.unlock(withCondition: NO_DATA) //Unlock e define como NO_DATA
        }
    }
}
let pt = ProducerThread()
let ct = ConsumerThread()
ct.start()
pt.start()
```

Concorrência

NSCondition:

- Uma NSCondition oferece uma maneira limpa de esperar pela ocorrência de uma condição.
- Quando a thread que obteve o lock verifica que uma condição adicional necessária para executar sua função não é atendida, ele precisa ser colocado em espera e continuar seu trabalho quando essa condição for atendida.

Concorrência

```
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() // Notifica e acorda as
threds em espera
            cond.unlock()
        }
    }
}
```

```
class PrinterThread : Thread {
    override func main(){
        for _ in 0..<5 { //Realiza apenas 5 vezes
            cond.lock()
            while(!available){ //Proteger contra
sinais espúrios
                cond.wait()
            }
            print(SharedString)
            SharedString = ""
            available = false
            cond.unlock()
        }
    }
}
let writet = WriterThread()
let printt = PrinterThread()
printt.start()
writet.start()
```

Concorrência

NSDistributedLock:

- Eles são feitos para serem compartilhados entre vários aplicativos e são apoiados por uma entrada no sistema de arquivos.
- Esse tipo de lock é adquirido pelo uso do método try(), um método não bloqueável que retorna imediatamente com um booleano indicando se o bloqueio foi adquirido ou não. A aquisição de um bloqueio geralmente requer mais de uma tentativa, para ser executada manualmente e com um atraso adequado entre as tentativas sucessivas.
- Locks distribuídos são liberados com o uso do método unlock().

Concorrência

```
var dlock = NSDistributedLock(path: "/tmp/MYAPP.lock")
if let dlock = dlock {
    var acquired = false
    while(!acquired){
        print("Tentando adquirir o lock...")
        usleep(1000)
        acquired = dlock.try()
    }
    // Faz alguma coisa aqui...
    dlock.unlock()
}
```

Concorrência

Programação concorrente estruturada

- O Grand Central Dispatch (GCD) é uma API baseada em filas que permite executar closures em workers pools.
- Closures contendo um trabalho que precisa ser executado podem ser adicionados a uma fila que os executará usando uma série de threads sequencialmente ou em paralelo, dependendo das opções de configuração da fila. Porém, independentemente do tipo da fila, os trabalhos sempre serão iniciados após o pedido de entrada, o que significa que os jobs serão sempre iniciados respeitando a ordem de inserção. A ordem de conclusão dependerá da duração de cada job.

Concorrência

Dispatch Queues

- O GCD permite a criação de filas mas também garante acesso a algumas filas pré-definidas pelo sistema.
- Para criar uma fila serial básica, a fila que executará suas closures sequencialmente, você precisa fornecer uma string de rótulo que irá identificá-la e geralmente é recomendado usar um prefixo de nome de domínio de ordem inversa para simplificar o rastreamento do proprietário da fila em rastreamentos de pilha.

```
let serialQueue = DispatchQueue(label: “br.ufes.Serial", attributes: .serial)

let concurrentQueue = DispatchQueue(label: "br.ufes.Concurrent", attributes:
.concurrent)
```

Concorrência

- A segunda fila que criamos é simultânea, o que significa que a fila usará todos os encadeamentos disponíveis em seu conjunto de encadeamentos subjacente ao executar as tarefas que ela contém. A ordem de execução é imprevisível nesse caso, não presuma que a ordem de conclusão de seus fechamentos será de alguma forma relacionada ao pedido de inserção.
- As filas padrão podem ser recuperadas do objeto DispatchQueue:

```
let mainQueue = DispatchQueue.main

let globalDefault = DispatchQueue.global()
```

Concorrência

Using Queues:
Tarefas, na forma de closures, podem ser enviadas a uma fila de duas maneiras: de forma síncrona, utilizando o método sync, ou de forma assíncrona, utilizando o método async.

```
globalDefault.async {
    print("Async na MainQ, first?")
}

globalDefault.sync {
    print("Sync na MainQ, second?")
}
```

Concorrência

Várias chamadas de despacho podem ser aninhadas, por exemplo, quando após alguma operação em segundo plano, baixa prioridade, executada em uma fila de nossa escolha, precisamos atualizar o formulário da interface do usuário na fila principal.

```
DispatchQueue.global(qos: .background).async {
    // Algum trabalho de background aqui...
    DispatchQueue.main.async {
        // Hora de atualizar a User Interface
        print("UI atualizada na fila principal")
    }
}
```

Concorrência

- Closures ainda podem ser executadas após um delay específico. Swift 3 permite especificar de uma maneira mais confortável o intervalo de tempo utilizando o tipo Enum DispatchTimeInterval que permite compor intervalos utilizando as quatro seguintes unidades de tempo: .seconds(Int), .milliseconds(Int), .microseconds(Int) e .nanoseconds(Int).
- Para agendar uma closure para uma futura execução é necessário utilizar o método asyncAfter(deadline:execute:) com o intervalo de tempo.

```
globalDefault.asyncAfter(deadline: .now() + .seconds(5)) {
    print("Apos 5 segundos")
}
```

Concorrência

Ainda é possível realizar a execução com múltiplas interações da mesma closure utilizando o método concurrentPerform(iterations:execute:), porém é necessário cuidado, essas closures serão executadas simultaneamente, se possível, no contexto da fila atual, portanto, lembre-se de sempre incluir uma chamada de método sync ou async em uma fila que suporte a simultaneidade.

```
globalDefault.sync {
    DispatchQueue.concurrentPerform(iterations: 5) {
        print("\($0) vezes")
    }
}
```

Concorrência

Barriers

Digamos que você tenha adicionado uma série de closures a uma fila específica (com diferentes durações), mas agora deseja executar uma tarefa somente depois que todas as tarefas assíncronas anteriores forem concluídas.

Vamos adicionar 5 tarefas (que duram por um tempo limite de 1 segundo) à fila simultânea que criamos anteriormente e usar uma barreira para imprimir algo assim que as outras tarefas forem concluídas. Isso será feito especificando uma sinalização DispatchWorkItemFlags.barrier no final da chamada async.

Concorrência

```
let concurrentQueue = DispatchQueue(label: "LP.Swift.Concurrent", attributes: .concurrent)
concurrentQueue.async {
    DispatchQueue.concurrentPerform(iterations: 5) {
        (id:Int) in
        sleep(1)
        print("Async on concurrentQueue, 5 times: "+String(id))
    }
}
concurrentQueue.async (flags: .barrier) {
    print("Todas as 5 tarefas foram completadas.")
}
```

Avaliação da Linguagem

Avaliação da Linguagem

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)

Avaliação da Linguagem

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

Avaliação da Linguagem

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

Avaliação da Linguagem

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

Referências

Referências

https://arstechnica.com/gadgets/2014/10/os-x-10-10/22/

https://insights.stackoverflow.com/survey/2018/#technology-most-loved-dreaded-and-wanted-languages

https://swift.org

https://en.wikipedia.org/wiki/Swift_(programming_language)

https://developer.apple.com/swift-playgrounds/

https://medium.com/@sdrzn/functional-programming-in-swift-221a8cabb8c

https://www.uraimo.com/2017/05/07/all-about-concurrency-in-swift-1-the-present/