Swift (Beta 08/2014)

XCode 6 beta 5

em 45 minutos

helder da rocha

helder@argonavis.com.br

Agenda

- O que é Swift?
- Viagem rápida e abrangente pela sintaxe do Swift através de exemplos
- Como criar apps com Swift
- Por que aprender Swift? Quando?

Я

- HTML, CSS, Java, JavaScript, XML, XSLT, XPath, SVG, Objective-C
  - www.argonavis.com.br
- www.helderdarocha.com.br

Swift

- Nova linguagem da Apple para aplicativos iOS e OS X
  - Chris Lattner + colaboradores, desde 2010
- Linguagem em Beta disponibilizada para desenvolvedores registrados Apple desde Junho 2014
  - O registro é gratuito (não requer iOS ou Mac DP)
- Baseada em C e Objective-C mas sem preservar compatibilidade de código
- Inspirada em várias linguagens e paradigmas modernos
  - Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, ...
- Sintaxe familiar com baixa curva de aprendizado

Linguagem beta?

- Não está pronta!

2-Jun-2014 (não foi lançamento... ainda)

- Como...
  - Java em 1995
  - LiveScript a.k.a JavaScript em 1995
  - HTML e CSS em 1994 e eternamente
  - etc.

Como iniciar

- **XCode 6 playground**: em tempo real!

Digite aqui

valor da variavel

console

MySwiftPlayground.playground — Edited

MySwiftPlayground.playground 〉 No Selection

```
// Playground - noun: a place where people can play

var variable = 123
let constant = 456
variable = 789

var typed:Float = 345
typed = 12.3 + 5;

let frase = "Hoje é dia "
let dia = 16
let texto = frase + String(dia)

let numero = 30
let conta = 2

let texto2 = "eu tenho \(numero) gatos e \(conta) cachorros "

let grupo1 = [texto, "dog"];
var grupo2 = [6, 7];
grupo1[1] = "cat";
var a = grupo1;
grupo2[1] = 6;
var b = grupo2;

var jeison = [
    "abc": "5",
    "casa": "dono"
];
```

```
123
456
789

345.0
17.3

"Hoje é dia "
16
"Hoje é dia 16"

30
2

"eu tenho 30 gatos e 2 cachorros "

["Hoje é dia 16", "dog"]
[6, 7]
"cat"
["Hoje é dia 16", "cat"]
6
[6, 6]

["casa": "dono", "abc": "5"]
```

(no results)

Console Output

```
Sides 10
area: 314.0
```

var a = grupo1;

```
[0] "Hoje é dia 16"
[1] "cat"
```

```
[0] "Hoje é dia 16"
[1] "cat"
```

30 sec

TDC2014 the developer's conference

Hello, world

Nao precisa importar bibliotecas

Nao precisa de main()

```
println("Hello,  world")
```

Nao precisa de ponto-e-virgula no final

Operadores

- Praticamente mesmos operadores de outras linguagens similares a C, Java, C#, JavaScript, etc.

- % calcula resto de ponto-flutuante

- a..<b e a...b para faixa de valores

- === e !== para testar se referencia de objeto e' a mesma

Comentários aninhados

- comentarios de bloco podem ser aninhados

```
/*
   Bloco
  /* sub bloco */
   Continua
*/
```

Constantes e variáveis

```
var variable = 123
let constant = 456
```

```
variable = 789
```

alteracao de variavel

Constantes e variáveis

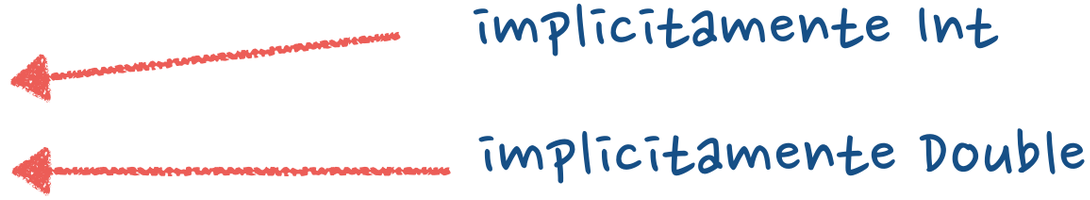
```
var variable = 123
let constant = 456

variable = 789
//constant = 111
```

alteracao de variavel

nao pode alterar constante

Tipos

```
var numero1 = 345
var numero2 = 3.45
```



Tipos

```
var numero1 = 345
var numero2 = 3.45

numero1 = numero2
```

implicitamente Int

implicitamente Double

sera' que posso fazer isto?

Int

Double

Tipos

```
var numero1 = 345
var numero2 = 3.45

//numero1 = numero2
numero2 = numero1
```

implicitamente Int

implicitamente Double

e isto?

Double

Int

Tipos

```
var numero1 = 345
var numero2 = 3.45
```

implicitamente Int

implicitamente Double

```
//numero1 = numero2
//numero2 = numero1
```

atribuicoes incompativeis

variavel so aceita MESMO tipo!

```
numero1 = Int(numero2)
numero2 = Double(numero1)
```

conversao

Tipos

```
var numero1 = 345
var numero2 = 3.45
```

implicitamente Int

implicitamente Double

```
//numero1 = numero2
//numero2 = numero1
```

atribuicoes incompativeis

```
numero1 = Int(numero2)
numero2 = Double(numero1)
```

conversao

```
var numero3:Double = 345
```

anotacao de tipo

Tipos

```
var numero1 = 345
var numero2 = 3.45
```

implicitamente Int

implicitamente Double

```
//numero1 = numero2
//numero2 = numero1
```

atribuicoes incompativeis

```
numero1 = Int(numero2)
numero2 = Double(numero1)
```

conversao

```
var numero3:Double = 345
```

anotacao de tipo

```
var numero4 = numero2 + numero3
```

implicitamente Double

Tipos

```
var numero1 = 345
var numero2 = 3.45
```

implicitamente Int

implicitamente Double

```
//numero1 = numero2
//numero2 = numero1
```

atribuicoes incompativeis

```
numero1 = Int(numero2)
numero2 = Double(numero1)
```

conversao

```
var numero3:Double = 345
```

anotacao de tipo

```
var numero4 = numero2 + numero3

numero3 = numero4 + numero2 + 123
```

conversao automatica de literal

Concatenação

```
let frase = "Hoje é dia "
let dia = 16
//let texto1 = frase + dia
let texto2 = frase + String(dia)
```

conversao ilegal

concatenacao de String

texto2

Hoje é dia 16

Interpolação

```
let frase = "Hoje é dia "
let dia = 16
//let texto1 = frase + dia
let frase2 = "Hoje é dia \(dia)"
```

conversao ilegal

interpolacao de variavel

frase2

Hoje é dia 16

Interpolação

```
let frase = "Hoje é dia "
let dia = 16
//let texto1 = frase + dia
let frase2 = "Hoje é dia \(dia + 10)"
```

conversao ilegal

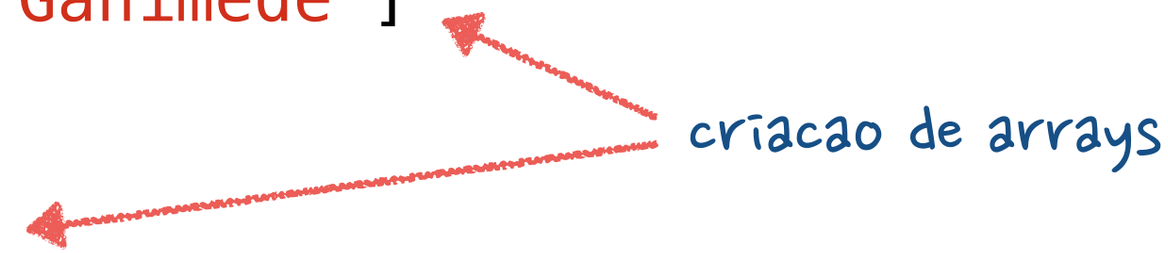
frase2

Hoje é dia 26

interpolacao de expressao

Arrays

```
var luas = ["Miranda", "Ganimede"]

let numeros = [1, 2, 3]
```



Arrays

```swift
var luas = ["Miranda", "Ganimede"]
```

criacao de arrays

```swift
let numeros = [1, 2, 3]
```

alterando dados

```swift
luas[1] = "Oberon"
```

resultado → ["Miranda", **"Oberon"**]

Arrays

```swift
var luas = ["Miranda", "Ganimede"]
```

criacao de arrays

```swift
let numeros = [1, 2, 3]
```

alterando dados

```swift
luas[1] = "Oberon"
```

resultado → ["Miranda", "Oberon"]

alterando dados

```swift
numeros[2] = 5
```

resultado → [1, 2, 5]

Arrays

```
var luas = ["Miranda", "Ganimede"]
```

criacao de arrays

```
let numeros = [1, 2, 3]
```

alterando dados

```
luas[1] = "Oberon"
```

resultado → ["Miranda", "Oberon"]

alterando dados

```
numeros[2] = 5
```

resultado → [1, 2, 5]

atribuindo novo valor

```
// numeros = []
```

(nao pode por ser constante - let)

```
luas = []
```

(pode por ser variavel - var)

Mais arrays

```
var luas: [String] = []
luas.count   ⟶ 0
luas.isEmpty ⟶ true
```

Mais arrays

```
var luas: [String] = []
luas.count   ⟶ 0
luas.isEmpty ⟶ true

luas.append("Puck")
luas.count   ⟶ 1
```

Mais arrays

```
var luas: [String] = []
luas.count   ⟶ 0
luas.isEmpty ⟶ true

luas.append("Puck")
luas.count   ⟶ 1

luas += ["Europa", "Io", "Callysto"]
luas.count   ⟶ 4
```

Mais arrays

```
var luas: [String] = []
luas.count   ⟶ 0
luas.isEmpty ⟶ true

luas.append("Puck")
luas.count   ⟶ 1

luas += ["Europa", "Io", "Callysto"]
luas.count   ⟶ 4
luas[1..<3]  ⟶ ["Europa", "Io"]
```

Tuplas

- valores compostos

```
let estado = ("SP", "Sao Paulo", 1)
let (uf, nome, seq) = estado
```

Tuplas

- valores compostos

```
let estado = ("SP", "Sao Paulo", 1)
let (uf, nome, seq) = estado
```

uf → "SP"

nome → "Sao Paulo"

seq → 1

Tuplas

- valores compostos

```
let estado = ("SP", "Sao Paulo", 1)
let (uf, nome, seq) = estado

estado.0
uf

estado.1
nome

estado.2
seq
```

"SP"

"Sao Paulo"

1

Tuplas

- valores compostos

```
let estado = ("SP", "Sao Paulo", 1)
let (uf, nome, seq) = estado

estado.0
uf          → "SP"

estado.1
nome        → "Sao Paulo"

estado.2
seq         → 1

let a: (String, String, Int) = estado
a.2         → 1
```

Mapas

```
var mapa: [String, String] = [:]

mapa = ["Cidade": "Recife", "CEP": "50123"]

mapa["Telefone"] = "12345678"

mapa.keys   ⟶  ["Cidade", "CEP", "Telefone"]

mapa.values ⟶  ["Recife", "50123", "12345678"]

for(key, value) in mapa { ... }
```

para navegar pelo mapa

Tipagem **forte**

variaveis sempre contem valor do seu tipo

NENHUM outro tipo e´ permitido, nem mesmo nil

```
var item1 = "texto"
```

tipo String determinado de forma implicita

```
var item2: String
```

tipo String determinado de forma explicita

Tipagem **forte**

variaveis sempre contem valor do seu tipo

NENHUM outro tipo e´ permitido, nem mesmo nil

```
var item1 = "texto"
//item1 = nil
var item2: String
//item2 = nil
```

tipo String determinado de forma implicita

tipo String determinado de forma explicita

nao permitido valor que nao seja String

Tipos **opcionais**

variaveis sempre contem valor do seu tipo

NENHUM outro tipo e´ permitido, nem mesmo nil

A menos que o valor seja empacotado como OPCIONAL

```
var item1 = "texto"
//item1 = nil
var item2: String
//item2 = nil
var item3:String?
item3 = "texto"
item3 = nil
```

tipo String determinado de forma implicita

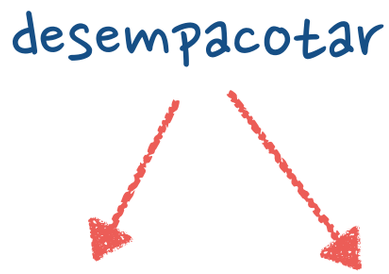
tipo String determinado de forma explicita

indica que valor de item3 e'opcional!

Desempacotando opcionais

- Se valor nao e´ nil, opcional e´ desempacotado com ! para que possa ser usado

```
var texto = "123"
var numero: Int? = texto.toInt()

if numero != nil {
    println("O numero ao quadrado é \(numero!*numero!)")
}
```



Desempacotando opcionais

- Se valor nao e´ nil, opcional e´ desempacotado com ! para que possa ser usado

```
var texto = "123"
var numero: Int? = texto.toInt()

if numero != nil {
    println("O numero ao quadrado é \(numero!*numero!)")
}

if let n = numero {
    println("O numero ao quadrado é \(n*n)")
}
```

desempacotar

desempacota automaticamente

Controle de fluxo

- condicionais com switch e if
- Loops com for-in, for, while e do-while

```
for c in "Texto" { println(c) }
```

- controle dentro de loops com break, continue, e fallthrough em switch
- loops com labels

Exemplo: **for-in** e **if**

```
let vendas = [120, 45, 233, 57, 23, 91]
var maior = 0

for venda in vendas {
    if venda >= maior {
        maior = venda
    }
}
```

expressao Boolean

Sintaxe opcional

declaracao opcional de tipo

```
let vendas: [Int] = [120, 45, 233, 57, 23, 91];
var maior: Int = 0;
```

declaracao opcional de tipo

```
for venda: Int in vendas {
    if (venda >= maior) {
        maior = venda;
    }
}
```

parenteses opcionais

ponto e virgula opcional

Teste é sempre **Boolean**...

ZERO nao equivale a false

expressao PRECISA ser Boolean

```
let vendas = [120, 0, 233, 0, 23, 91]

for venda in vendas {
    //if venda {
    //    println("Venda foi zero")
    //}
}
```

... ou **opcional**

OPCIONAL: conteudo do array
pode ser Int ou nil

```
let vendas: [Int?]
    = [120, nil, 233, nil, 23, 91]
```

neste caso, nil
funciona como false

```
for venda in vendas {
    if venda {
        println("Venda foi nil")
    }
}
```

Atribuicao condicional

```swift
let vendas: [Int?]
    = [120, nil, 233, nil, 23, 91]

for venda in vendas {
    if let valor = venda {
        println("Venda foi \(valor)")
    }
}
```

imprime

```
Venda foi 120
Venda foi 233
Venda foi 23
Venda foi 91
```

Switch

Diferente de c, Java, JavaScript, etc.

Break e' automatico em cada case!

```
let planeta = "Jupiter"
var tipo: String

switch planeta {
  case "Terra", "Marte","Venus","Mercurio" :
    tipo = "Planeta rochoso."
  case "Netuno","Urano","Saturno","Jupiter":
    tipo = "Planeta gasoso."
  case "Ceres","Plutão","Haumea","MakeMake","Eris":
    tipo = "Planeta anão"
  default:
    tipo = "Nao é um planeta."
}
```

executa APENAS este case!

Switch: + recursos

- comportamento fallthrough, break, continue
- value binding
- clausula where (condicoes adicionais)

Mais loops

0..<10 para 0 <= i < 10

0...10 para 0 <= i <= 10

```
var contagem = 0

for i in 0..<10 {
    contagem += i
}

contagem = 0

for (var i = 0; i < 10; ++i) {
    contagem++
}
```

```
var n = 60
while n < 100 {
    n = n * 2
}

n = 100
do {
    n *= 2
} while (n < 100)
```

parenteses

sao opcionais

Funções

nome

parametro: tipo

tipo de retorno

```
func data(dia:Int, mes:Int, ano:Int) -> String {
    return "\(dia)/\(mes)/\(ano)"
}
```

```
var inicio = data(8,7,2014)
```

chamada da funcao

Funções com labels

nome

parametro: tipo

tipo de retorno

```
func data(#dia:Int, #mes:Int, #ano:Int) -> String {
    return "\(dia)/\(mes)/\(ano)"
}


var inicio = data(dia: 8, mes: 7, ano: 2014)
```

chamada da funcao

Funções retornando tuplas

```
func parseData(var data: String) ->(Int, Int, Int) {

    let comp = data.componentsSeparatedByString("/")

    return ( comp[0].toInt()!,
             comp[1].toInt()!,
             comp[2].toInt()! )
}
```

comp terá array de String
["8","8","2014"]

retorna tupla de Int
(8, 8, 2014)

Funções retornando tuplas

```
func parseData(var data: String) ->(Int, Int, Int) {

    let comp = data.componentsSeparatedByString("/")

    return ( comp[0].toInt()!,
             comp[1].toInt()!,
             comp[2].toInt()! )
}

var resultado = parseData("8/8/2014")
```

comp terá array de String
["8","8","2014"]

retorna tupla de Int
(8, 8, 2014)

chamada da funcao

resultado.0 → 8
resultado.1 → 8
resultado.2 → 2014

Parâmetros **inout**

- Parametros usados em funcoes sao copias
  - Nao alteram o valor!

- Esse comportamento requer que sejam declaradas como inout

```
func trocar(inout a: Int, inout b: Int) {
    let temp = a
    a = b
    b = temp
}
```

Argumentos variáveis

```
func media(numbers: Int...) -> Int {
    var total = 0
    let count = numbers.count
    if count == 0 { return 0 }
    for number in numbers {
        total += number
    }
    return total / numbers.count
}
```

é um array!

Argumentos variáveis

```
func media(numbers: Int...) -> Int {
    var total = 0
    let count = numbers.count
    if count == 0 { return 0 }
    for number in numbers {
        total += number
    }
    return total / numbers.count
}
media()
media(100,200)
media(33, 386, 128, 99)
```

Funções aninhadas

retorna funcao que

recebe String e retorna String

```
func asteriscos() -> (String->String) {
    let star = "*"
    return ...
}
```

Funções aninhadas

retorna funcao que

recebe String e retorna String

```
func asteriscos() -> (String->String) {
    let star = "*"
    func alterar(texto:String) -> String {
        return "\(star)\(texto)\(star)"
    }
    return alterar
}
```

Funções aninhadas

retorna funcao que

recebe String e retorna String

```
func asteriscos() -> (String->String) {
    let star = "*"
    func alterar(texto:String) -> String {
        return "\(star)\(texto)\(star)"
    }
    return alterar
}

var quote = asteriscos()
```

(Function)

Funções aninhadas

retorna funcao que

recebe String e retorna String

```
func asteriscos() -> (String->String) {
    let star = "*"
    func alterar(texto:String) -> String {
        return "\(star)\(texto)\(star)"
    }
    return alterar
}

var quote = asteriscos()
quote("Bom dia!")          → *Bom dia!
quote("12345")             → *12345*
quote("Hello, world")      → *Hello, world!*
```

Funções aninhadas

retorna uma funcao que

recebe o parametros String e retorna tupla (Int,Int)

```
func a() -> (()->(Int, Int)) {
    func embutida() ->(Int, Int) {
        return (3,4)
    }
    return embutida
}
```

esta funcao

recebe o parametros e retorna

tupla (Int,Int)

Funções aninhadas

retorna uma funcao que

recebe dois parametros String e retorna tupla (Int,Int)

```
func a() -> ((String,String)->(Int,Int)) {
    func embutida(a:String, b:String) ->(Int,Int) {
        return (3,4)
    }
    return embutida
}
```

esta funcao

recebe dois parametros String e

retorna tupla (Int,Int)

Funções aninhadas

retorna uma funcao que

recebe dois parametros String e retorna tupla (Int,Int)

```
func a() -> ((String,String)->(Int,Int)) {
    func embutida(a:String, b:String) ->(Int,Int) {
        return (3,4)
    }
    return embutida
}
```

esta funcao recebe dois parametros String e retorna tupla (Int,Int)

```
var emb = a()
emb("a", "b").0
emb("a", "b").1
```

Acesso a funcao embutida

3 (primeiro elemento da tupla)

4 (segundo elemento da tupla)

Funções aninhadas

esta funcao recebe String

retorna funcao que recebe String e retorna String

```
func delim(sep: String) -> (String->String) {
    func alterar(texto:String) -> String {
        return "\(sep)\(texto)\(sep)"
    }
    return alterar
}
```

funcao aninhada

Funções aninhadas

esta funcao recebe String

retorna funcao que recebe String e retorna String

```
func delim(sep: String) -> (String->String) {
    func alterar(texto:String) -> String {
        return "\(sep)\(texto)\(sep)"
    }
    return alterar
}

var tag = delim("<tag>")
tag("Bom dia!")
```

funcao aninhada

<tag>Bom dia!<tag>

Funções aninhadas

esta funcao recebe String

retorna funcao que recebe String e retorna String

```
func delim(sep: String) -> (String->String) {
    func alterar(texto:String) -> String {
        return "\(sep)\(texto)\(sep)"
    }
    return alterar
}

var tag = delim("<tag>")
tag("Bom dia!")
var dash = delim("-----")
dash("Bom dia!")
```

funcao aninhada

<tag>Bom dia!<tag>

-----Bom dia!-----

Funções recebendo funções

representa o TIPO da funcao calc()

```
func numero(funcao: String->Int) -> String->Int {
    func text(value: String) -> Int {
        var number = funcao(value)
        return number * 4;
    }
    return text
}
```

Funções recebendo funções

representa o TIPO da funcao calc()

```
func numero(funcao: String->Int) -> String->Int {
    func text(value: String) -> Int {
        var number = funcao(value)
        return number * 4;
    }
    return text
}

func calc(txt: String) -> Int {
    return txt.utf16Count
}

var myFunc: (String->Int) -> String -> Int
```

representa o TIPO da funcao numero()

Funções recebendo funções

representa o TIPO da funcao calc()

```
func numero(funcao: String->Int) -> String->Int {
    func text(value: String) -> Int {
        var number = funcao(value)
        return number * 4;
    }
    return text
}

func calc(txt: String) -> Int {
    return txt.utf16Count
}

var myFunc: (String->Int) -> String -> Int
myFunc = numero
```

representa o TIPO da funcao numero()

Funções recebendo funções

representa o TIPO da funcao calc()

```
func numero(funcao: String->Int) -> String->Int {
    func text(value: String) -> Int {
        var number = funcao(value)
        return number * 4;
    }
    return text
}

func calc(txt: String) -> Int {
    return txt.utf16Count
}

var myFunc: (String->Int) -> String -> Int
myFunc = numero
var getLength = myFunc(calc)
```

representa o TIPO da funcao numero()

Funções recebendo funções

representa o TIPO da funcao calc()

```
func numero(funcao: String->Int) -> String->Int {
    func text(value: String) -> Int {
        var number = funcao(value)
        return number * 4;
    }
    return text
}

func calc(txt: String) -> Int {
    return txt.utf16Count
}

var myFunc: (String->Int) -> String -> Int
myFunc = numero
var getLength = myFunc(calc)

getLength("abcd")
```

16

representa o TIPO da funcao numero()

Funções recebendo funções

representa o TIPO da funcao calc()

```
func numero(funcao: String->Int) -> String->Int {
    func text(value: String) -> Int {
        var number = funcao(value)
        return number * 4;
    }
    return text
}

func calc(txt: String) -> Int {
    return txt.utf16Count
}

var myFunc: (String->Int) -> String -> Int
myFunc = numero
var getLength = myFunc(calc)

getLength("abcde") ——→ 25
```

representa o TIPO da funcao numero()

Representação de tipos

tipo escalar → `var x: String`

colecao (array) → `var x: String[]` *ou* `x: Array<String>`

tupla → `var x: (String, String)`

Representação de tipos

tipo escalar → `var x: String`

colecao (array) → `var x: [String]` *ou* `x: Array<String>`

tupla → `var x: (String, String)`

x e´ funcao que recebe String e retorna Int

funcao → `var x: String -> Int`

Representação de tipos

tipo escalar → `var x: String`

colecao (array) → `var x: [String]` *ou* `x: Array<String>`

tupla → `var x: (String, String)`

x e´ funcao que recebe String e retorna Int →

funcao → `var x: String -> Int`

funcao → `var x: (String->Int)->(()->(Float,Float))`

Representação de tipos

tipo escalar → `var x: String`

colecao (array) → `var x: [String]` *ou* `x: Array<String>`

tupla → `var x: (String, String)`

x e' funcao que recebe String e retorna Int

funcao → `var x: String -> Int`

retorna: funcao sem argumentos que retorna uma tupla de Floats

funcao → `var x: (String->Int)->(()->(Float,Float))`

recebe: funcao com um argumento tipo String que retorna Int

Representação de tipos

tipo escalar → `var x: String`

colecao (array) → `var x: [String]` *ou* `x: Array<String>`

tupla → `var x: (String, String)`

x e' funcao que recebe String e retorna Int

funcao → `var x: String -> Int`

retorna: funcao sem argumentos que retorna uma tupla de Floats

funcao → `var x: (String->Int)->(()->(Float,Float))`

parenteses opcionais

recebe: funcao com um argumento tipo String que retorna Int

Representação de tipos

tipo escalar → `var x: String`

colecao (array) → `var x: [String]` *ou* `x: Array<String>`

tupla → `var x: (String, String)`

x e' funcao que recebe String e retorna Int

funcao → `var x: String -> Int`

funcao → `var x: (String->Int)->(()->(Float,Float))`

funcao → `var x: () -> ()`

retorna: funcao sem argumentos que retorna uma tupla de Floats

parenteses opcionais

recebe: funcao com um argumento tipo String que retorna Int

Representação de tipos

tipo escalar → `var x: String`

colecao (array) → `var x: [String]` *ou* `x: Array<String>`

tupla → `var x: (String, String)`

x e' funcao que recebe String e retorna Int

funcao → `var x: String -> Int`

funcao → `var x: (String->Int)->(()->(Float,Float))`

funcao → `var x: () -> ()`

x e' funcao que recebe: funcao sem argumentos que retorna funcao sem argumentos

recebe: funcao com um argumento tipo String que retorna Int

parenteses opcionais

retorna: funcao sem argumentos que retorna uma tupla de Floats

Funções e closures

```
func calc(txt: String) -> Int {
    return txt.utf16Count
}

var getLength = numero(calc)

getLength("abcd")
```

Funções e closures

pode-se colocar esta funcao...

```
func calc(txt: String) -> Int {
    return txt.utf16Count
}

var getLength = numero(calc)

getLength("abcd")
```

no local onde e' chamada, tornando-a "anonima"

Funções e closures


```
func calc(txt: String) -> Int {
    return txt.utf16Count
}

var getLength = numero(
    calc
)

getLength("abcd")
```

Funções e closures

este conteudo...

```
func calc(txt: String) -> Int {
    return txt.utf16Count
}

var getLength = numero(
    calc
)

getLength("abcd")
```

aqui

Funções e closures

```
            (txt: String) -> Int {
        return txt.utf16Count
}

var getLength = numero(
            (txt: String) -> Int {
        return txt.utf16Count
}
)

getLength("abcd")
```

Funções e closures

```
        (txt: String) -> Int {
    return txt.utf16Count
}
```

```
var getLength = numero(
    (txt: String) -> Int {
        return txt.utf16Count
    }
)

getLength("abcd")
```

Funções e closures

depois mova esta chave para o inicio

```
var getLength = numero(
    (txt: String) -> Int {
        return txt.utf16Count
    }
)

getLength("abcd")
```

Funções e closures

e substitua { por

in

```
var getLength = numero(
 { (txt: String) -> Int in
        return txt.utf16Count
    }
)

getLength("abcd")
```

Funções e closures

```
var getLength = numero(
 { (txt: String) -> Int in
        return txt.utf16Count
    }
)

getLength("abcd")
```

Funções e closures

```
var getLength = numero(
 { (txt: String) -> Int in
        return txt.utf16Count
    }
)

getLength("abcd")
```

nao precisa dos parenteses, nem tipo, nem return

Funções e closures

```
var getLength = numero(
 {  txt                -> Int in
                  txt.utf16Count
    }
)

getLength("abcd")
```

Funções e closures

```
var getLength = numero(
    { txt -> Int in
        txt.utf16Count
    }
)

getLength("abcd")
```

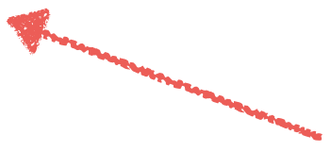
agora cabe tudo em uma linha

Closures

```
var getLength =
  numero({ txt -> Int in txt.utf16Count })

getLength("abcd")
```

Closures

```
var getLength =
  numero({ txt -> Int in txt.utf16Count })

getLength("abcd")
```



substitua o nome

pelo bloco

Closures

```
numero({txt -> Int in txt.utf16Count})("abcd")
```

Classes

Nao requerem implementacao e interface separadas como Objective-C

Membros:

metodos (func), variaveis e constantes (var, let), indices (subscript)

Classes

Nao requerem implementacao e interface separadas como Objective-C

Membros:

metodos (func), variaveis e constantes (var, let), indices (subscript)

```
class Figura {
    var altura = 2
    var largura = 2
    func area() -> Double {
        return altura * largura
    }
}


var fig = Figura()
fig.area()
```

Classes

Nao requerem implementacao e interface separadas como Objective-C

Membros:

metodos (func), variaveis e constantes (var, let), indices (subscript)

declaracao de classe

propriedades

declaracao de metodo

```
class Figura {
    var altura = 2
    var largura = 2
    func area() -> Double {
        return altura * largura
    }
}

var fig = Figura()
fig.area()
```

referencia para objeto na memoria

inicializador

chamada de metodo

Inicializadores

```
class Figura {
    var altura: Double
    var largura: Double
    init(altura: Double, largura: Double) {
        self.altura  = altura
        self.largura = largura
    }
    func area() -> Double {
        return altura * largura
    }
}

var fig = Figura(altura:10,largura:5)
fig.area()
```

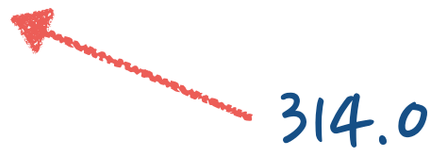
inicializador

mesmos parametros de init()

50

Herança e override

```
class Circulo : Figura {
    init(raio: Double) {
        super.init(altura:raio,largura:raio)
    }
    override func area() -> Double {
        return 3.14 * largura * largura
    }
}

var circ = Circulo(raio: 10)
circ.area()
```



Propriedades calculadas

```
import Cocoa
class Circulo2 : Figura {
    init(raio: Double) {
        super.init(altura:raio,largura:raio)
    }
    var area: Double {
        get {
            return 3.14 * largura * largura
        }
        set {
            largura = sqrt( newValue / 3.14 )
            altura = largura
        }
    }
}

var circ2 = Circulo2(raio: 10)
circ2.area
circ2.area = 628
```

altura == largura == 14.142...

Enums

```
enum Simbolo: Int {
    case As = 1, Dois, Tres, Quatro, Cinco, Seis, Sete, Oito, Nove,
         Dez, Valete, Dama, Rei

    func valor() -> String {
        switch self {
        case .As:
            return "Ás"
        case .Valete:
            return "Valete"
        case .Dama:
            return "Dama"
        case .Rei:
            return "Rei"
        default:
            return String(self.toRaw())
        }
    }
}

let dama = Simbolo.Dama.valor()
let nove = Simbolo.Nove.valor()
```

valor numerico (Int)

"Dama"

"9"

Enums

```
enum Naipe {
    case Espadas, Copas, Ouros, Paus
    func nome() -> String {
        switch self {
        case .Espadas:
            return "Espadas"
        case .Ouros:
            return "Ouros"
        case .Paus:
            return "Paus"
        case .Copas:
            return "Copas"
        }
    }
    func cor() -> String {
        switch self {
        case .Espadas, .Paus:
            return "preto"
        case .Copas, .Ouros:
            return "vermelho"
        }
    }
}

let cor = Naipe.Copas.cor()
let naipe = Naipe.Copas.nome()
```

Estruturas (struct)

- Parecidas com classes e enums, mas sempre passadas por valor e nao por referencia
- Nao suportam heranca

```
struct CartaDeBaralho {
    var simbolo: Simbolo
    var naipe: Naipe
    func nome() -> String {
        return "\(simbolo.valor()) de \(naipe.nome())"
    }
}

let carta1 = CartaDeBaralho(simbolo: .Dama, naipe: .Espadas).nome()
let carta2 = CartaDeBaralho(simbolo: .Sete, naipe: .Copas).nome()
```

"Dama de Espadas"

"7 de copas"

Protocolos

- Pura interface (sem implementacao)
- classes, estruturas e enums podem adotar um ou mais protocolos
- Polimorfismo sem heranca - herda-se a interface mas a implementacao precisa ser fornecida
- como Protocolos do Objective-C.
- Similar a interfaces em Java/C#

Protocolos

interfaces

```
protocol Dimensionavel {
    var pontos: [(Int,Int)] { get set }
    mutating func adicionarPonto(ponto: (Int,Int))
}

class Poligono {
    var pontos: [(Int,Int)]
    init(pontos: [(Int,Int)]) {
        self.pontos = pontos
    }
}

class PoligonoRedimensionavel: Poligono, Dimensionavel {
    func adicionarPonto(ponto: (Int,Int)) {
        self.pontos.append(ponto)
    }
}
```

implementacao

implementacao

Protocolos

interfaces

```
protocol Dimensionavel {
    var pontos: [(Int,Int)] { get set }
    mutating func adicionarPonto(ponto: (Int,Int))
}

class Poligono {
    var pontos: [(Int,Int)]
    init(pontos: [(Int,Int)]) {
        self.pontos = pontos
    }
}

class PoligonoRedimensionavel: Poligono, Dimensionavel {
    func adicionarPonto(ponto: (Int,Int)) {
        self.pontos.append(ponto)
    }
}

let triangulo = PoligonoRedimensionavel(pontos:[(0,0), (0,3), (4,0)])
triangulo.pontos.count
triangulo.adicionarPonto((5,3))
triangulo.pontos.count
```

implementacao

3

implementacao

4

Extensões

- categorias em Objective-C sao extensoes em Swift

- Permite adicionar funcionalidade (implementacao) a tipo existente (normalmente adotando um protocolo)

- Pode estender qualquer tipo (inclusive tipos nativos como Int, String)

Extensões

```
protocol Documentavel {
    var log: String { get }
}
```

```
extension Poligono: Documentavel {
    var log: String {
        return "\(self.pontos)"
    }
}
```

```
triangulo.log
```

[(0, 0), (0, 3), (4, 0), (5, 3)]

Genéricos

- Permite grande controle e flexibilidade sobre tipos usados em classes, enums, estruturas e membros

```
func repetir<T>(item: T, vezes: Int) -> [T] {
    var array = [T]()
    for i in 0..<vezes {
        array.append(item)
    }
    return array
}
repetir("au", 5)
repetir(9, 4)
repetir(carta1, 2)
```

["au","au","au","au","au"]

[9,9,9,9]

["Dama de Espadas", "Dama de Espadas"]

Memória?

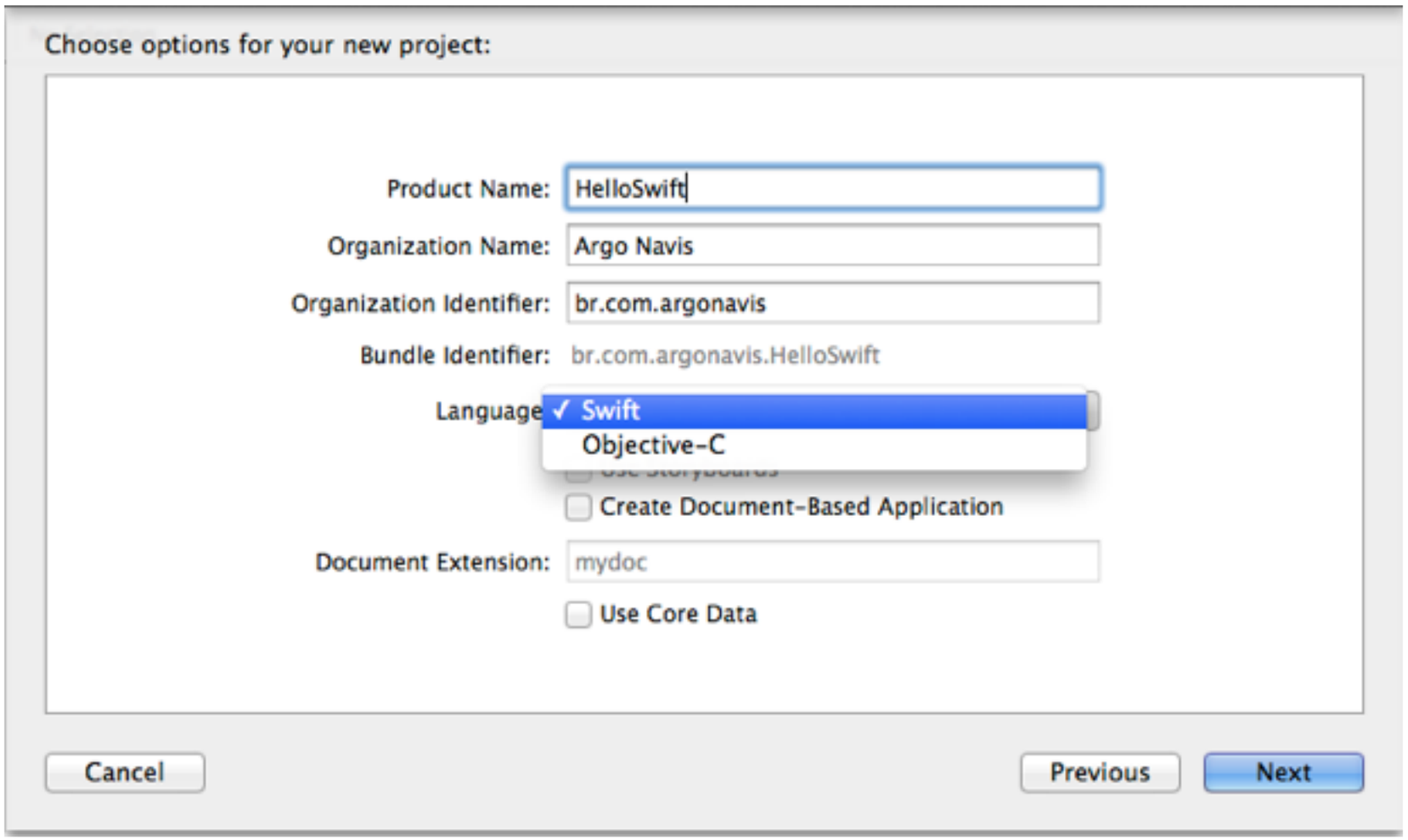
- Swift automaticamente usa ARc - Automatic Reference counting para gerenciar memoria

- Nao e´ garbage collection

- Programador geralmente nao precisa se preocupar com gerencia de memoria
  - Problema: ciclos de referencias fortes
  - Solucoes incluem referencias fracas

Criando uma app

- Quando criar a aplicacao escolha Swift em vez de Objective-C
  - Pode-se misturar classes Objective-C com Swift
  - Pode-se importar qualquer modulo



Usando os frameworks

- Importe os frameworks

```
import Cocoa
```

- Acesse os recursos usando o "estilo Swift"
  - AnyObject no lugar de id
  - String no lugar de NSString
- Exemplo (fonte: "Using Swift with Cocoa and Objective-C")

```
UITableView *myTableView =
    [[UITableView alloc] initWithFrame:CGRectZero
                                 style:UITableViewStyleGrouped];

let myTableView: UITableView =
    UITableView(frame: CGRectZero, style: .Grouped)
```

Dificuldades

- Swift foi lançada em **Beta** em Junho 2014
  - Linguagem ainda em desenvolvimento
  - Acontecem **mudanças** a cada beta do XCode (inclusive na sintaxe - programas deixam de funcionar!)
- O suporte aos frameworks está melhor a cada release, mas ainda há muitos bugs
- Tempo ideal para aprender Swift e sair na frente (mas não investir em grandes aplicações)

Vale a pena aprender Swift?

- É uma **linguagem moderna** com sintaxe **familiar**
  - Por ser familiar, a curva de aprendizado da linguagem é pequena para quem conhece outras linguagens

- Você vai precisar para desenvolver aplicacoes iOS e OS X no futuro
  - Swift vai **substituir** Objective-C no desenvolvimento de aplicacoes iOS e OS X
  - Se você já conhece Objective-C, em poucas semanas estará fazendo apps

Como começar

- Baixe o último **XCode 6** Beta
- Baixe os **livros** via iBooks Store e siga o tutorial (1 a 2 semanas)
  - Swift
  - Using Swift with Cocoa and Objective-C
- Não esqueça de **atualizar** os livros a cada atualização do XCode
  - A sintaxe pode mudar!
- Dúvidas
  - Melhor fórum: stackoverflow.com

Swift

em 45 minutos

helder da rocha

helder@argonavis.com.br