Aqui estão listadas todas as mudanças que precisaram ser feitas aos arquivos Playground do curso Swift em 4 Semanas para atulizá-los com relação a Swift 2.0.

No GitHub vocês vão encontrar pastas e arquivos com (7.0) escritos ao final, que são as versões para rodaram no Xcode 7.0.

Além disso, novos exemplos e materiais sobre **Guard**, **Tratamento de Erros** e **Protocol Extensions.**

Também conseguimos agrupar todos os arquivos playgrounds em um só, usando o novo recurso 'Pages' =)

Xcode 6.4	Xcode 7.0

Geral:

println()	print()

Fluxo de Controle:

- Mudanças nas Strings: https://developer.apple.com/swift/blog/?id=30

for letra in "Cafe"{
println("Letra é \(letra)")
}
	for letra in "Cafe".characters {
	println("Letra é \(letra)")
	}

- Para não haver confusão no código entra a nova estrutura 'do { }' para criar escopos e a estrutura de repetição 'do{ }while', esta última foi substituida por 'repeat{ }while'.

do { print("Pode pedir a última! Saldo: \(dinheiro)") --dinheiro } while dinheiro > 3	repeat { print("Pode pedir a última! Saldo: \(dinheiro)") --dinheiro } while dinheiro > 3

- Foi incluido novo exemplo sobre a cláusula ‘where’ Loops ‘for-in’

```
(NEW!)
for par in 0...10  {

   if par % 2 == 0 {

      par

   }
}
```

```
(NEW!)
for par in 0...10 where par % 2 == 0 {

   par

}
```

Continua & Break:

```
for character in frase  {

   switch character {

   case "o":

      ++contadorLetraO

   default:

      continue   //atua sobre o loop
   }
}
```

```
for character in frase.characters {

   switch character {

   case "o":

      ++contadorLetraO

   default:

      continue   //atua sobre o loop
   }
}
```

Optionals:

- Foi incluido novo exemplo sobre múltiplos OB’s na mesma linha, e cláusulas ‘where’ em estruturas condicionais ‘if’

```
(NEW!)
var celular1: String? = "9867-5671"
var celular2: String? = "9978-6756"

if let cel1 = celular1, cel2 = celular2 where cel1.characters.first == "9" {

   "Ambos possuem número inicial 9"

}
```

Funções:

- Agora apenas o nome do primeiro parametro de uma funcao não possui nome externo, os outros em seguida são locais e externos obrigatórios, a nao ser que use _ antes do nome para poder omitir seu nome externo.

- Agora não existem mais nomes externos CURTOS de parâmetros (#), precisamos agora dobrar o nome do parametro local para tornar ele externo com o mesmo nome.

func juntar(s1: String, s2: String, comJunção junção: String) -> String {
 	
return s1 + junção + s2
 	
}
 	
juntar("flamengo", "vasco", comJunção:" x ")
	func juntar(s1: String, _ s2: String, comJunção junção: String) -> String {
	
	return s1 + junção + s2
	
	}
	
	juntar("flamengo", "vasco", comJunção:" x ")

func possuiLetra(#string: String, #letraEncontrar: Character) -> Bool {
 	
for letra in string {
 	
if letra == letraEncontrar {
 	
return true
 	
}
 	
}
 	
return false
 	
}
	func possuiLetra(string string: String, letraEncontrada letraEncontrar: Character) -> Bool {
	
	for letra in string.characters {
	
	if letra == letraEncontrar {
	
	return true
	
	}
	
	}
	
	return false
	
	}

func novaJuntar(s1: String, s2: String, junção: String = " x ") -> String {
 	
return s1 + junção + s2
 	
}
 	
novaJuntar("Flamengo", "Vasco", junção: " e ")
	func novaJuntar(s1: String, _ s2: String, junção: String = " x ") -> String {
	
	return s1 + junção + s2
	
	}
	
	novaJuntar("Flamengo", "Vasco", junção: " e ")

func chamarIntensidade(var nome: String, intensidade: Int) -> String {
 	
for _ in 1...intensidade {
 	
nome += "!"
 	
}
	func chamarIntensidade(var nome: String, _ intensidade: Int) -> String {
	
	for _ in 1...intensidade {
	
	nome += "!"
	
	}

return nome } chamarIntensidade("Maria", 7)	return nome } chamarIntensidade("Maria", 7)

func trocaAssento(inout a1: String, inout a2: String) {
 	
let assentoTemporario = a1
 	
a1 = a2
 	
a2 = assentoTemporario
 	
}
	func trocaAssento(inout a1: String, inout _ a2: String) {
	
	let assentoTemporario = a1
	
	a1 = a2
	
	a2 = assentoTemporario
	
	}

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

func resultado(funcaoMatematica: (Int, Int) -> Int, a: Int, b: Int) -> String
{
 	
return "Resultado: \(funcaoMatematica(a,b))"
 	
}
	func resultado(funcaoMatematica: (Int, Int) -> Int, _ a: Int, _ b: Int) -> String
	{
	
	return "Resultado: \(funcaoMatematica(a,b))"
	
	}

Closures:

- Por causa de Protocol Extenions, as funções globais sorted(), filter(), map(), …, passaram a ser implementadas por extensões de protocolos como ColletionType que são adotados pelo tipo Array e Dicionário por exemplo, podendo agora ser usadas através da dot syntax deixando o código ainda mais claro.

  Antes: filter(sorted(array){$0 > $1}) {$0 * 2}
  Agora: array.sort{$0 > $1}.filter{$0 * 2}

var megaOrdenadaDecrescente = sorted(megaSena, decrescente) megaOrdenadaDecrescente = sorted(megaSena, {(n1: Int, n2: Int) -> Bool in return n1 > n2}) megaOrdenadaDecrescente = sorted(megaSena, {(n1, n2) in return n1 > n2}) megaOrdenadaDecrescente = sorted(megaSena, {$0 >	var megaOrdenadaDecrescente = megaSena.sort(decrescente) megaOrdenadaDecrescente = megaSena.sort({(n1: Int, n2: Int) -> Bool in return n1 > n2}) megaOrdenadaDecrescente = megaSena.sort({(n1, n2) in return n1 > n2}) megaOrdenadaDecrescente = megaSena.sort({$0 > $1})

$1}) megaOrdenadaDecrescente = sorted(megaSena) {$0 > $1}	megaOrdenadaDecrescente = megaSena.sort {$0 > $1}

(NEW!)

```
var array: [Int?] = [2,1,5,4,nil,7,4,nil,8]

var novaArray = array.flatMap{$0}.sort{$0 > $1}.filter{$0 % 2 == 0}

novaArray
```

Guard:

- Novo arquivo com conteúdo explicativo

Tratamento de Erros:

- Novo arquivo com conteúdo explicativo

Propriedades:

- Melhor explicação sobre propriedades do tipo

(NEW!)

```
struct Circulo  {

    static let PI = 3.14

    var radius: Double

    //Resto da Implementação...

}

Circulo.PI
```

Métodos:

- Melhor explicação sobre métodos do tipo usando o UIColor como exemplo

```
UIColor.greenColor()
```

Inicializadores que falham:

```
(NEW!)

var número = Double("texto")   //nil
número = Double(10)   //10
```

Generics:

- Mesma alteração das funções referente aos nomes de parâmetros.

func permutaDoisInts(inout a: Int, inout b: Int) {
 	
let aTemporário = a
 	
a = b
 	
b = aTemporário
 	
}
	func permutaDoisInts(inout a: Int, inout _ b: Int) {
	
	let aTemporário = a
	
	a = b
	
	b = aTemporário
	
	}

func permutaDoisValores<T>(inout a: T, inout b: T) {
 	
let aTemporário = a
 	
a = b
 	
b = aTemporário
 	
}
	func permutaDoisValores<T>(inout a: T, inout _ b: T) {
	
	let aTemporário = a
	
	a = b
	
	b = aTemporário
	
	}

func todosOsItensCombinam<C1: Container, C2: Container where C1.TipoItem == C2.TipoItem, C1.TipoItem: Equatable>(umContainer: C1, outroContainer: C2) -> Bool { ...	func todosOsItensCombinam<C1: Container, C2: Container where C1.TipoItem == C2.TipoItem, C1.TipoItem: Equatable>(umContainer: C1, _ outroContainer: C2) -> Bool { ...

Protocol Extensions:

Novo arquivo com conteúdo explicativo

Lista App Semana 3 e Desafio 1:

- Agora, a propriedade text de um textField é do tipo String?, então precisamos forçar o desempacotamente ou usar um Optional Binding seguido da clausula where.

if !textField.text.isEmpty {
novoItemDaLista = ItemLista(nome: textField.text)
}
	if !textField.text!.isEmpty {
	novoItemDaLista = ItemLista(nome: textField.text!)
	}

OU...	if let texto = textField.text where !texto.isEmpty {
	novoItemDaLista = ItemLista(nome: texto)
	}

- O `UITableViewController` já adota os protocolos `UITableViewdataSource` e `UITableViewDelegate`. Ao fazer com que o `ListaTableViewController` seja subclasse de `UITableViewController`, essa subclasse automaticamente adota os mesmos protocolos do `UITableViewController`. Por isso não há necessidade de (re)adotá-los no `ListaTableViewController`.

class AdicionarItemViewController: UIViewController, UITableViewDelegate, UITableViewDataSource { ...	class AdicionarItemViewController: UIViewController { ...

Lista App Desafio 2:

- o método dequeueReusableCellWithIdentifier("ListaPrototypeCell", forIndexPath: indexPath) agora retorna um valor do tipo UITableViewCell, não havendo mais a necessidade de realizar o downcast

let cell = tableView.dequeueReusableCellWithIdentifier("ListaPrototypeCell", forIndexPath: indexPath) as! UITableViewCell	let cell = tableView.dequeueReusableCellWithIdentifier("ListaPrototypeCell", forIndexPath: indexPath)

- NSCalendarUnit agora é uma struct, e não mais uma Enum.

let unidades: NSCalendarUnit = .CalendarUnitDay | .CalendarUnitHour | .CalendarUnitMinute let components = calendar.components(unidades, fromDate: startDate, toDate: endDate, options: nil)	let unidades: NSCalendarUnit = [.Day, .Hour, .Minute] let components = calendar.components(unidades, fromDate: startDate, toDate: endDate, options: [ ])

Lista Core Data:

- Abra o projeto e mande converter a sintaxe, dessa forma, pelo menos o arquivo do core data estará atualizado e mais algumas outras partes dos outros.

- O método executeFetchRequest(fetchRequest) agora não retorna mais um tipo Opcional e pode lançar um erro. Por isso devemos capturá-lo e tratá-lo caso aconteça. Também não precisaremos mais usar o Optional Binding, já que o método não retorna mais valores opcional.

var erro: NSError?
 	
let fetchedResults = moc.executeFetchRequest(fetchRequest) as! [Item]?
 	
if let results = fetchedResults {
 	
itens = results
 	
} else {
 	
print("O fetch não foi possível. \(erro), \(erro?.userInfo)")
 	
}
	var erro: NSError
	
	do {
	
	itens = try moc.executeFetchRequest(fetchRequest) as! [Item]
	
	} catch {
	
	erro = error as NSError
	
	print("O fetch não foi possível. \(erro), \(erro.description)")
	
	}

if !novoItemTextField.text.isEmpty {
 	
novoNomeParaItem = novoItemTextField.text
}
	if let texto = novoItemTextField.text where !texto.isEmpty {
	
	novoNomeParaItem = texto
	}

App Ecoenergia:

- No Swift 2.0, não existe mais o método toInt() -> Int? do tipo String. Agora temos um failable initializer do tipo Int que tenta converter texto para inteiro. Usando a forma mais

correta, vamos usar o Optional Binding com 2 sentenças: primeiro vamos desempacotar o valor da propriedade .text caso existe, se existir, vamos fornecer esse valor como parâmetro para o inicializador do tipo Int.

- Para simplificar, existe uma outra opção para todos os exemplos abaixo, é fazer como no primeiro e forçar o desempacotamento (Forced Unwrapping) do valor armazendo em .text!, pois consta na documentação da Apple que mesmo que text seja nil, automaticamente será atribuido “” a ela, não tendo o risco de gerar um erro na hora de desempacotar.

`if textField.text.isEmpty { return true }`	`if let texto = textField.text where texto.isEmpty { return true }`
OU...	`if textField.text!.isEmpty { return true }`

`if let value = potenciaTextField.text.toInt() {`
`  if value > 0 {`
`  potenciaEmWatts = value`
`  return true`
`  }`
`}`
	`if let texto = potenciaTextField.text, value = Int(texto) {`
	`  if value > 0 {`
	`  potenciaEmWatts = value`
	`  return true`
	`  }`
	`}`

`if let value = minutosUsoDiarioTextField.text.toInt() {`
`  if (value <= 60*24) && value > 0 {`
`  minutos = value`
`  return true`
`  }`
`}`
	`if let texto = minutosUsoDiarioTextField.text, value = Int(texto) {`
	`  if (value <= 60*24) && value > 0 {`
	`  minutos = value`
	`  return true`
	`  }`
	`}`

`if !potenciaTextField.text.isEmpty && !minutosUsoDiarioTextField.text.isEmpty {` `....`	`if let text1 = potenciaTextField.text, text2 = minutosUsoDiarioTextField.text where !text1.isEmpty && !text2.isEmpty {` `...`