Centro Integrado de Tecnologia da Informação - CITi
Centro de Informática, Universidade Federal de Pernambuco - CIn/UFPE
Av. Jorn. Aníbal Fernandes, s/n - Cidade Universitária, Recife - PE, 50740-560

CURSO DE DESENVOLVIMENTO iOS

Jornada de Cursos (CITi) 2018

AUTOR: Hilton Pintor (hiltonpintor@gmail.com)

Introdução

Essa apostila tem como objetivo servir de referência durante e após o curso de Desenvolvimento iOS ministrado na Jornada de Cursos do CITi 2018. O conteúdo presente propõe introduzir o leitor aos conceitos básicos de desenvolvimento em Swift para a plataforma iOS.

Os conceitos apresentados serão sempre contextualizados em código escrito em Swift 4, usando o Xcode 9. Ambas versões são as mais recentes.

A leitura desse material não substitui as aulas presenciais. Quando feita em conjunto com as aulas teóricas e práticas, o leitor deverá ser capaz de desenvolver um aplicativo iOS básico do zero.

Todos os exemplos de código presentes nesta apostila se encontram num projeto do xCode, com as classes mostradas e três playgrounds. Encorajo que ao ler a apostila, faça experimentos com o código, mude valores, brinque com ele, para poder melhor entender como realmente funciona.

Swift

Uma das principais linguagens de programação utilizadas para desenvolver aplicações para o ecossistema Apple (iOS, tvOS, watchOS, macOS). Swift foi desenvolvida pela própria empresa como uma solução mais moderna e intuitiva a Objective-C.

Swift é grátis e de código aberto (*open source*), fomentando uma comunidade que é responsável por contribuições diretas ao código fonte, fazendo com que a linguagem esteja sempre melhorando.

Xcode

A principal IDE (*Integrated Development Environment*) utilizada no desenvolvimento de aplicativos do ecossistema Apple, o Xcode está diretamente integrado com os *frameworks* essenciais, e simuladores facilitando o desenvolvimento de aplicações.

Fundamentos

Variáveis e Constantes

Para declarar uma variável em Swift utilizamos a palavra-chave *var* seguida do nome que queremos dar à variável. Podemos também indicar explicitamente qual o tipo da variável, ou deixar que o compilador infira o tipo de acordo com o valor que está sendo atribuído.

```
// declaração de variáveis
var idade: Int = 21
var nome = "Hilton"
```

Outra forma de guardar valores é em constantes. Diferentemente das variáveis, o valor guardado em uma constante não pode mudar, ou seja, só podemos atribuir um valor a uma constante exatamente uma vez. Declarações de constantes são análogas às de variáveis, diferenciadas pelo uso da palavra-chave *let.*

```
// declaração de constantes
let diaNascimento = 02
let nacionalidade: String = "Brasileiro"
```

Uma vez que o tipo de uma variável ou constante foi definido, seja implicitamente ou explicitamente, ela só poderá guardar valores que sejam desse mesmo tipo. Atribuição de novos valores a uma variável é feita da seguinte forma:

```
// atribuição de novo valor
```

```
idade = 22
```

Tipos

Tipos Numéricos

Em Swift, tipos denotam características dos valores. Por exemplo, se quisermos dizer que o valor é um número inteiro, dizemos que ele é do tipo *Int*; se ele for um número fracionário, dizemos ser do tipo *Double* (ou *Float*).

```
// exemplos de tipos numéricos
let inteiro: Int = 10
let decimal: Double = 3.14159265359
let pi: Float = 3.14159265359 // arredondado para 3.141593
```

Booleans

Quando precisamos dizer se um valor é falso ou verdadeiro, utilizamos o tipo *Bool*. Veremos mais aplicações para esse tipo quando explorarmos o Controle de Fluxo da aplicação.

```
// exemplo de booleans
let souRecifense: Bool = true
var odeioSwift = false
```

Strings

Strings são usadas para representar palavras ou caracteres. Para que um valor seja do tipo String, basta colocá-lo entre aspas duplas.

```
// exemplo de Strings
let sobrenome: String = "Pintor"
```

```
let cpf = "111-404-222.16"
```

Tuplas

Tuplas são agrupamentos de um ou mais valores, que podem ser do mesmo tipo ou de tipos diferentes. Tuplas são muito úteis quando precisamos expressar alguma ligação entre os valores, mas não são recomendadas para construções mais complexas.

```
// exemplo de Tuplas
let estado: (String, String) = ("Recife", "PE")
estado.0 // primeiro valor da tupla estado: "Recife"

let rg = (numero: 9153865, orgao: "SDS", UF: "PE")
rg.numero // valor de "numero" da tupla rg: 9153865
```

Opcionais

Ao definir um tipo para uma variável, estamos dizendo que sempre que ela for acessada, teremos um valor do tipo definido. No desenvolvimento de uma aplicação nem sempre podemos afirmar com certeza que haverá um valor dentro de uma variável, por isso em Swift temos o conceito de *Optional*.

Quando dizemos que uma variável é de um tipo opcional, estamos dizendo que dentro dela pode haver um valor ou não (*nil*). Por exemplo, quando pedimos alguma entrada do usuário, o mesmo pode não dar nenhuma, o que nos deixaria com *nil*. A sintaxe dos *Optionals* é formada pela adição de uma interrogação ao nome do tipo original.

```
// exemplo de Optionals
var sentidoVida: Int?
sentidoVida //nesse ponto o valor é nil

sentidoVida = 42
```

```
sentidoVida //nesse ponto o valor é 42
```

Forced Unwrapping

Quando temos certeza que uma variável de tipo opcional tem realmente um valor armazenado nela, e queremos acessar esse valor, podemos fazer um procedimento chamado de *forced unwrapping* adicionando uma exclamação ao nome da variável.

Esse procedimento extrai o valor de dentro da variável opcional, porém deve ser feito com cautela, pois se não houver um valor armazenado (*nil*), resultará em um erro em tempo de execução, matando o aplicativo bruscamente.

```
// forced unwrapping de Optionals
var titulo: String?
titulo = "PhD"
titulo! // extrai o valor da String: "PhD"
```

Optional Binding

É uma forma mais segura de extrair o valor de uma variável opcional, onde primeiro é feito uma conferência para saber se existe um valor, e caso exista, o valor é atribuído a uma constante. Após a atribuição bem sucedida, é possível acessar o valor evitando erros.

```
// optional binding de Opcionais
var opcional: Bool?
opcional = true

if let concreto = opcional {
    concreto // valor extraído: true
}
```

Operadores Básicos

Operadores são símbolos ou frases reservadas que servem para conferir, mudar e combinar valores. Eles são classificados de acordo com a quantidade de operandos, podendo ser unários, binários e ternários (1, 2, ou 3 operandos). Operadores unários também podem ser prefixados ou pós-fixados, ou seja, aparecem antes ou depois do operando.

Até agora já vimos alguns operadores, como o de atribuição, onde o valor da direita é atribuído ao nome da esquerda (variável ou constante):

```
// operador de atribuição
nome = "Hilton"
idade = 22
```

Operadores Aritméticos

Assim como a maioria das linguagens de programação tradicionais, Swift fornece os operadores aritméticos conhecidos por todos, como: adição (+), subtração (-), multiplicação (*) e divisão (/), além de outros que abordaremos em mais detalhes posteriormente.

Os operadores aritméticos só podem ser aplicados a valores de um mesmo tipo, ou seja `Int + Int`, ou `Double + Double`, … Tentar realizar a operação com tipos diferentes (ex: `Int + Double`), resultará em um erro de compilação.

```
// operadores aritméticos
let dez: Int = 10
let dois: Int = 2

dez + dois   // 12
dez - dois   // 8
dez / dois   // 5
dez * dois   // 20
```

Concatenação de Strings

O operador de adição além de somar tipos numéricos, também tem a responsabilidade de concatenar Strings, ou seja, ele terá comportamento diferente de acordo com o tipo de seus operandos.

A concatenação de duas Strings se dá pela junção da primeira com a segunda, de forma a que resultem em uma única String

```
// concatenação de Strings
"olá" + "mundo" // resulta em: "olámundo"
"oi " +  nome   // resulta em: "oi Hilton"
```

Operador de Resto (%)

Swift também apresenta o operador de resto, que retorna o resto de uma divisão inteira de seus operandos, ou seja, ele calcula quantos do segundo operando cabem dentro do primeiro, e retorna o que sobrou de espaço do primeiro.

```
// operador de resto
5 % 2   // retorna: 1
5 % 5   // retorna: 0
2 % 5   // retorna: 2
```

Operadores de Comparação

Durante o desenvolvimento de uma aplicação, muitas vezes precisamos verificar se dois valores são iguais, ou se um é maior que o outro. Para isso existem operadores de comparação. Esse tipo de operador compara os dois operandos, e retorna um Bool de acordo com o resultado dessa comparação. Se a comparação for verdadeira, retornará `true`, caso seja falsa, `false`.

```
// operadores de comparação
2 == 2  // dois é igual a dois? true
10 < 1  // 10 é menor que 1? false
```

```
nome != "Elton" // o valor de nome é diferente de "Elton"? true
idade >= 21     // o valor de idade é menor ou igual a 21? false
```

Os operadores de comparação básicos de Swift são:

Igual a	`x == y`
Diferente de	`x != y`
Maior que	`x > y`
Maior que ou igual	`x >= y`
Menor que	`x < y`
Menor que ou igual	`x <= y`

Operadores Lógicos

São os operadores que modificam ou combinam os valores do tipo Bool. São eles: Not/Não (!), And/E (&&), Or/Ou (||). Operadores lógicos, assim como os de comparação, por retornarem valores Booleanos, são muito usados no controle de fluxo da aplicação.

Not (!)

O operador Not faz uma negação do valor de seu operando. Ou seja, inverte o sentido existente. Se for verdadeiro tornará falso, se for falso, verdadeiro. Embora o símbolo de exclamação seja o mesmo do *force unwrapping* de valores opcionais, no contexto da operação lógica, ele é utilizado antes do operando.

```
// Not - Não
odeioSwift = false
!odeioSwift // retorna true
```

And (&&)

Serve para verificar se dois valores do tipo Bool são verdadeiros (`true`). Caso algum dos operandos seja falso (`false`), o resultado também será falso.

```
// And - E
true && true    // retorna true
false && true   // retorna false
false && false  // retorna false
```

Or (||)

Serve para verificar se algum dos valores é verdadeiro. Diferentemente do And, basta que apenas um valor seja true para que o resultado também seja.

```
// Or - Ou
true || true    // retorna true
false || true   // retorna true
false || false  // retorna false
```

Operadores de Intervalos

São utilizados quando o programador precisa de uma sequência de valores para realizar uma ação. Geralmente são combinados com loops do tipo for, que explicaremos em detalhe posteriormente.

Intervalos Fechados (`x...y`)

Retornam a sequência de números que fazem parte do intervalo do primeiro operando até o segundo. No intervalo fechado, incluímos os valores dos operandos na sequência retornada.

Intervalos Semi-abertos (x..<y)

Retornam uma sequência de números como nos intervalos fechados, mas não incluem o valor do segundo operando. Ou seja, terminam a sequência exatamente antes de chegar no valor de y.

```
// operadores de intervalos
0...3   // representa 0, 1, 2, 3
0..<3   // representa 0, 1, 2
```

Coleções

Coleções são formas de agrupar dados de um mesmo tipo. Elas podem ser ordenadas, como os Arrays, ou não, como os Dicionários. A não ser quando atribuídas a uma constante, coleções em swift são mutáveis, podendo crescer ou diminuir de acordo com a necessidade.

Arrays

É uma forma de coleção de dados que são indexados sequencialmente, começando do zero, ou seja, ao primeiro elemento de um Array, é atribuído o índice 0 (zero), ao segundo elemento, 1, e assim por diante. Cada elemento pode ser acessado pelo seu índice correspondente.

```
// arrays
var cidades: = ["Recife", "Olinda"]
cidades[0]  // retorna "Recife"
```

Podemos adicionar novos elementos à coleção através do uso do método append. O append coloca o novo elemento no fim do array.

```
// adicionando elementos ao array
cidades.append("Jaboatão")
cidades[2]  // retorna "Jaboatão"
```

Analogamente, podemos remover elementos do array. Para isso temos alguns métodos à disposição, que executam a remoção de um, todos, ou alguns elementos especificados.

```
// removendo elementos do array
cidades.removeLast()    // remove o último elemento ("Jaboatão")
cidades.removeFirst()   // remove o primeiro elemento ("Recife")
cidades.removeAll()     // remove todos os elementos
```

No exemplo acima, o array foi inicializado já com valores, mas também podemos iniciá-lo vazio, sem nenhum elemento. Para sabermos quantos elementos temos num array utilizamos a propriedade count.

```
// array vazio e contagem de elementos
var cidadesVisitadas: [String] = []
cidadesVisitadas.count  // retorna 0 (zero)

cidadesVisitadas.append("Recife")
cidadesVisitadas.count  // retorna 1
```

Como atribuímos o array a uma variável (cidadesVisitadas), podemos modificar seus elementos. Para substituir um valor específico em um array, basta que seja atribuído o novo valor ao índice desejado.

```
// modificando elemento
cidadesVisitadas[0] = "Olinda"
cidadesVisitadas    // ["Olinda"]
```

Tentar acessar um elemento que não faz parte do array, por exemplo, cujo índice é maior do que o tamanho do array, resultará em um erro de *Index out of Range* em tempo de execução. Por isso temos que tomar cuidado para nos mantermos dentro dos limites do array.

```
// acessando elemento inexistente (ERRO)
cidadesVisitadas[cidadesVisitadas.count]
cidades[-1]
```

No exemplo acima, ao tentar acessar o elemento do array cujo índice é igual a `cidadesVisitadas.count` nos deparamos com o erro mencionado acima. Arrays são

indexados a partir do zero, o que faz com que a contagem de seus elementos seja maior do que o valor do índice de seu último elemento, quando existe pelo menos um elemento.

Devido à sua grande importância no desenvolvimento de aplicações, existem diversas maneiras de modificar e manipular arrays, que podem ser encontradas na documentação de Swift.

Dicionários

São coleções de pares chave-valor (*key-value*). Diferente dos arrays, onde números são usados para indexar seus valores, nos dicionários utilizamos chaves. As chaves podem ser de tipos numéricos, strings, booleans, ou qualquer outro tipo que implemente o protocolo *Hashable* (veremos mais sobre protocolos nas próximas sessões). O importante é que as chaves devem ser de um mesmo tipo, ou seja, se uma delas for String, todas as outras devem ser também.

Os valores do dicionário também devem ser de um mesmo tipo, que não necessariamente é o mesmo das chaves. Valores são sempre associados a uma chave, de forma que quando quisermos acessar algum, buscaremos por sua chave no dicionário.

```
// dicionários
var capitais: [String: String] = ["PE": "Recife"]
capitais["PE"]   // retorna "Recife"
```

No exemplo acima, foi criado um dicionário cujas chaves são do tipo String, e cujos valores também são desse mesmo tipo. Acessamos o valor "Recife" guardado no dicionário através de sua chave correspondente: "PE". As chaves exercem função similar aos índices nos Arrays.

Para adicionarmos novos valores a um dicionário, basta que adicionemos uma nova chave e digamos seu valor correspondente.

```
// adicionando elementos ao dicionário
capitais["PB"] = "João Pessoa"
capitais    // ["PB": "João Pessoa", "PE": "Recife"]
```

Caso seja necessário modificar o valor armazenado por uma chave, basta que seja atribuído o novo valor a ser associado à chave já existente.

```
// modificando valor de uma chave
capitais["PB"] = "Maria Pessoa"
capitais    // ["PB": "Maria Pessoa", "PE": "Recife"]
```

Para remover uma entrada do dicionário, dizemos que uma chave não tem valor correspondente, ou seja, atribuímos `nil`. Também é possível fazer a remoção utilizando os métodos auxiliares, como o `removeValue(forKey:)` que assim como na outra abordagem, retira do dicionário a entrada cuja chave foi especificada.

```
// removendo elementos do dicionário
capitais["PB"] = nil    // remove entrada "PB: "Maria Pessoa"
capitais.removeValue(forKey: "PE")  // remove entrada "PE": "Recife"
```

Analogamente a Arrays, Dicionários podem ser inicializados sem elementos, e o número de elementos em um certo momento pode ser descoberto acessando a propriedade `count`.

```
// dicionários vazios e contagem de elementos
var dictVazio: [Int: String] = [:]
dictVazio.count // retorna 0
```

Ao tentar acessar um valor associado a uma chave que não faz parte do dicionário, teremos um retorno de *nil*, que evidencia que para aquele dicionário não existe valor associado àquela chave.

```
// acessando chave sem valor
dictVazio[0]    // retorna nil
dictVazio[-1]   // retorna nil
```

Voltaremos a falar mais sobre coleções quando abordamos *for-In loops*.

Fluxo de Controle

Durante o desenvolvimento de uma aplicação, por vezes necessitamos repetir a execução de uma parte do código, ou executar código de acordo com alguma condição. Essas ações são formas de alterar o fluxo de controle, e Swift nos dá algumas formas de fazer isso.

Desvios Condicionais

É muito comum precisar executar uma parte do código caso uma condição seja satisfeita, como por exemplo, quando um usuário preenche um campo ou não, quando ocorre um erro, quando é necessário verificar se o usuário tem permissão para fazer uma ação. Tudo isso são exemplos de desvios condicionais.

If

A forma mais simples de fazer desvios condicionais é usando If. O If serve para testar se o valor da variável ou expressão é true. Caso seja, será executado o bloco de código contido entre as chaves do if.

```
// if (se)
var meuNome = "José"
var saudacao = ""

if meuNome == "José" {
    // executar caso verdadeiro:
    saudacao = "olá, José"
}

saudacao    // tem valor de "olá, José"
```

No exemplo acima, conferimos se o valor guardado na variável meuNome é igual a "José". Se isso for verdadeiro, saudacao recebe o valor "olá, José".

If / else

Caso desejemos executar uma ação quando a expressão avaliada é verdadeira, e uma outra diferente quando a mesma for falsa, acrescemos ao if o else. Assim como o if, o else recebe um bloco de código entre chaves, porém este só será executado caso a expressão avaliada seja false.

```
// if/ else
idade = 17
var permissao = ""

if idade >= 18 {
    permissao = "Você é maior de idade"

} else {
    // executar caso falso:
    permissao = "Você é menor de idade"
}

permissao // tem valor de "Você é menor de idade"
```

No exemplo acima, dentro do if é feita a avaliação da expressão (idade >= 18), que resultará em false, visto que 17 é menor que 18. Sendo a condição do if falsa, o bloco de código do else é executado, atualizando o valor da permissão para indicar que a idade passada corresponde a um menor de idade.

else if

Caso seja necessário considerar mais de uma condição, e assim ter blocos de códigos correspondentes a essas novas condições, podemos encadear mais ifs após o else usando as palavras chaves else if.

Construindo em cima do exemplo anterior, não queremos mais só discriminar entre maior e menor de idade, queremos além disso saber se a idade é de um adolescente. Para isso adicionamos mais um if, com uma nova condição e um bloco de código associado. Caso nenhuma das condições dos dois ifs sejam satisfeitas, o bloco do else será executado.

```
// else if
idade = 16
var mensagem = ""
```

```
if idade >= 18 {
    mensagem = "você é um adulto"

} else if idade >= 12{
    mensagem = "você é um adolescente"

} else {
    mensagem = "você é uma criança"
}

mensagem // tem valor de "você é um adolescente"
```

Switch

Na medida que o número de condições que queremos checar cresce, o switch é uma melhor opção para tornar o código mais organizado do que encadear ifs e else ifs continuamente. No switch, um valor será testado se equivale a algum dos casos listados, executando o bloco de código correspondente.

Os casos do switch devem listar todos os possíveis valores que o valor comparado pode assumir. Como diversas vezes listar todos os possíveis casos não é desejável, ou necessário, adicionamos um caso *default*, que serve para quando nenhum dos outros casos é satisfeito.

```
// switch
var animal = "Cachorro"
var som: String

switch animal {

case "Gato":
    som = "miado"

case "Cachorro":
    som = "latido"

case "Leão":
    som = "rugido"

default:
    // se for qualquer outro animal
    som = "indefinido"
}
```

```
som // tem valor de "latido"
```

No exemplo acima, o valor a ser analisado é o que está na variável `animal`. Temos 3 casos onde comparamos o valor dela, e um default para tornar nosso switch exaustivo, isto é, ter uma ação para qualquer valor que `animal` possa guardar. Como o valor em questão é "Cachorro", o segundo `case` será acionado, colocando o valor "latido" na variável `som`.

Diferentemente de algumas linguagens de programação, em Swift não é necessário usar `break` ao final de cada caso do `switch`, pois apenas o primeiro caso que corresponder será executado, não correndo risco de ter mais de um caso executado na mesma iteração do switch.

Podemos fazer algumas modificações no exemplo anterior para que ele funcione para mais animais que fazem o mesmo som, como por exemplo leão e leoa. Para isso usaremos casos compostos.

```
// switch com casos compostos
animal = "Gata"
som = ""

switch animal {

case "Gato", "Gata":
    som = "miado"

case "Cachorro", "Cadela":
    som = "latido"

case "Leão", "Leoa":
    som = "rugido"

default:
    // se for qualquer outro animal
    som = "indefinido"
}

som // tem valor de "miado"
```

Agora cada caso verifica se o valor de animal é igual a qualquer um dos dois listados, ou seja, "Gato" ou "Gata" entrariam no mesmo `case`, atribuindo "miado" à variável `som`.

Mais informações sobre desvios condicionais podem ser encontrados na documentação da linguagem.

Loops

São formas de executar um mesmo bloco de código repetidas vezes. Usar loops evita replicação de código, por exemplo, se precisarmos executar uma mesma linha mil vezes não precisamos escrevê-la as mil vezes, basta que coloquemos ela dentro de um loop de mil iterações.

For-In

Loops deste tipo servem geralmente para varrer cada elemento de uma coleção (Array, Dicionários, …) ou de um intervalo numérico.

Um exemplo de uso de loop for-in é o seguinte: dado um array de cidades que já visitamos (cidadesVisitadas), como adicionar cada elemento desse array a uma mensagem?

```
// for-in
```

```
cidadesVisitadas = ["Recife", "Olinda", "Jaboatão"]
mensagem = "nós visitamos:"

for cidade in cidadesVisitadas {
    mensagem = mensagem + ", " + cidade
}

mensagem // tem valor de "nós visitamos: Recife, Olinda, Jaboatão"
```

No código acima, `cidade` é uma constante temporária que vai assumir o valor de cada elemento do array, um por vez. No caso, o bloco de código do for-in, que se encontra entre chaves({`mensagem = mensagem + " " + cidade`}), será executado uma vez para cada elemento do array cidadesVisitdas. E como o valor de cidade vai guardar o elemento da vez, teremos uma mensagem que cobre todas as cidades visitadas.

Para entender melhor como foi formada a mensagem, podemos visualizar como ela está em cada iteração do loop:

Execução	Valor de cidade	Valor de mensagem
Antes do loop	Não existe	nós visitamos:
Fim da 1a Iteração	Recife	nós visitamos: Recife
Fim da 2a Iteração	Olinda	nós visitamos: Recife Olinda
Fim da 3a Iteração	Jaboatão	nós visitamos: Recife Olinda Jaboatão
Após o loop	Não existe	nós visitamos: Recife Olinda Jaboatão

Outro exemplo de uso de loops for-in é: Construir um array com o quadrado dos dez primeiros números naturais diferentes de zero. Nesse caso não varremos uma coleção, e sim um intervalo numérico de um a dez.

```
// for in range
var quadrados: [Int] = []

for numero in 1...10 {
```

```
    let quadrado = numero * numero
    quadrados.append(quadrado)
}

quadrados // tem valor de [1, 4, 9, 16, 25, 36, 49, 68, 81, 100]
```

No exemplo acima utilizamos o operador de intervalo fechado (1...10) para gerar um intervalo de um a dez, incluindo o dez. Esse intervalo será varrido pelo for, em cada iteração numero assumindo o valor do número atual, primeiro 1, depois 2, e assim por diante até 10.

Dentro do corpo do loop calculamos o valor do quadrado do número atual e atribuímos ele a uma constante temporária, que em seguida será adicionada ao array de quadrados. Ao final do loop temos os valores esperados: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].

While

O while é um tipo de loop que será executado enquanto uma condição for verdadeira. Antes da primeira execução de seu corpo, e após cada execução do mesmo, uma nova avaliação é feita para saber se a condição é verdadeira, continuando a execução do loop, ou se é falsa, terminando assim o loop.

```
// while
var preco = 100
var devoComprar = false
```

```
//enquanto o preço for caro não devo comprar
while !devoComprar {

    preco = preco - 1 // preço diminui

    //se o preço chegar em 50, eu devo comprar.
    if preco == 50 {
        devoComprar = true
    }
}

preco       // tem valor de 50
devoComprar // tem valor de true
```

No exemplo acima, observamos o preço de um produto diminuir ao longo do tempo (em cada iteração do loop), até que chegue em um valor desejado (50). Uma vez que esse valor seja atingido, já posso comprar o produto (`devoComprar = true`).

Como o loop ocorria enquanto eu não devesse comprar, ao inverter o valor de `devoComprar` a condição do `while` não mais é verdadeira, o que faz com que o loop se encerre, dando continuidade à execução do programa, agora com os novos valores de `preco` e `devoComprar`.

Funções

Funções são blocos de códigos que recebem um nome para que possam ser chamados de outras partes do código, ou até de dentro de si mesmo (recursão). Funções são partes integrais do desenvolvimento de uma aplicação, e proporcionam uma modularização que ajuda a obter um código mais organizado.

Abaixo veremos um exemplo de uma função simples que retorna o maior valor entre dois números, e em seguida comentaremos sobre cada parte de sua declaração.

```
// Funções
func maior(primeiro: Int, segundo: Int) -> Int{
    if primeiro > segundo {
        return primeiro
    } else {
        return segundo
    }
}
```

Para definir uma função é necessário iniciar com a palavra-chave func, seguida do nome que queremos dar para a função, seus parâmetros (caso existam), seu tipo de retorno (caso exista), e o corpo da função envolto em chaves.

O nome de uma função é como ela será chamada pela sua aplicação. É boa prática tentar dar o nome mais descritivo possível, que realmente reflita o que a sua função faz, de forma com que muitas vezes não seja necessário abrir sua definição para entendê-la. No exemplo acima, foi escolhido o nome `maior`.

Parâmetros são nomes dados aos valores que vão ser passados para serem utilizados no corpo da função. Assim como numa declaração de uma variável, a definição dos parâmetros segue o modelo de `nome: tipo`, onde nome é como o parâmetro será referenciado dentro do corpo da função, e tipo representa o tipo do valor (Int, Bool, String, …).

Uma função pode ter quantos parâmetros necessários, ou até nenhum. No caso de não ter parâmetros, os parênteses ainda são necessários. No exemplo acima, os parâmetros são chamados primeiro e segundo, e ambos são do tipo Int.

Após a definição dos parâmetros, se indica o tipo de retorno da função. O tipo de retorno representa qual o tipo do valor que será retornado ao fim da execução da função. Caso a função não retorne nenhum valor, dizemos que o tipo de retorno dela é Void. Void é

um tipo especial que é representado por uma tupla vazia: (). Nesse caso, a inclusão do tipo de retorno na declaração da função é opcional.

A sintaxe usada para expressar o tipo de retorno é um hífen seguido de um maior-que, formando uma seta, e em seguida o tipo a ser retornado. Na função de exemplo, como ela retorna um dos elementos que recebeu como parâmetro, seu tipo de retorno tem que ser o mesmo dos parâmetros: `-> Int`.

No corpo da função é onde estará o bloco de código que ela abstrai. Dentro desse bloco pode ser feito basicamente qualquer coisa que é feita fora dele, de declaração de variáveis a definição de classes, porém tudo que for declarado dentro dentro do corpo não existe fora dele. Na função acima, o corpo consiste de um if/else onde é feita a conferência de qual valor é maior que o outro.

Por fim, a palavra-chave return denota qual valor vai ser retornado pela função. Caso a função não tenha retorno, o return não estará presente no corpo da mesma. O return é executado somente uma vez, pois logo após, é finalizada a execução da função. No exemplo de função utilizado, existe mais de uma linha contendo o return, porém é garantido que só um deles será executado por estarem em lados opostos de um if/else: `return primeiro`, e `return segundo`.

Para chamar uma função, isso é, fazer com que o código no corpo dela seja executado, basta colocar o nome da função seguido de parênteses envolvendo os parâmetros da mesma. Os valores passados como parâmetros são chamados de argumentos.

```
// chamando função
let um = 1
let dois = 2

let maximo = maior(primeiro: um, segundo: dois)
maximo // tem valor de 2
```

Tipo de Funções

Em swift, funções também são tipos, o que faz com que elas possam ser armazenadas em variáveis/constantes, e até passadas como parâmetros de outras funções. O tipo de uma função é dado pelo tipo de seus parâmetros e de seu retorno, separados por uma seta. A função maior definida acima é do tipo: `(Int, Int) -> Int`, dado que seus dois parâmetros de entrada e seu retorno são do tipo Int.

Funções que esperam outras funções como parâmetro são chamadas de funções de primeira ordem. Em Swift temos algumas funções desse tipo, como por exemplo `map`. Map é um método dos arrays que recebe uma função como entrada, e aplica essa função a cada elemento do array, retornando o novo array modificado.

```
// funcao de primera ordem (map)
var array = [1, 2, 3]

func somaUm(valor: Int) -> Int {
    return valor + 1
}

// map aplica a funcao somaUm a cada elemento de array
```

```
var novoArray = array.map(somaUm)

novoArray   // tem valor de [2, 3, 4]
```

No exemplo acima, a função `somaUm` recebe um valor e retorna o mesmo somado a 1, ela é do tipo `(Int -> Int)`. Nesse exemplo, o map espera como parâmetro uma função exatamente desse tipo, pois Int é o tipo dos elementos do array. O uso do map no código acima é equivalente a usar:

```
for element in array {
    let elementoSomado = somaUm(valor: element)
    novoArray.append(elementoSomado)
}
```

Closures

Assim como funções, closures abstraem blocos de códigos. Em algumas linguagens de programação eles recebem o nome de blocos, lambdas, ou funções anônimas. Diferentemente das funções tradicionais que vimos na seção anterior, os closures não tem um nome, eles são auto-contidos, geralmente sendo usado como parâmetros de funções de primeira ordem.

Um closure pode ser definido diretamente onde ele é chamado, por exemplo, no código abaixo, transformaremos a função somaUm em um closure que será usado num map, obtendo o mesmo resultado.

```
// closures
array = [1, 2, 3]
novoArray = []

novoArray = array.map({ (numero: Int) -> Int in
    return numero + 1
})
```

```
novoArray // tem valor de [2, 3, 4]
```

Como vimos anteriormente, esse map espera como parâmetro uma função do tipo `(Int) -> Int`, e o closure é exatamente deste mesmo tipo. Um closure deve ser envolto em chaves, podendo ter os nomes e tipos dos parâmetros de entrada, o tipo de retorno, seguido da palavra chave `in`, e do seu corpo.

Analogamente a uma função, podemos dar nomes aos parâmetros, e indicar seu tipo, além do tipo de retorno. Em closures, toda essa parte da assinatura pode ser omitida, por ser inferida do contexto em que ele está sendo passado. No exemplo, o map já sabe o tipo de função que ele espera, então pode assumir que o closure será desse mesmo tipo.

Além de inferir os tipos dos parâmetros em closures, já existem nomes pré-determinados para esses parâmetros para serem usados quando não forem dados nomes explícitos aos mesmos. Esses nomes pré-determinados começam em `$0` e prosseguem de acordo com o número de parâmetros necessários, `$1`, `$2`, `$3`, ….

A palavra-chave `in` delimita o fim da assinatura, e o começo do corpo do closure. Quando a assinatura for omitida, o in também pode ser omitido, restando no closure apenas o seu corpo.

Para closures de apenas uma linha, é possível também omitir a palavra-chave return, quando se é esperado que seja retornado um valor pelos mesmos. Uma forma mais curta de escrever o mesmo closure acima seria a seguinte:

```
// closure enxuto
array = [1, 2, 3]
novoArray = []

novoArray = array.map {$0 + 1}

novoArray // tem valor de [2, 3, 4]
```

Quando o closure é o último argumento da função, podemos colocar as chaves após os parênteses, e caso o closure seja o único argumento, podem-se omitir os parênteses.

Enum

Enumerações (Enumerations) são uma forma de de criar um novo tipo para valores que são relacionados de alguma forma. Enums apresentam várias funcionalidades que

geralmente são atribuídas a classes, como métodos e propriedades computadas que serão explicados na seção de classes.

A sintaxe de um Enum em swift é a seguinte:

```
// enum
enum Familia {
    case pai
    case mae
    case filho
    case filha
}
```

A palavra-chave enum define o começo da enumeração, o nome do enum deve ser iniciado com letra maiúscula, pois vai ser o nome do novo tipo. Dentro do corpo do enum listamos os casos, os valores desse tipo. No exemplo acima, criamos um novo tipo, Familia, que pode ter valores pai, mae, filho, e filha.

Durante o desenvolvimento de uma aplicação, quando confrontado com um valor do tipo de um enum, é muito comum o uso do switch para executar ações de acordo com o valor específico. Para acessar um valor de um enum, utilizamos o nome do enum seguido de um ponto e o nome do caso.

```
// enum com switch
let membro = Familia.pai
var mensagem = ""

switch membro {
case .pai, .mae:
    mensagem = "primeira geração"

case .filho, .filha:
    mensagem = "segunda geração"
}

mensagem    // tem valor de "primeira geração"
```

Perceba que no exemplo acima, nos casos do switch não usamos a notação completa para representar os valores do enum. Isso é possível quando pode ser inferido pelo contexto qual o tipo do valor sendo comparado no switch. Como já se sabe que membro é do tipo Familia, podemos assumir que os valores comparados no case também são desse mesmo tipo, e assim omitir o nome do mesmo.

Valor associado

Às vezes torna-se necessário ter mais informações para cada caso do enum que existam juntamente ao valor do caso, e que possam variar em cada uso do mesmo. Para isso utilizamos valores associados, que podem ser de qualquer tipo, e inclusive diferentes para cada caso.

```
// enum com valores associados
enum NovoTipo {
    case palavra(valor: String)
    case inteiro(valor: Int)
}
```

No exemplo acima, estamos criando encapsulamentos em português para os tipos já existentes em Swfit, e chamando eles de NovoTipo. No caso de ser uma palavra, temos um valor associado que é do tipo String. Caso seja um inteiro, o valor associado é do tipo Int. Para realizar ações com os valores associados a cada tipo podemos fazer o seguinte switch:

```
// switch de valores associados
let numero = NovoTipo.inteiro(valor: 8)
mensagem = ""

switch numero {
case .palavra(let valor):
    mensagem = "a palavra tem valor: \(valor)"

case .inteiro(let valor):
    mensagem = "o inteiro tem valor: \(valor)"
}

mensagem    // "o inteiro tem valor: 8"
```

Dentro do casamento de valor do switch, adicionamos uma constante (let valor) no lugar do valor associado, para podermos utilizá-lo dentro do corpo do case. Na formação da mensagem utilizamos uma técnica chamada interpolação de strings, para colocar um valor dentro de uma string sem utilizar concatenaç ao. A sintaxe da interpolação consiste de uma barra invertida e o nome do valor desejado entre parênteses.

Raw value

Uma alternativa ao uso de tipos associados é utilizar o *raw value*. Nessa alternativa, é necessário que todos os cases armazenem valores de um mesmo tipo, como por exemplo:

```
// enum com raw value
enum Ordem: Int {
    case primeiro = 1
    case segundo = 2
    case terceiro
}

Ordem.primeiro.rawValue  // tem valor de 1 (Int)
Ordem.terceiro.rawValue  // tem valor de 3 (Int)
```

O enum Ordem tem a seus casos valores do tipo Int atribuídos, ou seja, quando acessarmos a propriedade rawValue de um de seus tipos, receberemos um valor do tipo Int. A sintaxe de Swift exige que indiquemos qual tipo do rawValue após o nome do enum. No exemplo acima, explicitamente atribuímos os valores 1 e 2 aos dois primeiros casos no nosso enum, mas mesmo omitindo o valor do terceiro caso, ele pode ser implicitamente atribuído.

Classes e Structs

Classes e Structures (struct) são parte integral do desenvolvimento de aplicações, servindo como blocos básicos para construção de abstrações do mundo real. Ambas têm grandes similaridades, por isso abordaremos as duas em conjunto. Elas podem conter propriedades que armazenam valores, e métodos que adicionam funcionalidades. As propriedades seguem a mesma sintaxe das variáveis/constantes, enquanto os métodos seguem a das funções. Abaixo veja um exemplo de classe e outro de struct com propriedades armazenadas.

```
// classes e structs
struct Endereco {
    var cidade: String
    var logradouro: String
    var numero: Int?
}

class Pessoa {
    var nome: String = ""
    var endereco: Endereco?
    var telefone: String?
}
```

Nomes dados a classes e structs devem ser iniciados com letra maiúscula, para manter o padrão dos tipos em Swift (Int, String, Bool, …), e o nome das propriedades devem ser iniciadas com letras minúsculas para diferencia-las dos tipos.

No exemplo acima, tanto Pessoa quanto Endereco tem cada um três propriedades armazenadas. Se uma classe não tiver um construtor explícito, é obrigado que a cada propriedade se tenha um valor atribuído, como em Pessoa acima, nome recebeu o valor da

string vazia (""). As outras propriedades de Pessoa, por serem opcionais recebem o valor nil.

Vamos adicionar o construtor de Pessoa agora, isso é, o método que é chamado toda vez que for criado um novo objeto da classe Pessoa.

```
class Pessoa {
    let nome: String
    var endereco: Endereco?
    var telefone: String?

    init(nome: String, endereco: Endereco?, telefone: String?) {
        self.nome = nome
        self.endereco = endereco
        self.telefone = telefone
    }
}
```

O construtor é um tipo de método especial, que inicia com a palavra-chave init, e em seguida recebe os parâmetros de entrada. Como o construtor tem função de inicializar os valores das propriedades da classe, é interessante que seus parâmetros sejam valores para serem atribuídos às propriedades. No corpo do construtor, são feitas as atribuições, usando a palavra-chave self. Self se refere à classe em que estamos, sendo assim uma forma de diferenciar a propriedade da classe do parâmetro, visto que elas têm o mesmo nome.

Ao escrever self.nome = nome, estamos dizendo que a propriedade da classe atual chamada nome, receberá o valor guardado no parâmetro de mesmo nome (nome). Agora que temos o construtor, vamos criar instâncias de Pessoa e Endereco:

```
// instanciando classe e struct
let apolo235 = Endereco(cidade: "Recife", logradouro: "Rua do Apolo", numero:
235)

let joao = Pessoa(nome: "João", endereco: apolo235, telefone: nil)
```

Para criar uma instância de uma classe/struct, chamamos o seu construtor através do nome da classe/struct, seguido dos parâmetros que o construtor recebe. No caso da struct Endereco, não havíamos definido um construtor explicitamente, mas diferentemente das classes, um construtor implícito é definido que recebe valores para cada propriedade dela.

Após atribuir as instâncias a constantes, podemos acessar os valores de suas propriedades usando a notação de ponto: nome da variável, seguido de um ponto e o valor da propriedade desejada.

```
// acessando propriedades
joao.nome       // tem valor de "João"
joao.telefone   // não tem valor (nil)

joao.endereco?.logradouro   // "Rua do Apolo"
apolo235.logradouro         // também "Rua do Apolo"
```

Value type vs Reference types

Uma grande diferença entre structs e classes são como seus tipos são passados para variáveis/constantes. Structs são tipos de valores (value type), o que significa que ao passar uma struct para uma variável, a mesma é copiada, gerando assim uma nova versão da struct original, de forma com que modificações feitas a uma cópia não resultam em mudanças na outra.

```
// value type
var copiaApolo = apolo235

apolo235.cidade = "Olinda"

copiaApolo.cidade   // tem valor de "Recife"
apolo235.cidade     // tem valor de "Olinda"
```

No exemplo acima, o valor em apolo235 que é do tipo Endereco é copiado para a variável copiaApolo. Ao mudar a propriedade cidade de apolo235 para "Olinda", o valor da mesma é alterado, mas a mudança não é refletida em copiaApolo por se tratar de uma

cópia feita no momento da atribuição. O que foi passado para copiaApolo foi um valor, e não uma referência.

Classes funcionam de forma diferente. Ao atribuir uma classe a uma segunda variável passamos uma referência dela (reference type), ou seja, qualquer mudança que ocorrer na segunda variável reflete na primeira, e vice-versa. Ambas variáveis guardam a referência para um mesmo objeto.

```
// reference type
var referenciaJoao = joao

joao.telefone = "99999999"

joao.telefone           // tem valor de "99999999"
referenciaJoao.telefone // tem valor de "99999999"
```

No exemplo acima, foi atribuído a referenciaJoao a mesma referência a um objeto da classe Pessoa que havia sido atribuído a joao, por isso, mudanças em joao refletem em mudanças em referenciaJoao.

Métodos

Métodos são funções declaradas dentro do corpo de uma classe/struct/enum. Geralmente eles têm a mesma sintaxe vista anteriormente em Funções, o construtor sendo um caso especial com sintaxe diferente.

O tipo mais comum de métodos são os de instância estão associados a uma instância específica de uma classe/struct/enum, isso é, é necessário inicializar um objeto da classe para poder chamar o método.

```
class Pessoa {
    let nome: String
    var endereco: Endereco?
    var telefone: String?

    init(nome: String, endereco: Endereco?, telefone: String?) {
        self.nome = nome
        self.endereco = endereco
        self.telefone = telefone
    }
    // método de instância
```

```
    func apresentar() -> String {
        return "Oi, meu nome é \(self.nome)"
    }
}

// chamando método de instância
joao.apresentar()   // retorna "Oi, meu nome é João"
```

No exemplo acima, adicionamos o método de instância apresentar, que retorna uma string de apresentação contendo a propriedade nome da instância. Para chamar o método, usamos a mesma notação de ponto utilizada para acessar propriedades, com o acréscimo dos parênteses contendo os argumentos da função (se houver algum).

Herança

Outra diferença entre classes e structs é que classes permitem o uso de herança. Herança ocorre quando queremos que uma classe herde métodos e propriedades de uma classe pré-definida. A classe que herda o comportamento é chamada de subclasse.

No desenvolvimento de aplicativos iOS é muito comum o uso de herança, por exemplo, a classe padrão que define um botão é chamada UIButton. Se quisermos criar nosso próprio tipo de botão, que faz tudo que um botão tradicional faz, porém também queremos adicionar novos comportamentos, devemos fazer nossa nova classe herdar de UIButton.

```
// herança
class Professor: Pessoa {
    var titulo = "Prof."
}

var pedro = Professor(nome: "Pedro", endereco: apolo235, telefone: "1234567")

pedro.nome      // tem valor de "Pedro"
pedro.titulo    // tem valor de "Prof."
```

No exemplo acima criamos uma subclasse de Pessoa, Professor. Professor herda todos os comportamentos de Pessoa, e adiciona uma nova propriedade chamada titulo com o valor de "Prof.". Podemos chamar o construtor de Pessoa para criar uma instância de Professor, que vai conter todas as mesmas propriedades de uma Pessoa e a nova propriedade definida.

Subclasses não precisam apresentar exatamente o mesmo comportamento da classe de que herda, ela pode sobrescrever métodos e propriedades, implementando seu próprio comportamento.

```
class Professor: Pessoa {
    var titulo = "Prof."

    //sobrescrevendo método
    override func apresentar() -> String {
        return "Oi, meu nome é \(self.titulo) \(self.nome)"
    }
}

// chamando método sobrescrito
pedro.apresentar()  // retorna "Oi, meu nome é Prof. Pedro"
```

Sobrescrevemos o método apresentar da classe Pessoa, para adicionar o título do professor na String de apresentação. Para sobrescrever um método basta adicionar a palavra-chave override e dar a nova implementação do método.

Se quisermos adicionar uma nova propriedade à nossa subclasse contendo o nome da universidade em que o professor ensina, precisamos que um valor para a mesma seja atribuído no construtor de Professor. Para isso vamos criar um novo construtor, mas para não ter que replicar todo o código do construtor da superclasse, podemos apenas chamar o mesmo usando a palavra chave super.

```
class Professor: Pessoa {
    //...

    // nova propriedade
    var universidade: String

    init(nome: String, endereco: Endereco?, telefone: String?, universidade:
String) {
        self.universidade = universidade
        //chamando construtor da superclasse
        super.init(nome: nome, endereco: endereco, telefone: telefone)
    }
}

var pedro = Professor(nome: "Pedro", endereco: apolo235, telefone: "1234567",
```

```
universidade: "UFPE")

pedro.universidade  // tem valor de "UFPE"
```

Professor agora tem uma propriedade universidade, com valor atribuído em seu construtor. Construtor que faz o mesmo que o de Pessoa (superclasse), acrescido da nova atribuição a universidade.

Protocolos

Classes que conformam com um protocolo devem implementar todos os requerimentos definidos pelo protocolo específico. Estes requerimentos podem ser propriedades ou métodos. Protocolos têm função parecida com Interfaces de outras linguagens de programação.

Para definir um protocolo, utilizamos a palavra-chave protocol seguida do nome do protocolo, e o seu corpo envolto em chaves. Dentro do corpo do protocolo definimos a assinatura de suas propriedades e/ou métodos requeridos. Cabem às classes que conformam com um protocolo apresentar uma implementação para tais métodos e propriedades.

Abaixo veremos um exemplo de um protocolo simples, que pede uma propriedade e um método:

```
// protocolo
protocol Mestre {
    var universidadeMestrado: String? {get set}

    func receberTitulo(universidade: String)
}
```

Para conformar com o protocolo Mestre, é necessário que uma classe possua uma propriedade chamada universidadeMestrado do tipo String?, e um método receberTitulo que recebe uma parâmetro chamado universidade do tipo String e não retorna nada. As palavras-chaves get e set indicam que a variável não pode ser uma constante, ou só de leitura. Deve ser possível acessar o valor da mesma (get), e alterá-lo (set).

Vamos agora fazer com que nossa classe Professor conforme com o protocolo Mestre. Para isso, precisamos adicionar o novo método e a nova propriedade, além de indicar o nome do protocolo na assinatura da classe, após o nome da sua superclasse.

```
class Professor: Pessoa, Mestre {
    // requisitos do protocolo
    var universidadeMestrado: String?

    func receberTitulo(universidade: String) {
        self.universidadeMestrado = universidade
        self.titulo = "Ms."
    }
    // ...
}
```

Na classe Professor, a implementação do método receberTitulo(universidade) atribui o valor recebido à propriedade universidadeMestrado, e atualiza a propriedade titulo (que antes tinha valor "Prof.") para refletir a nova graduação. Podemos ver o método do protocolo em ação abaixo:

```
// chamando método do protocolo
pedro.receberTitulo(universidade: "MIT")

pedro.titulo    // tem valor de "Ms."
pedro.universidadeMestrado  // tem valor de "MIT"
```

Embora não implementem nenhuma funcionalidade, protocolos são considerados tipos em Swift, ou seja, podemos declarar uma variável do tipo Mestre, e atribuir a ela um objeto da classe Professor, ou de qualquer outra classe que implemente o protocolo.

```
// protocolos como tipos
let mestre: Mestre = pedro
```

Casting de Tipos

É uma forma de conferir o tipo de uma instância, ou de tratar uma instância como sendo de uma subclasse. Para isso, Swift tem dois operadores: is e as. O operador is verifica se um valor é de uma determinada classe, retornando true caso seja, e false caso contrário.

```
// checagem de tipo
let maria = Pessoa(nome: "Maria", endereco: apolo235, telefone: nil)

maria is Pessoa     // retona true
maria is Endereco   // retorna false
```

No exemplo acima obtivemos true pois maria referencia uma instância da classe Pessoa, e false pois ela não é do tipo Endereco.

O operador as é usado quando temos uma instância de um tipo, mas queremos tratá-lo como sendo de uma subclasse. Como nem sempre será possível realizar esse tratamento, o operador de Downcasting geralmente é utilizado em conjunto com opcionais (as? ou as!). O uso de force unwraping (as!) deve ser feito apenas quando tivermos certeza de que o downcasting funcionará, caso contrário encontraremos um erro de execução.

```
// downcasting
var pessoas: [Pessoa] = [pedro, maria]
var mensagem = "título(s): "

for pessoa in pessoas {
    if let professor = pessoa as? Professor {
        mensagem = mensagem + professor.titulo
    }
}
```

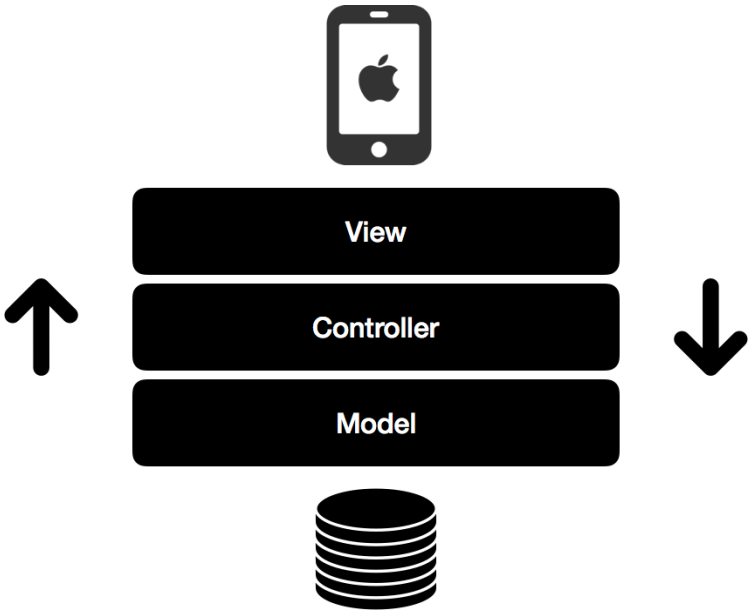
```
mensagem // tem valor de "título(s): Ms."
```

No exemplo acima, dentro do for-in tentamos fazer downcasting dos objetos que sabemos ser do tipo Pessoa para a subclasse Professor. Como apenas pedro é professor, só ele teve seu título (titulo) adicionado à mensagem. No caso de maria, o downcasting falha retornando nil, o que faz com que não passe no optional binding.

Model-View-Controler (MVC)

Agora que você já sabe os principais fundamentos de programação em Swift, vamos ver como funciona uma aplicativo iOS, como se dispõe os arquivos e como eles interagem.

Para desenvolver um aplicativo, a arquitetura mais comumente utilizada, e recomendada pela própria Apple é o padrão *Model-View-Controller* (MVC). Nessa arquitetura temos três camadas principais com responsabilidades definidas para nos ajudar a ter um código organizado e menos propenso a erros e inconsistências.



A primeira camada do modelo é o **Modelo** (*model*), que é responsável por manter os dados, objetos, comunicação com a rede, e regras de negócio da aplicação. O modelo é a camada mais back-end, estando mais distante do usuário final, idealmente não se conectando diretamente com a interface gráfica, porém sendo atualizado de acordo com mudanças provindas da mesma.

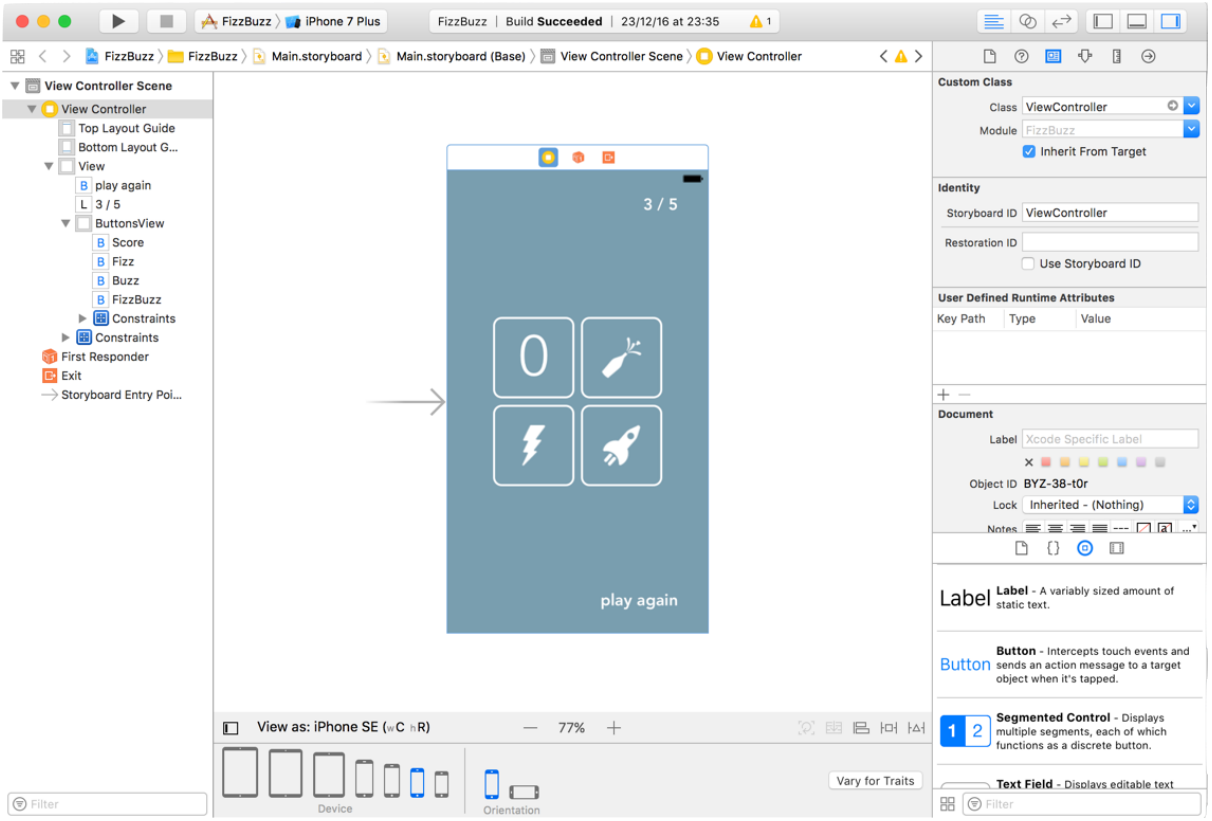
A segunda camada fica no lado oposto (front-end). A **View** é responsável por todas interações com o usuário, tudo que é mostrado graficamente no iPhone. É onde os dados do modelo são mostrados e possivelmente editados, mesmo as duas camadas não sendo diretamente conectadas. O framework UIKit apresenta uma variedade de views para serem usadas em apps iOS, que vão de simples labels que mostram texto, a botões e estruturas mais complexas como tableViews.

A terceira e última camada é chamada de **Controlador** (*controller*), e tem a responsabilidade de fazer a comunicação entre a view e o modelo em ambos os fluxos de dados. O controlador é responsável por interpretar as ações feitas pelo usuário nas views, propagando mudanças ou criação de dados para o modelo. Na mão contrária, também é responsável por atualizar as views de acordo com os dados do modelo.

Quando criamos um projeto iOS no Xcode, já são criados por padrão três arquivos: Main.storyboard, ViewController.swift, e AppDelegate.swift. Esses três arquivos são de suma importância no desenvolvimento da aplicação, e agora vamos falar um pouco mais sobre eles.

Main.storyboard

É o storyboard principal, onde vão ficar suas estruturas de views montadas através do interface builder. O storyboard é geralmente onde são dispostos e customizados os elementos da interface gráfica: botões, labels, ...



No storyboard podemos simular as dimensões e orientações de diferentes modelos de iPhones e iPads, para visualizarmos como nossas aplicações vão ser dispostas nestes dispositivos e assim ajustá-las de acordo.

O storyboard provê uma forma gráfica de montar interfaces gráficas. O mesmo resultado pode ser atingido por código, fazendo com que o uso do mesmo não seja obrigatório, mas por vezes recomendado.

As views dispostas no storyboard são estáticas a não ser que ligadas a um ViewController, que como o nome sugere é um controlador da view.

ViewController

Nos ViewControllers (VC) da aplicação passaremos os dados do modelo a serem mostrados na views, além de interpretar o significado de interações do usuários com as mesmas, refletindo as mudanças no modelo e na interface. ViewControllers são

subclasses de UIViewControllers, e por isso apresentam uma espécie de ciclo de vida evidenciado por, mas não restrito ao método viewDidLoad, já presente quando criamos um novo VC.

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

}
```

O viewDidLoad é um dos principais métodos utilizados nesses controladores, por ser chamado sempre logo após o carregamento da view, possibilitando que dados sejam mostrados na mesma. Se a view principal tem uma label dentro que mostra o nome do usuário, poderíamos no viewDidLoad requisitar essa informação do modelo e atualizar a label após o recebimento do dado.

É um problema comum no desenvolvimento iOS usando MVC que os ViewControllers fiquem inchados, com centenas de linhas, o que é ruim para testes, leitura, e manutenção do código. É importante designar ao VC apenas o que for responsabilidade dele para tentar evitar *Massive View Controllers*.

Para conectar views do storyboard com ViewControllers utilizamos outlets, ligações que dão um nome à view de forma que ela possa ser manipulada pelo código. Também podemos utilizar actions, que criam métodos que serão chamados quando ocorre a interação especificada com a view, por exemplo, o toque em um botão.


```
2  //  ViewController.swift
3  //  Apostila da Jornada 2017
4  //
5  //  Created by Hilton Pintor Bezerra Leite on 12/07/17.
6  //  Copyright © 2017 hpbl. All rights reserved.
7  //
8
9  import UIKit
10
11 class ViewController: UIViewController {
12
13     override func viewDidLoad() {
14         super.viewDidLoad()
15         // Do any additional setup after loading the view,
               typically from a nib.
16         tituloLabel.text = "aperte o botão"
17     }
18
19     override func didReceiveMemoryWarning() {
20         super.didReceiveMemoryWarning()
21         // Dispose of any resources that can be recreated.
22     }
23
24     @IBOutlet weak var tituloLabel: UILabel!
25
26     @IBAction func apertarBotao(_ sender: Any) {
27         tituloLabel.text = "Olá, Mundo!"
28     }
29
30 }
31
32
```

Na imagem acima, temos uma view no storyboard com duas subviews: um botão e um label. Ao lado o ViewController correspondente, com um outlet (linha 24) que nos permite referenciar a label, e uma action (linha 26-28) que é chamada quando ocorre um toque no botão.

No viewDidLoad trocamos o texto da label através de uma atribuição à propriedade text da mesma (linha 16). Após um toque no botão, o texto da label é atualizado para conter a string "Olá, Mundo!" (linha 27).

AppDelegate

O AppDelegate é onde é feita a comunicação da aplicação com o sistema operacional. Nele podemos definir como queremos que nossa aplicação reaja em diversas situações de uso, por exemplo, o que acontece quando o usuário abre a aplicação, quando troca de aplicativo (deixando em background), ou até quando fecha a aplicação.

Para reagir a essas situações de usos, o AppDelegate apresenta métodos similares aos do ciclo de vida do ViewControllers, que são chamados nos momentos certos.

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive
state. This can occur for certain types of temporary interruptions (such as an
incoming phone call or SMS message) or when the user quits the application and
it begins the transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and
invalidate graphics rendering callbacks. Games should use this method to pause
the game.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data,
invalidate timers, and store enough application state information to restore
your application to its current state in case it is terminated later.
        // If your application supports background execution, this method is
called instead of applicationWillTerminate: when the user quits.
```

```
    }

        // ...

    func applicationWillTerminate(_ application: UIApplication) {
        // Called when the application is about to terminate. Save data if
 appropriate. See also applicationDidEnterBackground:.
    }
}
```

Conclusão

Assim como qualquer outra linguagem de programação, para aprender Swift e desenvolver aplicativos iOS não basta apenas a leitura, é necessário praticar. Ao colocar a mão na massa, ao escrever e testar seu app, vão surgir dúvidas e questionamentos, para isso, swift conta com uma documentação extensa e bem escrita. Para mais informações e referências, recomendo leitura sob demanda da mesma.

Espero que ler esse material tenha sido um aprendizado tão grande para você como foi para mim escrevê-lo. Estou ansioso para ver seus futuros apps!