iOS Swift - Core 100

iOS Swift - Core100

Autoria:
Tiago Santos de Souza
Roberto Rodrigues Junior

Revisão:
Tiago Santos de Souza

Design, edição e produção:
Tiago Santos de Souza

Edição nº4
Novembro de 2017

Sumário

Apresentação

Durante muitos anos a **Apple** construiu uma linha de produtos altamente consumida por seus seguidores e a cada ano evolui mais com novas tecnologias e ideias. Um dos pilares desta evolução, é o seu sistema operacional amigável para o usuário e poderoso no processamento das informações.

Uma das propostas da Apple é a participação dos seus usuários na evolução dos produtos, e pensando nisso ofereceu ao longo dos últimos anos diversas ferramentas que foram utilizadas por programadores para o desenvolvimento de sistemas personalizados.

Desde o surgimento da linguagem C e da C++, diversas linguagens de programação surgiram, e a Apple sempre esteve presente nesses ciclos, com as linguagens **AppleScript, Objetive-C** e diversas outras linguagens e ferramentas para programação.

Podemos citar o sucesso do **Objective-C** que ao longo dos anos foi cada vez mais aceita junto aos seus usuários, pois tinham o melhor em hardware em suas mãos e uma linguagem que conseguia materializar os melhores aplicativos para o mercado.

Entretanto a evolução é necessária e pensando ainda mais sobre as novas e futuras tecnologias, a Apple avança mais um passo, oferecendo aos seus usuários uma nova linguagem de programação que celebra mais uma vez esta aliança com seus consumidores.

Neste treinamento iremos apresentar a linguagem de programação desenvolvida pela Apple, a **Swift.**

A Swift nasceu com uma quantidade de qualificações e novidades que a coloca em um patamar diferenciado de outras linguagens de programação. Qualidades que podemos destacar como segura, contemporânea, rápida, atualizada e de alto nível.

Todo este DNA da linguagem se mostra no momento do desenvolvimento de aplicativos, que vão desde os mais simples aos mais complexos e completos do mercado.

Para os que já programavam em Objective-C, foi um avanço no desenvolvimento de rotinas mais robustas e com processamento ainda mais ágil.

Para os novos programadores, será uma linguagem de programação muito mais fácil de aprender, flexível no desenvolvimento e principalmente divertida. Para aqueles com certa familiaridade com programação, será como um avanço relacionado a conceitos já implementados.

A Swift está em constante evolução desde o fim de 2015, quando a Apple disponibilizou o código fonte. Diversas sugestões da comunidade são atendidas e implementadas a cada ano!

Por isso, fique de olho no repositório da linguagem e no site oficial da linguagem para acompanhar sua evolução. Os endereços são:

Site da Swift: https://swift.org

Github: https://github.com/apple/swift

Consultas à documentação oficial sempre podem exclarecer aspectos em diversos pontos dos seus projetos. Sempre que possivel, mantenha-se atualizado na leitura e prática da documentação:

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Compatibility.html

Sejam bem-vindos a Swift, e para iniciarmos nosso treinamento não podemos deixar para trás a já conhecida e famosa frase de boas vindas para aprender uma nova linguagem de programação:

```
print (“Hello world!”)
print (“Bons Estudos!”)
```

Capítulo 1

Primeiros passos

Seção 1-1

O Xcode 9

Durante os módulos do nosso curso, utilizaremos o aplicativo de desenvolvimento integrado **Xcode**, que é distribuído gratuitamente pela **Apple.** Utilizaremos como base as versões baseadas em **9.0.**

O **Xcode** está disponível para download gratuito no endereço **https://developer.apple.com/xcode/**

O **Xcode** é uma poderosa ferramenta de ambiente integrado, onde podemos desenvolver simultaneamente códigos e interface gráfica para aplicativos.

Dentro do Xcode podemos desenvolver projetos completos de aplicativos para diferentes dispositivos Apple, como Apple TV, Apple Watch, iPad, iPhone e Macs.

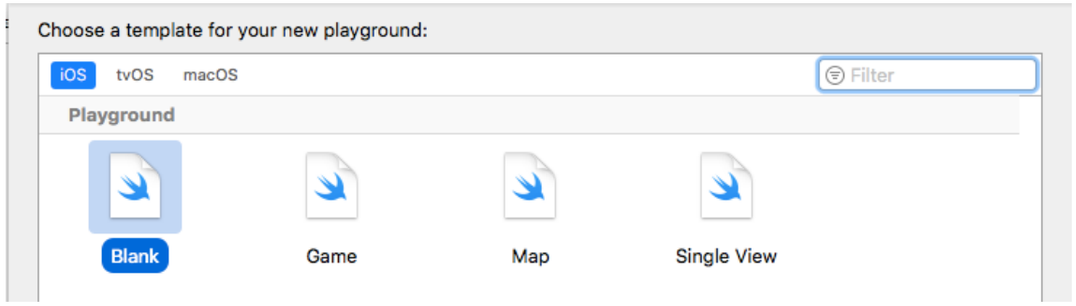
A IDE fornece opções de iniciarmos os projetos em formatos mais simples, chamados de Playground. Quando iniciamos um projeto dessa forma, temos uma interface mais simples e direta para testes com códigos e alguns recursos de apresentação.

Durante nossos primeiros passos utilizaremos projetos no estilo Playground para desenvolver os exemplos propostos. Para tal, basta iniciar o Xcode, representado pelo seguinte ícone:

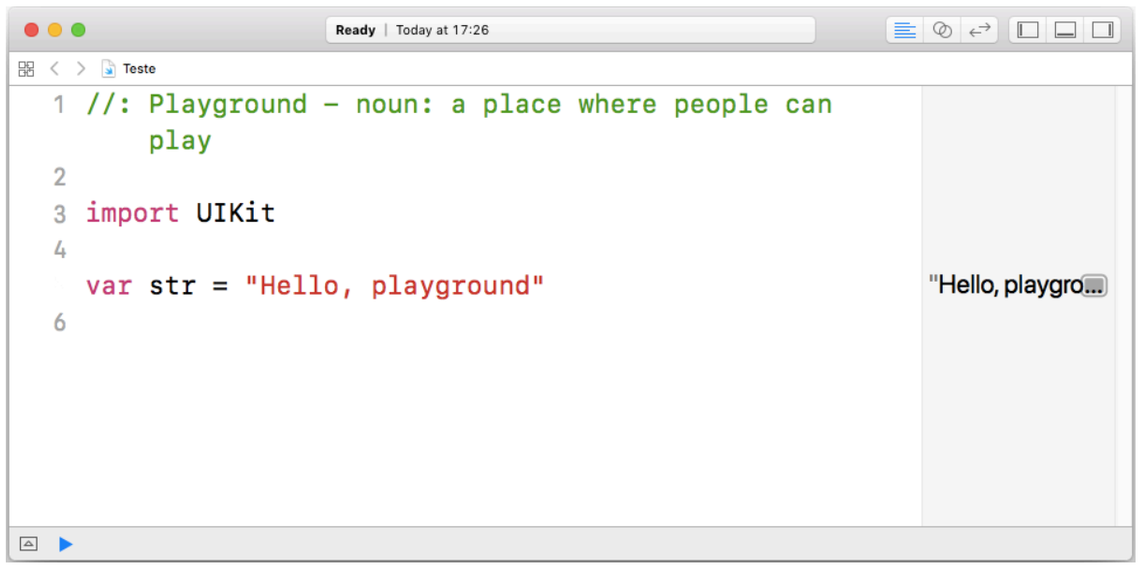
Na janela apresentada no início do Xcode, podemos escolher com qual tipos de projeto vamos iniciar um trabalho. Na mesma janela temos acesso a um histórico de projetos abertos, que podemos utilizar para continuar um trabalho.

Podemos escolher a opção **Get starded with a playground,** para começar a desenvolver um projeto que executa códigos em tempo real.

Em seguida vamos escolher a opção **Blank,** para iniciar um projeto em branco:



Por ultimo, escolhemos o local onde o projeto será salvo. A partir daí temos a disposição a tela para lançar códigos:



Seção 1-2

Atalhos e menus auxiliares

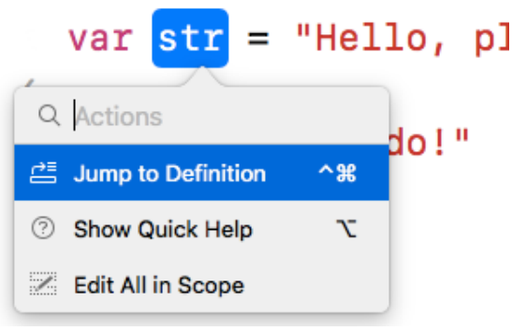
Assim como qualquer programa, o trabalho com o Xcode exige alguns atalhos para a prática cotidiana. Vamos trabalhar com alguns deles:

Aumentar ou diminuir o tamanho do código

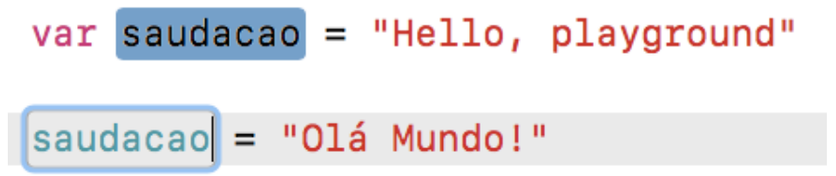
Podemos aumentar ou diminuir o tamanho do texto dos códigos utilizando os atalhos **command +** ou **command - :**

Editando itens com o mesmo nome

Quando clicamos sobre um elemento de código, com a tecla command pressionada, temos acesso a um menu de contexto, que pode variar de acordo com a região do código clicada:



Quando utilizamos a opção **Edit All in Scope,** podemos alterar todas as cópias dessa definição com um único recurso:



Salvando os projetos

Apesar do comando salvar (command+s) funcionar perfeitamente dentro do Xcode, salvando as ultimas definições feitas, ele é praticamente unútil dentro da IDE. Uma particularidade interessante dentro do Xcode, é que cada elemento digitado ou inserido no seu projeto é automaticamente salvo, sem a necessidade de demais comandos.

Outros atalhos

Ao longo dos módulos, veremos outros atalhos de trabalho, e outros elementos de menus auxiliares. É importante lembrar que atalhos melhoram a fluidez e a agilidade no trabalho dia a dia.

Capítulo 2

Code Style para Swift

Seção 2-1

Vamos “Codar” ?

Alguns editores de códigos de programação são estruturados de forma a auxiliar a digitação, dando ao desenvolvedor conforto no momento de programar.

A inserção de códigos em programação pode ser feita de forma livre e sem compromissos, o que evidentemente não é o recomendado, porém é importante adotar certos métodos de documentação para que a posterior manutenção dos códigos seja feita de forma mais rápida e eficaz.

Tabulações, espaços e quebras de linhas tem um poder duplo. Ao mesmo tempo que podem organizar seu código, também podem instaurar o caos dentro do seu editor.

Indentação

A organização do código é chamada de **Indentação.** Basicamente, um código saudável tem uma boa indentação, ou seja, um bom alinhamento das aberturas e fechamentos de chaves, parênteses, colchetes, etc.

Na Swift, as chaves `{ }` definem um bloco de execução de alguma tarefa. É interessante manter a abertura de um bloco junto ao comando que está sendo pedido:

```
if meuElemento {
}
```

Os colchetes `[ ]` definem elementos que contém coleções de dados. Podem ser utilizados envolvendo os dados com abertura e fechamento:

```
var numeros = [10, 20, 30]
```

O mesmo critério é válido quando falamos de parênteses `( )`. Podem ser utilizados envolvendo os parâmetros que serão definidos.

```
var numeros = (10, “Texto“, 30.0)
```

CamelCase

Quando codificamos, o espaço entre os caracteres abre um precedente para uma nova informação. Quando nomeamos elementos que devem ter um nome composto, não podemos utilizar espaços para separar as palavras.

A solução encontrada dentro da programação foi definir a separação de palavras por letras maiúsculas sem a adição de espaços, a esse sistema, dá-se o nome de **CamelCase:**

```
func nomearFuncoes (){
}
```

Dentro do mesmo conceito, ainda temos o **lowerCamelCase,** onde iniciamos a sentença com letra minúscula e o **UpperCamelCase,** inciado com letra maiúscula.

Quando nomeamos repositórios de dados, ou ações a serem executadas no nosso código, é uma boa prática utilizar o esquema **lowerCamelCase:**

```
var numerosPares = [2, 4, 6, 8, 10]
```

Quando criamos nossas próprias classes de objetos, o ideal é iniciarmos seus nomes com letras maiúsculas, e consequentemente adotar o esquema **UpperCamelCase:**

```
class PessoaJuridica {

}
```

A origem do termo **CamelCase** vem da semelhança do contorno das expressões, onde as letras em maiúsculo saltam no meio das minúsculas, como corcovas de um camelo.

Acentuação

Temos que ter em mente que nossa língua (Português) é estruturada de forma com que algumas palavras recebam acentuação para definição de silabas tônicas.

É de bom grado, que no momento da codificação, especificamente na nomenclatura dos elementos do código, deixemos de lado a acentuação.

A partir do momento em que vamos utilizar o texto em elementos que vão aparecer para o usuário final, podemos readotar a utilização de acentuação normalmente.

Objetividade e Clareza

Sempre que possível, seja objetivo e claro na nomenclatura de elementos no seu código. Utilizar nomes simples, compostos por caracteres únicos, não formam um bom código. Lembre-se que em um futuro, o trabalho possa carecer de revisões ou manutenções, e nomes que definam melhor os elementos farão a diferença.

Outras Limitações

Durante a nomenclatura, símbolos matemáticos e outros tipos de caracteres com pontos de interrogação ou exclamação, não são permitidos. Devemos fixar o nome dos elementos com caracteres simples.

Outro ponto importante a destacar, é que existem palavras que são reservadas ao sistema, nesse caso essas palavras também não podem ser utilizadas na nomenclatura.

Por exemplo: não podemos utilizar a palavra `var` para nomear um elemento, posto que a mesma já utilizada pelo sistema para designar a declaração de uma variável.

Capítulo 3

Introdução à linguagem Swift

Seção 3-1

Códigos Comentados

Vamos iniciar um novo Playground para incluir nossos primeiros códigos, que são os **Comentários**.

Comentários são trechos de códigos que serão ignorados pelo compilador ou interpretador da linguagem. Geralmente utilizamos os comentários para auxiliar na documentação e explanação de trechos do código.

É muito importante comentarmos nosso código, pois dessa maneira, a manutenção ou revisão do mesmo será mais fácil e simples de ser entendida.

Na linguagem Swift temos dois tipos de comentários: os de linha, válidos apenas para uma única linha, e os de blocos, que ignoram diversas linhas simultaneamente.

Veja abaixo como comentar uma linha:

```
// Esta linha será inteiramente ignorada
```

Quando surgir a necessidade de comentar mais de uma linha, podemos utilizar o comentário de um bloco:

```
/* Todo o Conteúdo
   desse bloco
   Será ignorado pelo compilador
*/
```

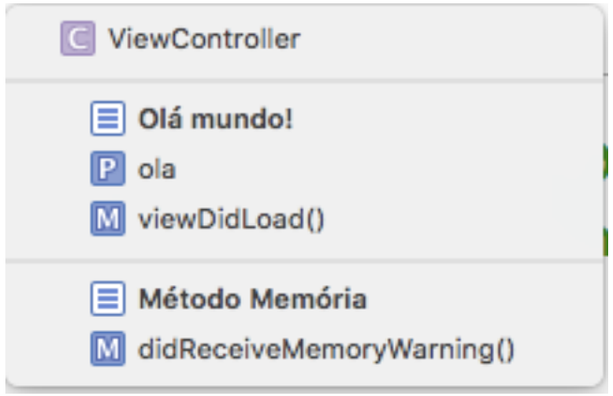
#Dica!

Fique atento com a utilização de comentários sem um bom método de organização. Caso os comentários sejam excessivamente desorganizados, ao invés de auxiliar e documentar, acabarão atrapalhando a compreensão do código.

Marcos de códigos

Quando comentamos, podemos determinar marcos no nosso código, que farão automaticamente uma separação. Para a criação de marcos, utilize o código **`//MARK:-`**:

```
//MARK:- Digite o nome do marco
```



Olá mundo - Função Print

Uma praxe no mundo da programação é fazer em seu primeiro código em uma linguagem as instruções que escrevam na tela **“Hello World!”.**

Podemos utilizar a prática para criarmos o nosso primeiro código, e consequentemente dar nossos primeiros passos na linguagem.

A função que imprime dados no console se chama **print ( ):**

```
//Primeiro passo na linguagem
print (“Olá Mundo!“)
```

```
Teste
1 //Primeiros passos
2
  print("Olá mundo!")
4
```

```
Olá mundo!
```

#Dica!

A função `print( )` pode ser utilizada sempre que surgir a necessidade de se imprimir dados em console. Não se esqueça que os dados impressos no console não são visualizados pelo usuário final da aplicação.

Seção 3-3

Variáveis e Constantes

As palavras-chave `var` e `let` são utilizadas para declarar repositórios de dados de forma variável ou constante.

Variáveis e **Constantes** são espaços em memória que podem reter dados em tempo de execução. Basicamente são "caixinhas" que podem armazenar informações para uso futuro em expressões, comparações lógicas, entre outras possibilidades.

Se você deseja que o valor não tenha alteração, utilize o `let`, caso contrário, se você estiver pensando em uma rotina que ao longo do processamento sofrerá alterações no valor de forma variável em memória, utilize o `var`.

Veja os exemplos:

```
//Utilize let para declarar o nome de uma empresa
let empresaDeCuppertino = "Apple"
print(empresa)
```

Quando definimos a constante `empresaDeCuppertino` como **Apple,** esse valor não será mudado ao longo do processamento. Mesmo que seja solicitado em código.

```
//Tente mudar o nome da empresa
empresa = "Apple Computers"
```

Quando falamos de variável, como por exemplo o nome de uma cidade em um cadastro, esse dado pode variar de acordo com a necessidade:

```
var cidade = "São Paulo"
print(cidade)

cidade = "Rio de Janeiro"
print (cidade)
```

Como o dado é variável, pode acontecer uma reatribuição de valor como vimos, onde a cidade variou de São Paulo para Rio de Janeiro.

Qualificadores

Variáveis e constantes são compostos por uma estrutura definida por qualificadores. Essa mesma estrutura sempre se repete.

Os qualificadores que definem essa estrutura são: **Escopo, nome, tipo e valor:**

```
escopo (var ou let) nome : tipo = valor

//Exemplos
var notaDoSemestre : Int = 10
var primeiroImperador : String = "Dom Pedro I"
```

Tipos de Dados

A Swift conta com os principais tipos de dados presentes em outras linguagens de programação. Abaixo uma tabela dos principais tipos utilizados.

Tipo de dado	Descrição	Exemplo
Character	Um único caractere	“Q“
String	Tipo texto	“Quaddro“
Int	Valores inteiros	25
Float	Números de pontos flutuantes (32 bits)	25.255
Double	Números de pontos flutuantes (64 bits)	25.2525255
Array	Coleção de valores ordenada de um mesmo tipo	[11,20,40,70]
Dictionary	Coleção que associa chaves e valores	[“nome”:”Zeus”, ”Idade:79”]
Set	Coleção de valores sem ordenação que não permite valores repetidos	[10,11,13,20]
Bool	Verdadeiro ou falso	true - false

Declaração de Repositório de Dados

A declaração do tipo de uma variável ou constante pode ser feita de forma implícita ou explícita. A linguagem de programação Swift aceita as duas maneiras para declarar um repositório. Veja no exemplo abaixo:

```
// Declaração implícita, onde omitimos o tipo
let anoDescobrimentoBrasil = 1500
print(anoDescobrimentoBrasil)

// Declaração explícita, onde declaramos o tipo
let diaDescobrimentoBrasil : Int = 22
print(diaDescobrimentoBrasil)

let mesDescobrimentoBrasil : String = “Abril”
print(mesDescobrimentoBrasil)
```

Como podemos observar nos exemplos acima, a constante `anoDescobrimentoBrasil` não teve o seu tipo definido, diferentemente da constante `diaDescobrimentoBrasil`, onde a mesma foi definida como um tipo `Int`.

Outra prática comum é a declaração de variáveis ou constantes vazias e inicializadas, que podem ter seus valores atribuídos em outros momentos do fluxo da aplicação:

```
var nome = String()
var idade = Int()
var altura = Double()
```

#Dica!

No desenvolvimento de sistemas, a manutenção poderá ser frequente e utilizar a declaração explícita é uma boa pratica de programação.

Strings e Charaters

String e **Character** são dois tipos muito utilizados na linguagem de programação Swift para gerenciar informações texto.

O tipo **Character** gerencia apenas um caractere e é utilizado para caracteres especiais e coleções de letras.

```
var letra: Character = “A”
print (letra)
```

O tipo **String** gerencia informações com mais de um caractere. Podemos utilizar para atribuir valores em nomes, endereços, telefones, e-mails, sites, entre outras formas de utilização:

```
var empresa: String = “Apple Computers”
print (empresa)
```

#Dica!

Observe que dentro de uma **String** podemos utilizar o espaçamento entre palavras e acentuação normalmente.

Textos mutilinhas

Podemos criar caixas de textos com múltiplas linhas dentro de um único objeto string. Uma particularidade nesse tipo de declaração é o uso de três aspas no inicio e no final da sintaxe:

```
var textoMultilinha = “””
Lorem ipsum dolor sit amet, consectetur adipiscing
    elit. Morbi et dui eget lorem vestibulum.

Vivamus vitae lobortis nunc.

Pellentesque ligula enim, volutpat sit amet nunc
   interdum, sollicitudin tristique justo.
“””
```

Interpolação de variáveis/constantes

Utilize a sequência de caracteres `\()` para interpolar variáveis e constantes dentro de um texto **(string)**. Interpolar é o mesmo que misturar variáveis ou constantes ao seu texto. Com essa prática, podemos alocar dentro de um texto, outro dado de um tipo diferente de String.

```
var nome = “José”
var idade = 30

print (“Nome: \(nome), Idade: \(idade)”)
```

Concatenação de variáveis/constantes

Podemos concatenar, ou seja somar dados de duas strings diferentes em um resultado:

```
var empresa1 = “Apple”
var empresa2 = “ – Computers“

print (empresa1 + empresa2)
```

Caracteres Unicode

Utilizamos `\u{ }` para criar caracteres **Unicode**. Unicode é um padrão que nos permite representar textos de qualquer forma de escrita existe. .

Exemplos:

```
let caracter1 = “caf\u{E9}”
let caracter2: Character = “\u{24}”
let caracter3 = “\u{2665}”

print(“palavra acentuada: \(caracter1)”)
print(“símbolo cifrão...: \(caracter2)”)
print(“coração.........: \(caracter3)”)
```

Comparando Strings e Caracteres

Podemos fazer comparações entre strings e caracteres para detectarmos igualdades e diferenças entre elas.

Usamos o operador **==** nos permite comparar duas strings, retornando um **Bool:**

```
print(empresa1 == empresa2)
```

Já os operadores **< (menor) e > (maior)** nos permite comparar os tamanhos das Strings:

```
print(empresa1 > empresa2)
```

Caracteres de Escape

Alguns caracteres que comumente utilizamos no dia a dia, como quebra de linhas ou tabulações, são utilizados pelo sistema com outras funções.

Em alguns momentos nos deparamos com a necessidade de utilizar esses caracteres. Por exemplo: sabemos que as aspas fazem parte da estrutura de uma string, porém pode surgir a necessidade de utilizarmos aspas em uma determinada frase. Para tal podemos utilizar um caractere de escape.

No caso das aspas temos a disposição o escape `\”`

Também temos um escape para quebra de linhas `\n`, e outro para espaçar tabulações: `\t`.

Capítulo 4

Operadores

Seção 4-1

Operador de atribuição

O sinal **=** é utilizado para atribuir valores a variáveis e constantes.

```
var numero = 100
```

No exemplo, atribuímos de forma implícita um valor inteiro de 100 a variável número.

Podemos utilizar o sinal de **=** para fazer reatribuições, ou seja, mudar o valor de variáveis:

```
var numero = 100
numero = 250
```

#Dica!

Constantes **(let)** não podem ter seus valores reatribuidos.

Podemos também, criar e atribuir valores a variáveis em cadeia:

```
var (numero1, numero2, numero3) = (100, 200, 300)

var (umTexto, umCaractere) = (String(), Character())
```

Seção 4-2

Operadores aritméticos

Os **operadores aritméticos** são aqueles utilizados para fazer operações matemáticas, tais como soma, subtração, multiplicação, divisão e resto da divisão:

```
let soma = 2 + 2
print (“Resultado da soma é \(soma)”)

let subtracao = 10 - 7
print (“Resultado da subtração é \(subtracao)”)

let multiplicacao = 100 * 5
print (“Resultado da multip. é \(multiplicacao)”)

let divisao = 200 / 4
print (“Resultado da divisão é \(divisao)”)

let resto = 33 % 5
print (“Resultado do resto é \(resto)”)
```

A Swift também permite transformar um valor em negativo, bastando acrescentar o sinal de menos ao valor, ou seja, sem a necessidade de multiplicar o resultado por -1:

```
var numero = 100
print (-numero)
```

Seção 4-3

Operadores compostos

Operadores compostos, são aqueles que utilizam o seu próprio valor, juntamente com algum operador aritmético para atribuir novos valores.

Podemos tomar como exemplo a situação na qual temos um valor **x** e desejamos atribuir a ele o seu próprio valor com o acréscimo de 10:

```
var valor = 100
valor = valor + 10
//O resultado será 110
```

Porém com o uso dos operadores compostos, podemos reduzir esta operação da seguinte forma:

```
valor += 10
//O resultado também será 110
```

Outros exemplos:

```
var preco = 300.00
//O preço é igual a preço mais 59.90
preco += 59.90

//O preço é igual a preço menos 10.10
preco -= 10.10

//O preço é igual a preço vezes 1.78
preco *= 1.78

//O preço é igual a preço dividido por 3
preco /= 3
```

Podemos também utilizar o operador de resto da divisão (remainder em inglês) que funciona da mesma forma:

```
var terResto = 7
terResto %= 2
```

Operadores de comparação

Operadores de comparação são utilizados para comparar valores. Essas comparações podem ser de igualdade, diferença, tamanho (maiores e menores) e de objetos idênticos. Observe a tabela abaixo:

Operador	Descrição
a == b	a é **igual** a b
a != b	a é **diferente** de b
a < b	a é **menor** que b
a > b	a é **maior** que b
a <= b	a é **menor ou igual** a b
a >= b	a é **maior ou igual** a b
a === b	os objetos são **idênticos**
a !== b	os objetos são **diferentes**

Como resposta a comparação, sempre teremos um objeto **bool** (verdadeiro ou falso)

Segue abaixo alguns exemplos com o uso de operadores de comparação:

```
var a = 1
var b = 3
```

```
a == b
//falso

a != b
//verdadeiro

a > b
//falso

a < b
//verdadeiro

a >= 1
//verdadeiro
```

```
a >= 2
//falso

a <= 1
//verdadeiro

a <= 3
//verdadeiro

a <= 0
//falso
```

#Dica!

Os operadores **=== (idênticos) e !== (diferentes)** não são utilizados para comparar valores. Devemos utilizar esses operadores para fazer a comparação da referência dos objetos.

Operador ternário

Utilizamos um **operador ternário** para associar valores a uma variável dependendo de uma determinada condição. Sua sintaxe é bastante simples:

```
condição ? acaoVerdadeira : acaoFalsa
```

Exemplo:

```
var nome = “Geraldo”
var resultado = nome == “Geraldo” ? “Seja bem
vindo” : “Desconhecido”
```

O resultado acima será **Seja bem vindo,** porque a condição logo após o sinal de interrogação foi validada como verdadeira. No exemplo abaixo o resultado será **Desconhecido,** porque a variável nome foi alterada para Roberto:

```
nome = “Antônio”
resultado = nome == “Geraldo” ? “Seja bem
vindo” : “Desconhecido”
```

Podemos utilizar os operadores ternários com qualquer tipo de variável, sejam dos tipos Inteiros, Float, String, entre outros:

```
var numero = 10
var parOuImpar = numero % 2 == 0 ? “Par” : “Impar”
//Retornará par
```

Capítulo 5

Tuplas, arrays e dictionaries

Seção 5-1

Tuplas

Tuplas são listas ordenadas de valores agrupados em uma variável ou constante. É um tipo simples de agrupamento de elementos que ajuda a poupar o desenvolvedor de ter que criar arrays para armazenar informações simples.

Os valores contidos em uma tupla podem ser de qualquer tipo, ou seja, as tuplas conseguem armazenar simultaneamente valores do tipo, Int, Float, String, etc.

Os valores de uma tupla podem ser acessados através de índices, que começam em 0 (zero) e formam uma ordem sequencial conforme a ordem de entrada de valores na tupla.

```
//Exemplo de declaração com valores de um mesmo
tipo:

var empresa = (“Quaddro”, “iOS”, “Swift”)

//Se decompormos essa tupla, ela ficaria da seguinte
maneira:

empresa (String, String, String)

índice 0 = “Quaddro”
índice 1 = “iOS”
índice 2 = “Swift”
```

```
//Declaração com valores de tipos diferentes:
var valores = (“Pedro Alvares Cabral”, 40, 87.3)

//Ao decompormos essa tupla, temos o seguinte
resultado:

valores (String, Int, Double)

índice 0 = “Pedro Alvares Cabral”
índice 1 = 40
índice 2 = 87.300000000000
```

Acessando valores de uma tupla

Para acessar os valores da tupla acima, basta indicar o nome da variável, seguida do seu índice. Não se esqueça que o índice do primeiro elemento do será sempre **0 (zero).**

```
print(empresa.0)
//Resultado: Quaddro

print(empresa.2)
//Resultado: Swift
```

Atribuindo nomes a elementos de uma tupla

Podemos declarar uma tupla, atribuindo nomes aos seus elementos e facilitando assim a leitura e organização de sua estrutura:

```
var ferrariEnzo = (cilindros: ”V12”, potencia: 660)
```

Para acessarmos o valor armazenado em um nome, utilizamos o nome da variável, acompanhado de um dot (.) mais o nome do elemento da tupla que deseja acessar:

```
print (ferrariEnzo.potencia)
```

Decompondo elementos de uma tupla

Também é possível decompormos os elementos de uma tupla:

```
var (x, y) = ferrariEnzo
//A variável x irá receber o valor de cilindros e
y o valor de potência.
```

Caso seja necessário ignorar algum valor no momento da decomposição de uma tupla, utilize o caractere _ **(underscore)** como no exemplo abaixo:

```
var (a, _) = ferrariEnzo
```

Arrays

Os **arrays** permitem que uma variável ou constante armazene mais de um valor em formato de lista para ser acessados e manipulados posteriormente através de um índice. Os valores de um array devem ser do mesmo tipo.

Podemos declarar arrays de várias formas, algumas explícitas outras implícitas. Também podemos iniciar um array vazio, e posteriormente adicionar os dados aos índices, ou declarar os dados que farão parte da coleção já no momento da sua criação.

```
//Sintaxes de declaração de forma explícita e vazio:
var arrayVazio = [String]()
var outroArrayVazio : [Int] = []

//Array com tipo explícito e com dados:
var arrayComTipoExplicito : [String] = [“valor1”,
“valor2”, “valor3”, “valor4”]

//Array com tipo implícita e com dados:
var arrayComTipagemImplicita = [“valor1”, “valor2”,
“valor3”, “valor4”]
```

Adição de valores a um Array

Podemos adicionar valores a um array, que serão acrescidos na sequência de espaços vazios nos índices. Para adicionarmos valores, podemos utilizar o recurso **.append,** ou ainda um operador de atribuição composto **+=,** conforme os exemplos:

```
var  arrayItens = [String]()
arrayItens.append(“PrimeiroValor”)
arrayItens.append(“SegundoValor”)
```

Ou ainda:

```
arrayItens += [“ValorAdicionadoSemAppend”]
```

Adição de item a um índice específico

Podemos adicionar um valor a um índice específico, vamos a sintaxe:

```
arrayItens.insert(“Adicionado em índice específico”,
at: 2)
```

Remoção de itens

Podemos remover um item de um índice específico conforme a necessidade, ou remover todos os itens de um array:

```
arrayItens.remove(at: 1) //Remoção de um item
arrayItens.removeAll() //Remoção de todos os itens
```

Substituição de itens

Por ultimo, vamos ver a substituição de itens em um índice específico:

```
arrayItens[0] = “valor substituído”
```

Acessando índices do array

Podemos acessar e utilizar objetos de um array através do seu índice. Essa é uma prática extremamente recorrente no dia a dia de programação.

```
var arrayFrutas = [“Maçã”, “Abacaxi”, “Laranja”]

print(arrayFrutas[1])
//Resultado: Abacaxi
```

#Dica!

Fique atento para não acessar índices inexistentes, pois esse acesso pode causar um erro devido a um valor nulo inesperado.

Acessando o primeiro ou ultimo elemento de um array

Um objeto Array disponibiliza propriedades para acessar os índices específicos dos elementos mais extremos. Podemos acessar o primeiro ou o ultimo elemento conforme os exemplos:

```
//Acessando o primeiro índice
print(arrayFrutas.first!)
//Resultado: Maçã

//Acessando o último índice
print(arrayFrutas.last!)
//Resultado: Laranja
```

Inspecionando um Array

Podemos verificar a quantidade de elementos existentes em um array, ou saber se o mesmo encontra-se vazio.

```
//Contando os elementos de um array
print(arrayFrutas.count)
//Resultado: 3

//Verificando se o array está vazio
print(arrayFrutas.isEmpty)
//Resultado: false
```

Ordenando elementos do array

Quando necessário, temos a disposição recursos para ordenar os elementos de um array. Podemos colocar uma determinada lista de elementos em ordem alfabética por exemplo, ou fazer a contagem de números em um array de forma progressiva.

```
//Ordenando arrays
print(arrayFrutas.sorted())
//Resultado: [“Abacaxi”, “Laranja”, “Maçã”]


var arrayNumeros = [3, 1, 3000, 200]
print(arrayNumeros.sorted())
//Resultado: [1, 3, 200, 3000]
```

Invertendo a ordem os elementos de um array

A ordem que os elementos são apresentados em um array pode ser invertida, conforme as necessidades de cada projetos.

```
//Invertendo os elementos de um array
var arrayReverso = Array(arrayFrutas.reversed())
print(arrayReversoFrutas)
//Resultado: [“Laranja”, “Abacaxi”, “Maçã”]
```

Enumerando os elementos de um array

Por ultimo, podemos enumerar elementos de um array conforme os seus respectivos índices:

```
//Enumerando os elementos de um array
var arrayEnumerado = Array(arrayFrutas.enumerated())
print(arrayEnumerado)
//Resultado: [(offset: 0, element: “Maçã”), (offset:
1, element: “Abacaxi”), (offset: 2,
element: “Laranja”)]
```

Dictionaries

Dictionaries assim como arrays podem organizar coleções de dados, porém a sua organização é feita através de um par de dados, o chamado par **key : value.**

O par **key : value** liga um valor a uma chave associativa, permitindo assim uma distinção mais completa dos elementos.

Assim como acontece com o array, podemos declarar dicionários de forma explícita ou implícita, já com valores ou vazia.

```
//Sintaxe de declaração de forma explícita e vazio
var dicionarioVazio = [String : Int]()
var outroDicionarioVazio : [String : Int] = [:]

//Dicionário com tipo explícito e com dados:
var dicionarioItens : [String : Int] =
["chaveA": 1, "chaveB": 2]

//Dicionário com tipo implícito e com dados:
var dicionarioItens = ["chaveA": 1, "chaveB": 2]
```

Adicionando ou trocando valores das chaves

Podemos incluir ou trocar o valor de uma chave utilizando uma simples atribuição:

```
dicionarioItens ["chaveA"] = 1
dicionarioItens ["chaveB"] = 2
dicionarioItens ["chaveC"] = 3
```

Removendo valores de uma chave

Quando for necessário remover qualquer objeto de um dictionary, utilizamos a função `removeValue.`

```
dicionarioItens.removeValue(forKey: "chaveA")
```

Podemos remover todos os itens de um dictionary utilizando a função `removeAll().`

```
dicionarioItens.removeAll()
```

#Dica!

Sempre faça uma avaliação para ver qual tipo de coleção se adéqua a necessidade do sou projeto. **Arrays ou Dicionários** podem ser utilizados de forma conjunta no processo, inclusive um dentro dos outros.

Acessando elementos de um Dictionary

Podemos acessar e utilizar objetos de um dictionary através das chaves. Essa é uma prática extremamente recorrente no dia a dia de programação.

```
var mensagensDeResposta = [ 200: “OK”, 403: “Access forbidden”, 404: “File not found”, 500: “Internal server error”]

print(mensagensDeResposta[403]!)
//Resposta: Access forbidden
```

Outros Recursos

Assim como os arrays, podemos utilizar os recursos .count, .first, .isEmpty, também com dictionaries.

```
//Contando elementos de um Dictionary
print(mensagensDeResposta.count)
//Resposta: 4

//Acessando o primeiro elemento de um Dictionary
print(mensagensDeResposta.first!)
//Resposta: (key: 500, value: “Internal server error”)

//Verificando se o Dictionary está vazio
print(mensagensDeResposta.isEmpty)
//Resposta: false
```

Capítulo 6

Condicionais

Seção 6-1

Estruturas condicionais

Durante a programação de aplicações, é relativamente comum chegarmos a determinado momento do código com a necessidade de mais de uma opção possível para uma ação.

Vamos imaginar um cenário de exemplo, onde dependendo do valor que o usuário digitar em um campo de cupom de desconto, devemos calcular X ou Y % de desconto para o produto em questão.

Todas as vezes que pensamos na execução de um aplicativo e "verbalizamos" as expressões "caso isso aconteça" ou então "se a resposta for verdadeira" estamos falando de condicionais.

Conceitualmente uma condicional é um bloco de código que nos permite determinar rumos diferentes para a execução de um programa. Normalmente o seu uso está ligado a perguntas e comparações relacionadas a valores iguais, maiores ou menores entre si, por isso os operadores mais comuns no em condicionais são os de **menor, maior, igual e diferente (<, >, ==, !=).**

A partir de agora nossos programas ganharão outro nível de recursos e complexidade pois é muito comum precisarmos usar condicionais em um aplicativo com interações do usuário.

Adiante serão abordados as condicionais **if/ else** com operadores **and, or,** e condicional **switch/case.**

#Dica!

Algumas vezes, dentro da logica com que pensamos nossa aplicação, acabamos utilizando varias condicionais em cadeia. É interessante em determinado momento, revisarmos nosso código na busca da simplificação e consequente supressão de condicionais desnecessárias.

Seção 6-2

Condicionais if / else

Apesar de simples, **if/else** é uma das estruturas mais importantes na hora de programar sistemas com entrada de dados pelo usuário.

Temos que entender que basicamente essa estrutura condicional fará uma pergunta a aplicação **(if),** tendo como resposta um valor booleano, que pode ser **verdadeiro** ou **falso**. Se a resposta for verdadeira, a aplicação seguirá o caminho indicado por **if,** caso contrario, optará pelas condições estabelecidas por **else.**

Podemos trabalhar apenas com a expressão verdadeira, desprezando-se a falsa. Nesse caso podemos utilizar apenas o **if,** sem a necessidade do **else.**

Vamos a sintaxe de if/else:

```
if expressaoLogica {
// Caso a expressão seja verdadeira este bloco
será executado

} else {
// Caso a opção seja falsa este bloco será executado
}
```

Inicialmente vamos trabalhar apenas com if, sem a aplicação de else, vamos aos exemplos:

```
var valor = 100

if valor == 100 {
   print(“O valor é igual a 100 “)
}

if valor > 2 {
   print(“O valor é maior que 2”)
}

if valor != 200 {
   print(“O valor é diferente de 200”)
}
```

Quando implementamos o **else**, sempre haverá a execução de um dos blocos, ou seja, um dos dois caminhos será seguido pela aplicação:

```
var gol = true

if gol == true {

   print(“Pode comemorar!”)

} else {

   print(“Continue torcendo...”)
}
```

Podemos também utilizar **else if** para verificarmos varias condições em uma mesma estrutura, veja o exemplo:

```
var numero = 20

if numero < 20 {

    print(“Menor que 20”)

} else if numero > 20 {

    print(“Maior que 20”)

} else {

    print(“O número é 20”)
}
```

Seção 6-3

Operadores Lógicos

Operadores AND e OR

A função desses operadores é criar condições compostas, onde várias condições serão necessárias para decidir o rumo do código.

Para codificarmos esses operadores utilizamos os caracteres **&&** para **AND, ||** para **OR.** Vamos às aplicações:

```
var cupom: Int = 1020
var hora: Int = 11

if cupom == 1010 && hora < 12{
   print("Desconto concedido \n")
}else{
   print("Não foi possível conceder o desconto :(")
}
```

Nesse código temos uma expressão com duas perguntas: se cupom é **igual a 1010** e se hora é **menor que 12,** o desconto será concedido, caso uma da das duas expressões não seja atendida, o desconto será negado.

#Dica!

Quando utilizamos **AND (&&),** existe a obrigatoriedade das duas (ou mais) expressões serem confirmadas.

Podemos testar as mesmas condições, mas agora vamos utilizar o operador **OR (||):**

```
if cupom == 1010 || hora < 12{
   print("Desconto concedido \n")
}else{
   print("Não foi possível conceder o desconto")
}
```

Nesse caso, se apenas uma das expressões for verdadeira, serão apresentadas as opções dadas no bloco if, ou seja, o desconto será concedido.

Operador NOT

Podemos utilizar o operador NOT para negativar uma condição. No exemplo adiante, vamos verificar se um array NÃO está vazio:

```
var arrayDias = ["Seg", "Qua", "Sex"]

if !(arrayDias.isEmpty){

    print("Dias disponíveis: \(arrayDias)")
}

//Resultado: Dias disponíveis: ["Seg", "Qua", "Sex"]
```

Seção 6-4

Condicional switch / case

Outro recurso para implementar condicionais é a estrutura **Switch/ Case**. A diferença em relação ao if/else é a facilidade em fazer comparações sucessivas caso a caso. Se por exemplo, tivéssemos que comparar um valor com 3 possíveis respostas, a abordagem de **switch** seria mais indicada.

Vamos a sintaxe:

```
switch valorASerConsiderado {
  case valorCaso1:
  //Bloco a ser considerado se valorCaso1
  seja verdadeiro

  case valorCaso2:
  //Bloco a ser considerado se valorCaso2
  seja verdadeiro

  case valorCaso3:
  //Bloco a ser considerado se valorCaso2
  seja verdadeiro

  default:
  //Bloco a ser considerado caso nenhum dos cases
  sejam verdadeiros
}
```

Sua estrutura permite a chamada comparação encadeada de maneira mais organizada, onde várias hipóteses podem ser verificadas sucessivamente. A seguir encontramos a estrutura descrita acima na forma de um código de exemplo para comparação de valores:

```
var valor: Int = 100

switch valor {
  case 1:
  print(“Não temos muito dinheiro”)

  case 100:
     print(“Estamos com algum dinheiro”)

  case 1000:
     print(“Estamos bem de dinheiro”)

  default:
  print(“Temos \(valor)”)
}
```

Uma das grandes evoluções do condicional Switch/Case na Swift é o fato de podermos comparar expressões no formato de tuplas, assim podemos deixar nossas condições mais complexas, eliminando a necessidade de se usar if/else em muitos casos:

```
var nomeUsuário = “Quaddro”
var senha = 1234

switch(nomeUsuário == “Quaddro”, senha == 1234){
    case (true, true):
        print(“Acesso permitido”)

    case(false, true):
        print(“Nome do usuário incorreto”)

    case(true, false):
        print(“Senha incorreta”)

    default:
        print(“Nome do Usuário e Senha incorretos”)
}

//Resultado: Acesso permitido
```

Capítulo 7

Controle de Fluxo

Seção 7-1

Estruturas de repetição

Estruturas de repetição são utilizadas para executar repetidas vezes determinados blocos ou sequências de código.

Atrelado a uma estrutura de repetição, normalmente está associada também uma expressão lógica que define a **condição de parada,** que determina que o laço deve parar de ser executado ou continuar sua execução.

Em uma estrutura de repetição é importante atentar a condição de parada, pois caso a mesma por algum motivo não tenha a possibilidade de se tornar verdadeira, o processo pode entrar em **loop infinito** e gerar inconsistências na aplicação.

Dentro das estruturas de repetição temos o **for-in, while e repeat while** que serão apresentados nos tópicos a diante.

For - in

Utilizamos o **for-in** quando a rotina de programação precisar ler ou armazenar dados dos objetos de uma coleção, analisar itens dentro de uma sequência, criar um intervalo ou uma progressão.

É a estrutura de repetição mais utilizada para chamadas de repetições controladas, onde se sabe a quantidade de vezes que o processo será executado. A seguir temos a sintaxe:

```
for objetoDeLeitura in intervaloOuColeção{
   // Bloco de execução a ser repetido para todos os
   itens da sequência
}
```

Vamos a um exemplo onde podemos especificar o número de dias em uma semana:

```
for semana in 1...7 {

   print(semana)
}
```

Uma outra forma que é conhecida de **half-open range,** onde o laço terá a repetição do valor máximo **menos 1 (um):**

```
for meses in 1..<13 {

   print(meses)
}
```

Agora vamos utilizar o for-in para fazer a leitura de uma coleção do tipo array:

```
let diasDaSemana = [“Seg”,”Ter”,”Qua”,”Qui”,”Sex”,
”Sab”,”Dom”]

for mostrarDia in diasDaSemana {

   print(mostrarDia)
}
```

Nesse caso, o for já tem a consciência que fará o loop por sete vezes, dada a quantidade de elementos existentes dentro do array.

Seção 7-3

While e repeat while

O **While** assim como o **for** é uma estrutura de repetição. É responsável por executar um bloco de instruções enquanto a sua condição for verdadeira. Sua sintaxe é:

```
while condição {
   //Bloco de execução
   //Tratamento de parada da execução
}
```

No exemplo abaixo faremos um contador que fará uma contagem regressiva de 100 para 1.

```
var contador = 100

while contador > 0 {
  print(contador)

  //tratamento de parada
  contador -= 1
}
```

#Dica!

No **While**, se não colocarmos o tratamento de parada da execução, o processo entrará em **loop infinito.** Esse processo pode gerar inconsistências na sua aplicação.

Repeat While

Com uma sintaxe e comportamento bastante similar ao while, o **repeat while** também irá executar um bloco de código.

A diferença é que primeiro será executado o tratamento de parada, para somente depois verificar a condição, ao contrário do que acontece com o while, que primeiro verifica a condição para depois procurar qual é o tratamento de parada.

```
var andarAtual = 0
var numeroDeAndares = 5

let andarConstruido = true

repeat {

   if andarConstruido {
      print(“Foram construídos \(andarAtual)
      andares”)

      andarAtual += 1
   }

} while (andarAtual <= numeroDeAndares)
```

Capítulo 8

Funções

Seção 8-1

Funções

Durante o desenvolvimento de uma aplicação, muitas vezes temos que executar uma mesma funcionalidade diversas vezes. Na Swift, como na maioria das linguagens de programação, esse problema é facilmente solucionado através da criação de **funções.**

Funções são rotinas que executam um determinado conjunto de instruções e podem ser chamadas sempre que necessário no código, evitando que existam repetições desnecessárias.

Na Swift, utilizamos o nome reservado `func` para declararmos uma nova função, como podemos ver no exemplo da sintaxe abaixo.

```
func nomeDaFuncao (nomeExternoDoParametro
nomeInternoDoParametro : tipoDeParâmetro) ->
tipoDeRetorno{

   //Bloco de execução da função

   return retornoDeAcordoComTipoDado
}
```

É uma praxe que o nome de uma função contenha um verbo, dado o fato das funções serem as ações de uma aplicação

Funções sem parâmetro e sem retorno

Como exemplo, vamos criar uma função para imprimir uma informação em console:

```
func imprimirPrimeiraFuncao() {
   print(“Primeira Função com a Swift”)
}

//Para chamar a função criada
imprimirPrimeiraFuncao()
```

Observe que, como não pedimos o tipo de retorno, o mesmo não foi solicitado pelo sistema, e consequentemente não foi utilizado. A partir do momento que estipularmos o tipo de retorno, o sistema automaticamente exigirá a sua reposta.

Podemos ter funções com ou sem parâmetros, e com ou sem retorno.

Funções com Parâmetros

Parâmetros são elementos declarados na assinatura da função que podem ser utilizados para trazer informações para dentro do escopo da função.

Na declaração de um parâmetro, podemos escolher para ele um nome externo, que será solicitado no momento que a função for instanciada, e um nome interno, que será utilizado apenas dentro do escopo da função.

```
func somarDoisNumeros (valorA umValorA : Int,
valorB umValorB : Int){

    print("Primeiro Valor Informado: \(umValorA)")
    print("Segundo Valor Informado: \(umValorB)")
    print("Somatória dos Valores:
         \(umValorA + umValorB)")
}

//Instância da Função
somarDoisNumeros(valorA: 20, valorB: 7)
```

Podemos ter apenas um único nome para os parâmetros. Nesse caso, o mesmo nome será utilizado tanto dentro do escopo, como na instância da função:

```
func exibirNome (nome: String, sobrenome: String){

    print("Nome completo: \(nome) \(sobrenome)")
}

exibirNome(nome: "Leonardo", sobrenome: "DaVinci")
```

Por ultimo, caso seja necessário, podemos ter um nome de parâmetro que atuará apenas dentro do escopo da função, não sendo exibido no momento da instância:

```
func calcularMetrosQuadrados (_ largura : Float,
  _ comprimento : Float){

    print("Quantidade de metros Quadrados:
    \(largura * comprimento)m²")
}

calcularMetrosQuadrados(5.0, 10.0)
```

Funções com Retorno

As funções podem ter algum retorno, de um tipo pré definido em sua declaração. O retorno nada mias é do que um dado que a função pode externar. Esse dado pode ser utilizado para definições de outros elementos, vamos a um exemplo:

```
func saudar(pessoa : String) -> String {
    let saudacao = "Olá, " + pessoa + "!"
    return saudacao
}
let nome = saudar(pessoa: "Tarsila")
print(nome)
```

Capítulo 9

Operadores customizados

Seção 9-1

Operadores customizados

Todas as linguagens de programação possuem bibliotecas com vários operadores para facilitar a vida do programador na hora de realizar cálculos ou pequenas tarefas.

Operações como soma, subtração, divisão e multiplicação, entre outras, são utilizadas no dia a dia de desenvolvimento com muita frequência.

A linguagem de programação Swift apresenta um novo modelo para o desenvolvimento de operadores especiais de acordo com a necessidade de desenvolvimento de um aplicativo. Através dele, podemos criar nossos próprios operadores.

Para a criação de **operadores customizados,** a Swift fornece três palavras chaves que são:

- `prefix`
- `infix`
- `postfix`

Para criarmos nosso próprio operador, devemos primeiramente declará-lo utilizando uma das palavras-chave acima citadas e logo em seguida implementar o operador utilizando uma função.

Nas próximas seções abordaremos como cada um destes operadores funcionam. Faremos vários exemplos práticos de implementação para vermos como customizar os operadores de nossas aplicações.

#Dica!

A Swift permite a utilização indiscriminada de caracteres para a criação de operadores, incluindo **+, - , *** e **/,** ou seja, devemos evitar a utilização desses caracteres sozinhos para não descaracterizarmos suas funções.

Utilizar uma combinação de caracteres como ******** por exemplo, é perfeitamente possível.

Um dos caracteres que a linguagem não permite customizar, é o caractere de atribuição, sinal de igual **(=).**

Prefix

Os operadores **prefix** atuam de foma similar ao operador de pré incremento, ou seja, primeiramente será executado a operação para depois associar o valor. Vamos a sintaxe:

```
prefix operator operadorCustomizado

prefix func operadorCustomizado (propriedade : Tipo)
-> tipoDoRetorno{

    return retorno
}
```

No exemplo a diante, criamos o operador **"<<"** que irá negativar um valor quando aplicado:

```
//Criando o operador customizado para negativação
prefix operator <<

prefix func << (numero : Int) -> Int{

    return -numero
}
//Testando o operador
print(<<(10 + 10))
//Resultado: -20
```

#Dica!

No caso da negativação, se o valor já fosse negativo, entraria a ação da regra de sinais, ou seja, se fosse -100, o valor da impressão em console seria positiva.

Postfix

O operador **postfix** atua de foma similar ao operador de pós incremento, ou seja, o valor será associado após a execução da operação:

```
postfix operator operadorCustomizado

postfix func operadorCustomizado (propriedade :
Tipo) -> tipoDoRetorno{

      return retorno
}
```

Como exemplo vamos criar um operador customizado que efetuará a elevação de um número ao cubo **(n3):**

```
//Criando o operador customizado para negativação
postfix operator ***

postfix func *** (numero : Int) -> Int{

      return numero * numero * numero
}
```

```
//Variaveis
var valor1 = 5

//Aplicando o operador customizado
print (valor1***)
//Resposta: 125
```

#Dica!

Como os próprios nomes sugerem, operadores customizados, devem ser posicionados a frente do valor, no caso de **prefix** ou após o valor, em **postfix.**

Infix

O operador **infix** opera entre os elementos, ou seja, ele não está atrelado ao um momento específico para a sua atuação. Devemos posicionar o operador logo após o objeto a ser operado.

```
infix operator operadorCustomizado

func operadorCustomizado (propriedade : Tipo)
-> tipoDoRetorno{

    return retorno
}
```

No exemplo vamos executar uma tarefa que replica um conjunto de caracteres:

```
// Operador que replicará caracteres
infix operator <->
func <-> (caract: String, vezes: Int) -> String {

    var linhaFinal = ""
    for _ in 1...vezes {
        linhaFinal = "\(linhaFinal)\(caract)"
    }
    return linhaFinal
}
```

```
//Aplicando o operador customizado
print("-!" <-> 20)
//Resultado: -!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!
```

Capítulo 10

Struct e Enum

Seção 10-1

Struct

Em Swift, classes e **estruturas** são extremamente semelhantes e possuem sintaxes bem parecidas entre si. A diferença fica por conta da utilização da palavra reservada `struct` ao invés de `class` na sua declaração. As diferenças entre as duas na prática, é que as `structs` são normalmente utilizadas para operações mais simples, que não demandam desalocação de informações em memória.

Structs não possuem características como herança, por exemplo. Suas instâncias são sempre passadas por valor, diferentemente de classes, onde são passadas por referências. Vamos a sintaxe:

```
struct NomeDaEstrutura {
   // Definição da estrutura com propriedades e
   métodos
}
```

Vamos a um exemplo de aplicação:

```
struct Dimensoes {

   var largura = Float()
   var comprimento = Float()
}
```

Podemos instanciar a struc e utilizar suas propriedades, assim como ocorre com as classes:

```
var cozinha = Dimensoes()

cozinha.largura = 4.35
cozinha.comprimento = 3.70
```

#Dica!

Podemos combinar a utilização de estruturas e classes, inclusive com instancias de um dentro do outro.

Enum

A palavra reservada `enum` é utilizada quando pensamos em **um número finito de possibilidades.** Estas possibilidades podem ser padronizadas se forem do mesmo tipo e seu conteúdo não for modificado, adicionado ou retirado.

A **tipagem** da enumeração é obrigatória quando houver inserção e recuperação de valores, mas pode ser omitida caso a necessidade seja apenas utilizar um caso como referência.

Dentro de cada **case,** teremos um dos elementos do nossa enumeração. Veja a sintaxe:

```
enum NomeDaEnumeracao : TipoSeForUtilizar {

    case nomeDoElemento = valorDoElemento
    case nomeDoElemento = valorDoElemento
    case nomeDoElemento = valorDoElemento
}
```

Como exemplo, vamos trabalhar com os quatro pontos cardeais, onde Norte será nosso índice 0:

```
enum PontosCardeais {
    case Norte
    case Sul
    case Leste
    case Oeste
}

print(PontosCardeais.Sul)
//Resultado impresso em console: Sul
```

Utilizando o número de posição

Podemos utilizar o número de posição da informação, basta utilizar o comando `hashValue` em complemento ao ponto:

```
print(PontosCardeais.Sul.hashValue)
//Resultado impresso em console: 1
```

Utilizando os valores

Para a utilização de valores dentro de cada caso, a tipagem da enumeração deve ser **explícita**, e devemos utilizar o comando `rawValue` em complemento ao ponto:

```
enum PontosCardeais : String {
   case Norte = “Vamos na direção Norte“
   case Sul = “Vamos na direção Sul“
   case Leste = “Vamos na direção Leste“
   case Oeste = “Vamos na direção Oeste“
}

print(PontosCardeais.Sul.rawValue)
//Resultado impresso em console: Vamos na
direção Sul
```

Atribuição de casos em cadeia

Podemos atribuir casos em cadeia, utilizando uma única linha, separando os itens por vírgula:

```
enum PontosCardeais {
   case Norte, Sul, Leste, Oeste
}
```

Atribuição de valores automáticos por tipagem

Podemos utilizar o tipo da enumeração para determinar de forma automática o valor a ser exibido. Se utilizarmos um tipo `String`, os valores adotados serão idênticos ao nome do caso:

```
enum PontosCardeais : String {
   case Norte, Sul, Leste, Oeste
}

print(PontosCardeais.Sul.rawValue)
//Resultado impresso em console: “Sul”
```

Quando trabalhamos com tipo `Int`, podemos determinar um valor para apenas um dos elementos, sendo que os demais atribuirão valores sequenciais ao indicado:

```
enum Semana : Int{
   case Domingo = 100
   case Segunda
   case Terça
   case Quarta
   case Quinta
   case Sexta
   case Sábado
}

print(Semana.Sexta.rawValue)
//Resultado impresso em console: 105
```

Capítulo 11

Optional chaining

Seção 11-1

Optional chaining

Optional chaining é o processo de consulta a propriedades, métodos e subscripts que podem ser ou não **nulos.**

Se o opcional contém um valor, a leitura o elemento é bem-sucedida. Se o opcional é nulo, a leitura do elemento retornará `nil` (abraviação para **not in list**).

Vamos a um exemplo. Imagine que durante um período de provas, seu professor de ciências esqueceu de transferir as notas de todos os alunos para o sistema de avaliação. Teríamos algo parecido com isso:

```
var matematica: Double? = 9.5
var portugues: Double? = 8.5
var ciencias: Double? = nil
var geografia: Double? = 10.0
```

Note que o valor que não foi lançado está sendo considerado como nulo, ou seja, `nil`.

Durante o fluxo da aplicação, quando o sistema se deparar com uma situação nula, poderão ocorrer dois tipos de sequenciamentos. Os sequenciamentos poderão ser gerenciados com os caracteres **?** (ponto de interrogação) e com o **!** (ponto de exclamação) sendo que:

? - Admite qualquer valor, inclusive nulo;

! - "Desembrulha" o objeto de maneira indiscriminada, esperando a garantia do programador.

Podemos relacionar diversas sequências (encadeamentos) de tipos. Com o optional chaining poderemos gerenciar de forma mais segura o retorno de valores das propriedades e métodos.

As próximas seções abordarão como podemos utilizar essas opções e como elas se comportam.

Optional - ?

Quando queremos que um determinado elemento permita o retorno de um valor nulo, utilizamos o **optional**, representado pelo caractere **?** ao lado tipo. Dentro de uma classe, é recomendado o uso de **?** ao declararmos propriedades, como no exemplo abaixo:

```
struct Treinamento {
   var titulo: String?
   var duracao: Int?
}
```

```
var swiftBootcamp = Treinamento(titulo: “iOS/Swift
Bootcamp”, duracao: nil)

print(swiftBootcamp.titulo)
//Resposta: Optional(“iOS/Swift Bootcamp”)
```

Perceba que o valor da propriedade **duracao,** do objeto **swiftBootcamp,** foi atribuído como **nil** graças ao **optional (?)** declarado na criação da propriedade, dentro da classe.

Porém, quando pedimos para imprimir o título, o resultado é um valor **Optioal().** Na seção a seguir veremos como desembrulhar esse valor sem essa situação.

#Dica!

Caso não queira gerenciar valores nulos, inicialize as propriedades com o método **init.**

Seção 11-3

Forced Unwrapping

O **forced unwrapping (!)** força o desempacotamento das informações de uma aplicação. Quando utilizamos esse recurso, estamos garantindo ao sistema que teremos todos os valores solicitados e nenhum deles será nulo.

Vamos voltar ao exemplo anterior:

```
class Treinamento {
   var titulo: String?
   var duracao: Int?
}

   var swiftBootcamp = Treinamento(titulo: “iOS/Swift
   Bootcamp”, duracao: nil)
```

Se fizermos um print do **título** de **swiftBootcamp**, teremos um valor retornado opcional:

```
print(swiftBootcamp.titulo)
//Resposta: Optional(“iOS/Swift Bootcamp”)
```

Agora, se fizermos um print da **duração** de **swiftBootcamp**, teremos um valor retornado **nil**:

```
print(swiftBootcamp.duracao)
//Resposta: nil
```

Se forçarmos o desempacotamento dessa informação nula, simplesmente acontecerá um erro fatal na nossa aplicação **(crash),** pois prometemos algo que simplesmente não existia.

```
print(swiftBootcamp.duracao!)
//Resposta impressa em console: Fatal Error
```

Quando utilizamos o forced em um valor que não seja nil, desembrulhamos o objeto fora do optional, veja o exemplo:

```
print(swiftBootcamp.titulo!)
//Resposta: iOS/Swift Bootcamp
```

Geralmente utilizamos o forced unwrapping para dar fluidez no desenvolvimento da aplicação, em momentos que o sistema solicita informações que ainda não existem mas o desenvolvedor tem a certeza que em algum momento entrarão no fluxo.

Optional Binding

O **Optional Binding** fornece uma maneira alternativa para testar e desembrulhar um opcional em uma condição `if`**.** Com esse método podemos desembrulhar informações sem o uso do **Forced Unwraping,** e sem correr o risco de quebrar a aplicação.

Vamos a sintaxe:

```
if let nomeDaConstante = algumaOpicional {
    //Declarações realizadas se o valor for válido
}else{
   //Declarações realizadas se o valor for nulo
}
```

Basicamente será feita uma verificação entro do `if/else`, a vantagem em desembrulhar com esse método é a garantia de não ocorrer um erro fatal (crash): Os exemplos a seguir mostram o resultado:

```
var nome: String? = “José”

if let meuNome = nome{
   print(“Meu nome é \(meuNome)”)
}else{
   print(“Valor nulo”)
}
//Resultado: Meu nome é José
```

```
var idade: Int? = nil

if let minhaIdade = idade{
   print(“Minha idade é \(minhaIdade)”)
}else{
   print(“Valor nulo”)
}
//Resultado: Valor nulo
```

Observe que estamos usando uma constante que, em conjunto com o condicional `if`**,** verifica se o valor é nulo e já desembrulha o opcional para utilizarmos dentro do bloco.

Condicional de falha - guard

A partir da versão 2.0, foi adicionado o condicional `guard` que também podemos chamar de **condicional de falha,** pois ele necessariamente deve sair do bloco que está sendo executado caso não consiga validar a condição.

Este elemento é muito utilizado para tratarmos optionals sem termos que aninhar muitos `if/else`.

No exemplo adiante, teremos uma função que recebe um valor opcional inteiro como parâmetro. Usaremos o `guard` para verificar se o valor recebido é um nulo:

```swift
func areaQuadrado(valor: Int?){

    guard let numero = valor else {

      print(“Nenhum valor foi passado”)

      return

    }

    print(numero * numero)
}

areaQuadrado(valor: 10)
areaQuadrado(valor: nil)

//Resultado impresso em console conterá apenas a
confirmação do valor nulo.
```

Depois do bloco, que só ira acontecer caso o desempacotamento falhe, podemos trabalhar normalmente com a constante valor, que já recebe o parâmetro desempacotado.

Se substituíssemos `guard` por `if`, mantendo esta estrutura, teríamos que escrever o restante do código dentro do bloco da condição.

Se tivermos mais valores para tratar, teríamos que aninhar vários comandos `if`, deixando o nosso código mais complexo e consequentemente menos legível.

#Dica!

O comando `guard` deve necessariamente interromper a execução de um determinado bloco de código com um `return`.

Seção 11-5

Nil Coalescing Operator

O **operador Nil Coalescing** ou em Português operador de coalescência nula, é responsável por retornar o operando esquerdo caso ele não seja nulo, senão o operador irá retornar o operando direto. No exemplo abaixo iremos observar a sua utilização:

```
var nome1: String? = nil
var nome2: String? = “Dante Alighieri”
var resultado = nome1 ?? nome2
print(“Resultado: \(resultado!)”)
//Impressão em console = Resultado: Dante Alighieri
```

#Dica!

Não esqueça de indicar que possíveis valores nulos como **Optional (?),** caso contrario teremos um **erro de compilação.**

Neste exemplo podemos observar que o resultado retornado será **Dante Alighieri,** porém caso a variável `nome1` não seja nula, o valor retornado será o próprio. Veja o exemplo abaixo:

```
var nome1: String? = “William Shakespeare”
var nome2: String? = “Dante Alighieri”
var resultado = nome1 ?? nome2
print(“Resultado: \(resultado!)”)
//Impressão em console = Resultado:
William Shakespeare
```

Podemos dizer que o **Nil Coalescing** é uma espécie de operador ternário para os tipos opcionais, que fará a comparação dos valores, e nunca deixará o valor nulo aparecer.

Capítulo 12

Classes, objetos, propriedades e métodos

Seção 12-1

Classes - criando objetos

A **Programação Orientada a Objetos** (ou **POO**), foi uma grande mudança de paradigma no desenvolvimento de aplicações, trazendo estruturas semelhantes as do mundo real para dentro da programação.

Utilizando este paradigma, podemos construir aplicações robustas e mais eficientes, gerando menos esforço para manutenção e mais reutilização de códigos.

O primeiro e um dos mais importantes conceitos a serem entendidos é o conceito de **Classes**, que podem ser entendidas como **abstrações de objetos** que possuem características semelhantes.

Classes podem herdar métodos, propriedades e outras características de outra classe, este conceito é conhecido como **herança.** Vamos ver esse assunto mais adiante.

A sintaxe para criação de uma classe é a seguinte:

```
class NomeDaClasse : SuperClasseHerdada{

    // Definição da classe com propriedades e métodos
}
```

Quando instanciamos uma classe, estamos criando um **objeto**. Um **objeto** é capaz de armazenar estados através de suas **propriedades** e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos. Vamos ao exemplo:

```
//Criando a classe
class Cor {
}

//Utilizando a classe para definir tipo do objeto
let amarelo : Cor()
let vermelho : Cor()
let azul : Cor()
```

#Dica!

Para a nomenclatura de **classes**, é aconselhável utilizar o estilo de código **UpperCamelCase.**

Seção 12-2

Propriedades e métodos

Propriedades, também chamadas de **atributos,** são as características de um objeto definidas dentro do escopo de uma classe. Podemos dizer que as propriedades de uma classe definem o que ela representa.

Criar propriedades dentro de uma classe é extremamente semelhante ao processo criar uma variável ou constante, bastando apenas escrever as palavras reservadas `var` ou `let` dentro do contexto `class`**.**

Se propriedades definem as características de uma classe, **métodos** definem o que uma classe faz. Para criarmos **métodos,** utilizamos a mesma sintaxe de criação de uma função, usando a palavra reservada `func` dentro do contexto de `class`.

Veja o exemplo a seguir:

```
//Criando a classe animal
class Animal {

   //Declaração de propriedades
   var especie : String = “”
   var raca : String = “”
   var peso: Double = 0.0
   var altura: Double = 0.0
   var emiteSom: Bool = false
   var idadeMeses: Int = 0

   // Declaração dos métodos
   func andar(){
      print(“Animal andando”)
   }

   func comer(){
      print(“Animal comendo”)
   }
} //Fechamento do escopo da classe

//Instanciando o objeto gato como Animal
var gato = Animal()

//Utilizando as propriedades do animal
gato.peso = 15.2
gato.emiteSon = true
gato.idade = 24

//Utilizando o método
gato.andar()
```

#Dica!

Para utilizarmos a propriedade de uma classe e atribuirmos um valor ao objeto, utilizamos o **ponto (.)** em uma sequência conhecida como **dot syntax.**

Seção 12-3

Métodos inicializadores

Podemos utilizar **métodos inicializadores** para serem executados no momento que instanciamos nossa classe. Através deles, podemos atribuir valores iniciais às propriedades, executar métodos da classe ou da superclasse e realizar qualquer tipo de rotina necessária no momento da criação do objeto.

Estes métodos também são chamados de construtores, e em Swift utilizam a palavra reservada `init` para serem criados no contexto da classe. Vamos a sintaxe:

```
init(parametro : TipoDoParametro) {
   //Indique os iniciadores
}
```

Vamos a um exemplo onde podemos ter inicializadores:

```
class Fahrenheit {
    var temperatura: Double

    init() {
        temperatura = 32.0
    }
}
```

Vamos comparar com uma sintaxe mais simples, sem o uso de inicializadores:

```
class Fahrenheit {
    var temperatura = 32.0
}
```

Na prática as duas sintaxes chegariam ao mesmo resultado, ou seja, teríamos armazenado o valor de 32.0 para a propriedade, e o mesmo seria utilizado na instância. Dessa forma, se não haverá customização da propriedade, podemos dizer que o correto é a não utilização de inicializadores.

No exemplo a seguir fica claro quando teremos a necessidade da utilização, ou seja, utilizaremos o bloco do `init` para customizar nossa propriedade:

```
class Conversor {
   var valorCelsius : Double

   init (valorFahrenheit : Double){
      valorCelsius = (valorFahrenheit - 32.0) / 1.8
   }
}

var fervuraAgua = Conversor(valorFahrenheit : 212.0)

print(fervuraAgua.valorCelsius)
//Resultado impresso em console: 100.0
```

Observers willSet e didSet

Os observers `didSet` e `willSet` fornecem uma maneira de responder corretamente quando uma propriedade tem seu valor definido/alterado.

O observador `willSet` é chamado antes de o valor ser atribuído a uma propriedade. Já o `didSet` é chamado depois de uma propriedade ter recebido um valor.

Quando utilizamos os observers temos a capacidade de captar os valores em dois momentos em tempo de execução. Temos o valor antes da alteração e depois da alteração.

```
let novoAbastecimento = Abastecer()

novoAbastecimento.contador = 8

novoAbastecimento.contador = 15

novoAbastecimento.contador = 23
```

```
class Abastecer {
   var contador : Int = Int() {
      willSet(novaContagem) {
       print(“Abastercer \(novaContagem) litros!”)
      }
      didSet {
         var novoValor = oldValue
         if contador > novoValor {
          print(“Abasteceu \(contador + novoValor)
            litros”)
         }
      }
   }
}
```

Lazy

A palavra reservada **lazy** é utilizada quando queremos retardar a criação de um objeto ou de algum processo que exija grande utilização de memória.

Veja o exemplo abaixo:

```
class Doces {
   var nomeDoce = String()
}

class Refeicao {
   lazy var sobremesa = Doces()
   var pratoPrincipal = String()
}

let almoco = Refeicao()
let posAlmoco = Doces()

almoco.pratoPrincipal = “Feijoada“
almoco.sobremesa.nomeDoce = “Pudim“

print (“Prato: \(almoco.pratoPrincipal)”)
print (“Sobremesa: \(almoco.sobremesa.nomeDoce)“)
```

Capítulo 13

Herança e polimorfismo

Seção 13-1

Herança

A capacidade de uma classe de captar as propriedades e métodos de outra classe é chamada de **herança**. Essa é uma das ferramentas mais poderosas da orientação a objetos, pois através dela podemos criar hierarquias e aumentamos o nível de abstração.

A **herança** nos fornece muitas possibilidades durante o desenvolvimento através da transferência de **propriedades** e **métodos** da **classe pai** para suas **subclasses.**

As classes pai, que são utilizadas na criação de novas classes são chamadas de **superclass,** ao passo que as classes que estão herdando as características das superclasses são chamadas de **subclasses.**

Vamos fazer a releitura da sintaxe de classes para entendermos como a implementação de heranças funciona:

```
class SubClass : SuperClassHerdada{

     // Definição da classe
}
```

Observe a utilização dos pois pontos **(:)** para a declaração, seguido do nome da superclasse.

#Dica!

A **subclass** pode ter suas próprias **propriedades** e **métodos**, e estes não podem ser acessados pela **superclass,** já que o fluxo da herança é sempre da superclass para a subclass.

Vamos criar uma classe que depois será herdada por outras classes:

```
//Classe Pai – SuperClass Humano
class Humano {
   var nome = String()
   var idade =  Int()

   func andar(){
      print(“O humano está andando”)
   }
}
```

Agora vamos criar outra classe que possa utilizar as propriedades e métodos de Humano como base:

```
//Classe Filha - Subclass Atleta
class Atleta : Humano {
   var esporte = String()
   var categoria = String()

   func indicarLesao(){
      print("O Atleta está lesionado")
   }
}
```

Se criarmos um objeto do tipo **Humano**, ele não terá acesso as propriedades de Atleta. Porém se criarmos um objeto do tipo **Atleta,** por ter herdado as propriedades de Humano, todas as opções estarão disponíveis para o objeto.

```
let maratonista = Atleta()

//Propriedades herdadas de Humano
maratonista.nome = "João"
maratonista.idade = 30

//Propriedades de Atleta
maratonista.esporte = "Atletismo"
maratonista.categoria = "Maratona"

maratonista.indicarLesao()
```

Seção 13-2

Polimorfismo

Um conceito importante dentro do paradigma de orientação a objetos é o **polimorfismo,** que é a capacidade de uma subclasse sobrescrever **métodos** e **propriedades** de uma **superclasse.**

Em Swift, utilizamos a palavra reservada `override` antes do nome do método para dizermos que este está sendo alterado.

A palavra `override` fará com que o compilador verifique se existe na superclasse o método em destaque. Este processo é altamente importante para definição correta de um método herdado.

Vamos criar a classe a ser utilizada como base:

```
//Classe Pai – SuperClass Humano
class Humano {
   func andar() {
      print(“Estou Andando”)
   }
}

let donaDeCasa = Humano()

donaDeCasa.andar()
//Resultado impresso em console “Estou Andando”
```

Agora vamos trabalhar a classe que herdará e modificará o método que criamos:

```
//Classe Filha – Subclass Atleta
class Atleta : Humano {
   override func andar() {
      print(“Estou Andando Rápido”)
   }
}

let corredor = Atleta()

corredor.andar()
//Resultado impresso em console “Estou Andando
Rápido”
```

Caso queira que a classe pai não dê a possibilidade de substituição, podemos definir a propriedade ou método com a palavra reservada `final` antes da nomenclatura:

```
//Classe Pai – SuperClass Humano
class Humano {
   final func andar() {
      print(“Estou Andando”)
   }
}
```

Capítulo 14

Xcode para UI

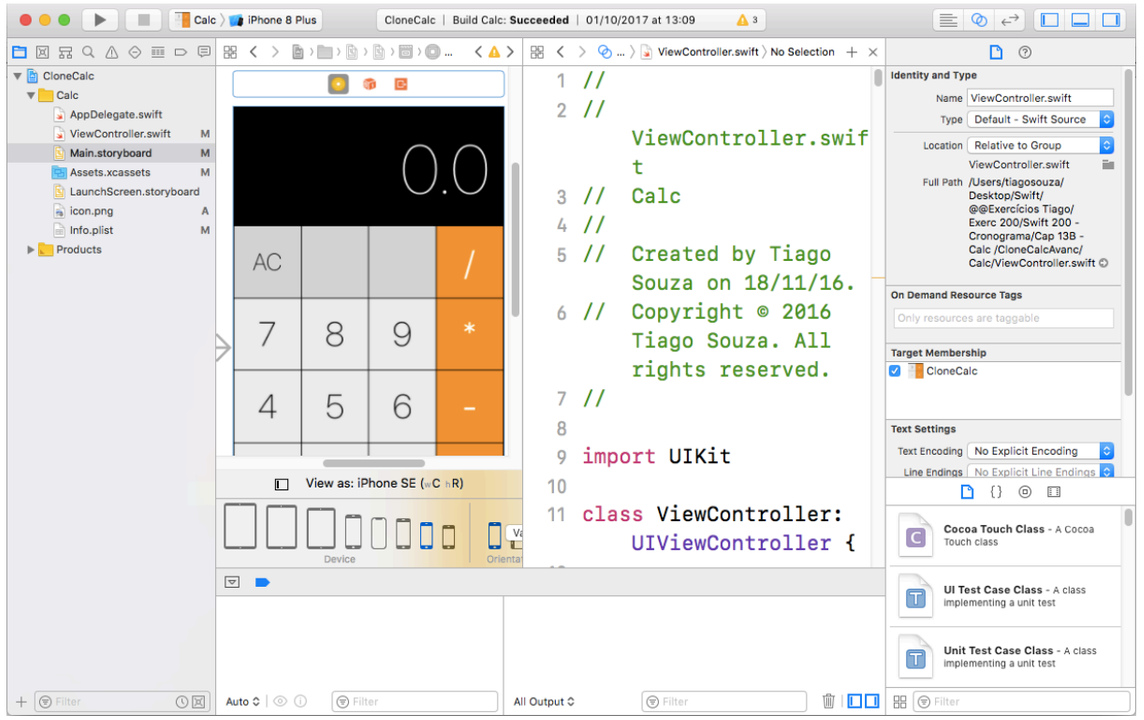
Xcode para UI

Xcode é o ambiente de desenvolvimento integrado da **Apple (IDE)** utilizado na criação de aplicativos para os produtos da Apple, incluindo iPad, iPhone, Apple Watch, e Mac.

O Xcode fornece ferramentas para gerenciar todo o fluxo de trabalho no desenvolvimento de aplicativos, desde testes, otimização e publicação na App Store.

Interface multitarefa

A interface do Xcode integra a edição de código, design de interface, gerenciamento de arquivos, teste e depuração dentro de uma janela espaço de trabalho simples. Você pode se concentrar em uma tarefa e exibir apenas o que você precisa, como código-fonte por exemplo, ou apenas o seu layout da interface do usuário, e alternar de forma simples e direta quando for necessário.



Interface Builder

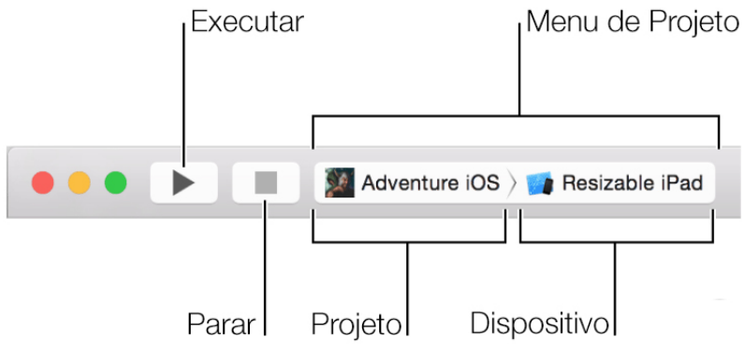
Interface Builder é um editor de design visual integrado ao Xcode, com ele podemos criar interfaces de usuário de seus aplicativos através da montagem de janelas, vistas, controles, menus e outros elementos de uma biblioteca de objetos configuráveis.

Outro conceito interessante no desenvolvimento da inteface, são os **Storyboards,** que são utilizados para especificar o fluxo de seu aplicativo e as transições entre as cenas.

Executando sua aplicação

Um dos recursos mais utilizados na criação de aplicativos já com interface é a execução do aplicativo no simulador do Xcode.

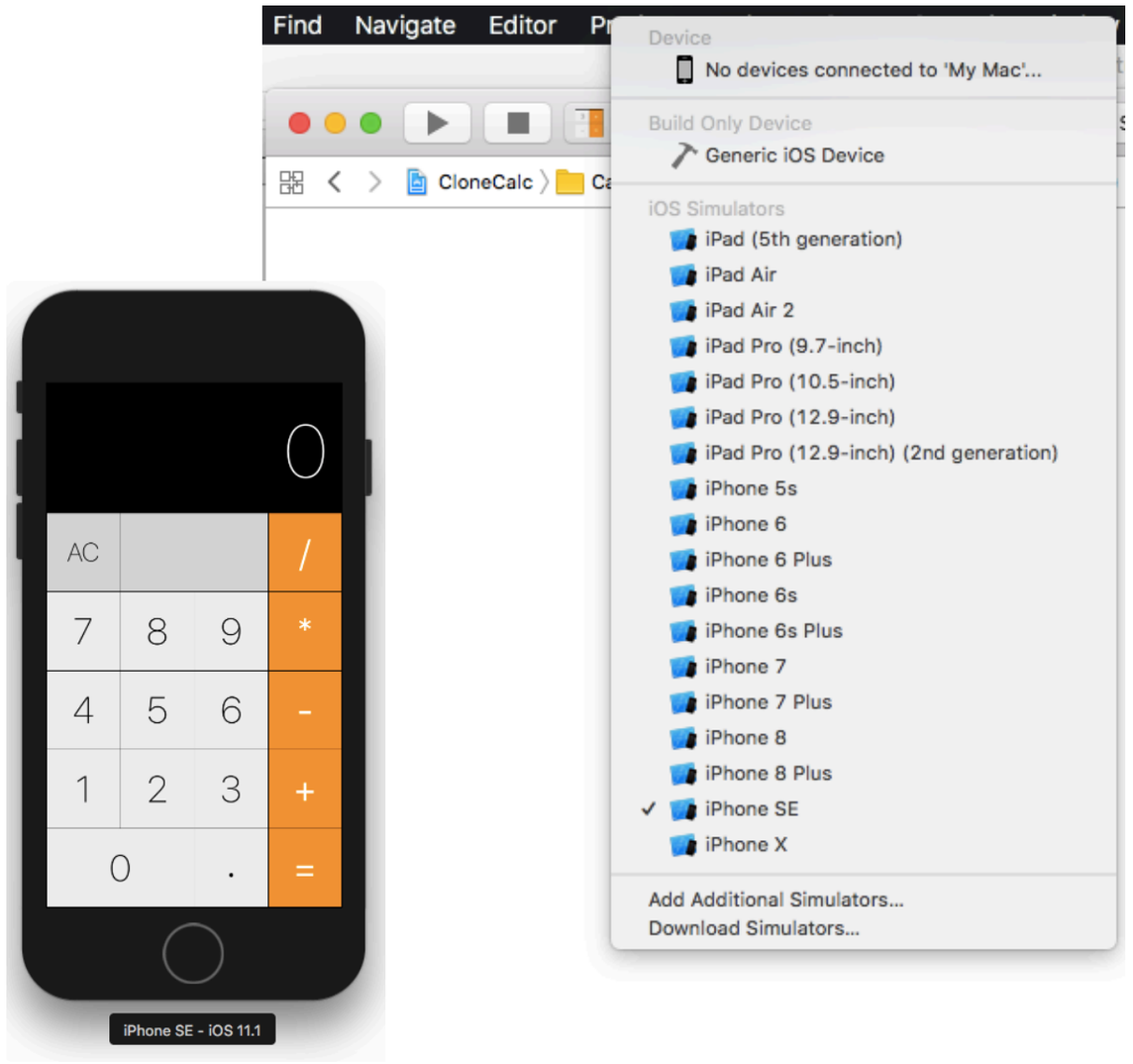
Temos um conjunto de botões e menus que nos possibilita executar e parar a simulação:



#Dica!

Uma alternativa ao botão executar é a combinação de teclas **⌘ + R.**

Podemos determinar qual dispositivo será simulado, basta escolher qual se encaixa em seu projeto na lista de dispositivos.



Além dessas áreas, ainda temos a disposição o console, onde serão exibidas as opções, mensagens e respostas que não serão visualizadas pelo usuário final da aplicação.

Capítulo 15

Preparando o projeto UI

Seção 15-1

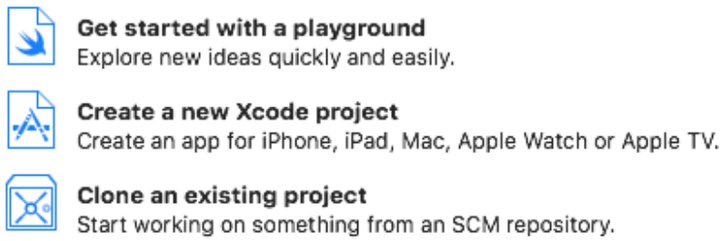
Iniciando o projeto com UIKit

Durante os capítulos iniciais, trabalhamos com o framework **Foundation,** que define a estrutura das classes, protocolos e funções para o desenvolvimento em dos elementos básicos e fundamentais em Swift.

Para trabalharmos com a interface gráfica, temos a necessidade de importarmos um outro framework, o **UIKit,** isso pode ser feito logo no início do código, assim como foi feito com o Foundation.

O **UIKit** é responsável pelos elementos de interface gráfica. É totalmente orientado para trabalhar com iOS.

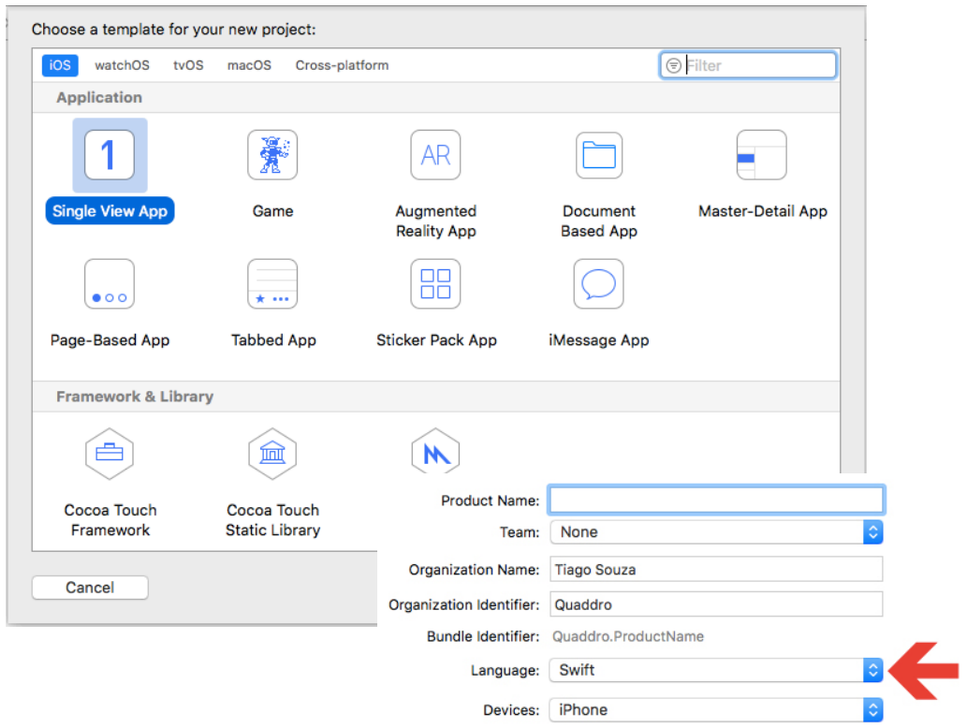
A partir daqui, deixaremos de lado o trabalho com arquivos Playground para iniciarmos a prática de projetos mais complexos. Para isso, na tela inicial do Xcode, vamos escolher a opção **Create a new Xcode project:**



Quando iniciamos um novo projeto, temos a opção de direcionar esse projeto para aplicativos iOS. Inicialmente vamos criar projetos com apenas uma vista **(Single View Application).**

Quando procedemos dessa forma, os frameworks **Foundation** e **UIKit** já estarão presentes no nosso projeto.

Também é importante atentar para a linguagem que será utilizada, nosso caso, **Swift.**



Heranças

Os elementos de **UIKit** invariável mente herdarão propriedades e métodos provenientes de classes já pré estabelecidas pelo **framework.**

A principal classe que irá orientar o UIKit é a **NSObject.** Dela derivará os objetos que vamos utilizar na nossa interface gráfica.

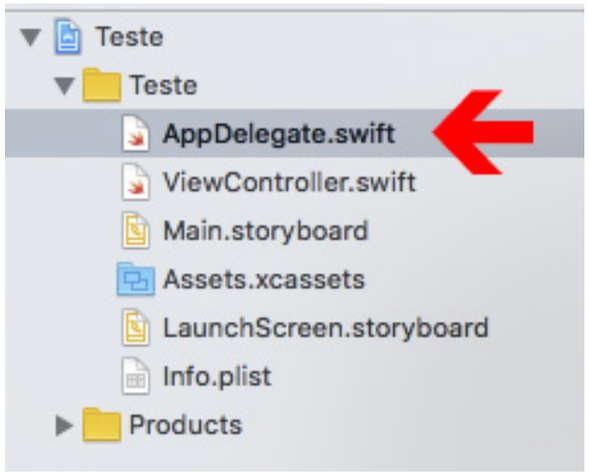
No exemplo a seguir temos um objeto de **UIView,** que é uma subclasse de **UIResponder,** que consequentemente é uma subclasse de **NSObject:**

É importante acompanharmos a trajetória de heranças dos nossos objetos, pois muitas propriedades e métodos que são importantes para a estrutura do elementos são provenientes de sua cadeia hierárquica.

Durante todo o decorrer desse módulo, cada objeto apresentado terá toda a sua herança exibida logo no início do capítulo correspondente.

Opções de AppDelegate

Quando começamos um novo projeto para trabalhar com UI, como um projeto por Single View por exemplo, automaticamente temos a nossa disposição dentro do projeto, um arquivo chamado **AppDelegate.swift:**



A classe **AppDelegate** é composta por métodos que são disparados em determinados momentos do fluxo da aplicação, como início, entrada ou saída de primeiro plano, ou término da aplicação.

A seguir temos a ação que cada uma das funções adiciona e também qual momento do fluxo cada uma delas agem:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool

   //Disparado logo após o lançamento da aplicação (Launch)

   return true

}


func applicationWillResignActive(_ application:
UIApplication) {

   //Disparado quando está prestes a passar do estado ativo
   para o estado inativo

}


func applicationDidEnterBackground(_ application:
UIApplication) {

   //Disparado quando a aplicação efetivamente entra em
   background

}


func applicationWillEnterForeground(_ application:
UIApplication) {

   // Disparado quando a aplicação vai passar do estado
   inativo para o estado ativo

}
```

```
func applicationDidBecomeActive(_ application:
UIApplication) {

   //Disparado quando a aplicação entra em primeiro plano

}


func applicationWillTerminate(_ application:
UIApplication){

   //Disparado antes as aplicação ser encerrada

}
```

#Dica!

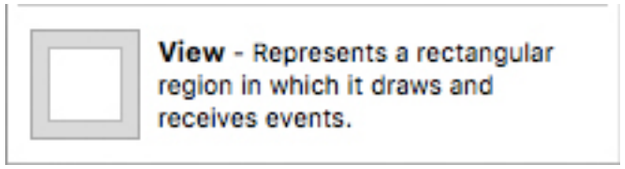
É importante que sejam feitos testes em conjunto com o **simulador, console** e **função print** para entendermos como o processo funciona.

Capítulo 16

Objetos Views

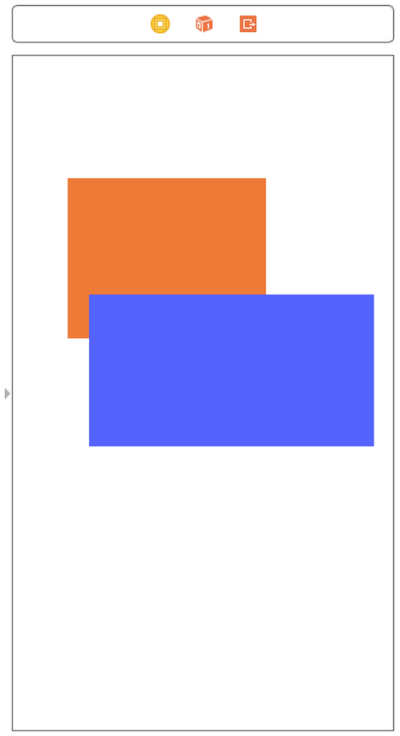
UIView

Herança: NSObject > UIResponder > UIView

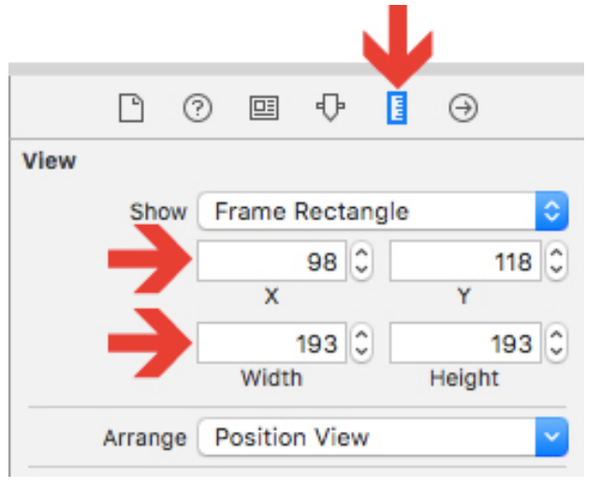


A classe **UIView** é a base para a maioria dos objetos gráficos presentes no **framework UIKit.** Por isso, grande parte dos seus recursos são compartilhados por objetos de outras subclasses como **UILabel** e **UIButton,** que veremos mais adiante.

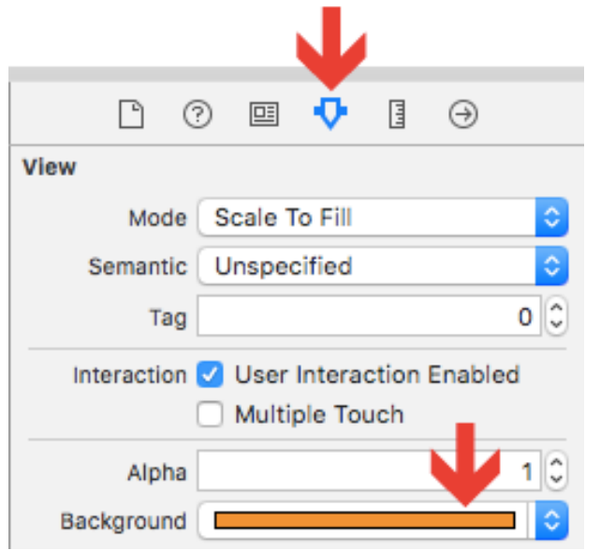
Basicamente podemos utilizar uma **UIView** para trabalharmos objetos de cor sólida.



Podemos **dimensionar** e alterar a **posição** do objeto através do painel **Size** exibido a seguir:



Também temos a opção de alterar a cor de fundo do objeto, presente no painel **Attributes**, opção **Background:**



Os objetos **UIView** possuem uma grande importância no desenvolvimento para **iOS,** pois executam tarefas que vão desde a renderização e animação de objetos gráficos, até o agrupamento de outros objetos com o conceito de **container.**

Devido a dimensão reduzida dos dispositivos com **iOS**, o desenvolvimento conta com apenas uma janela, denominada **window.**

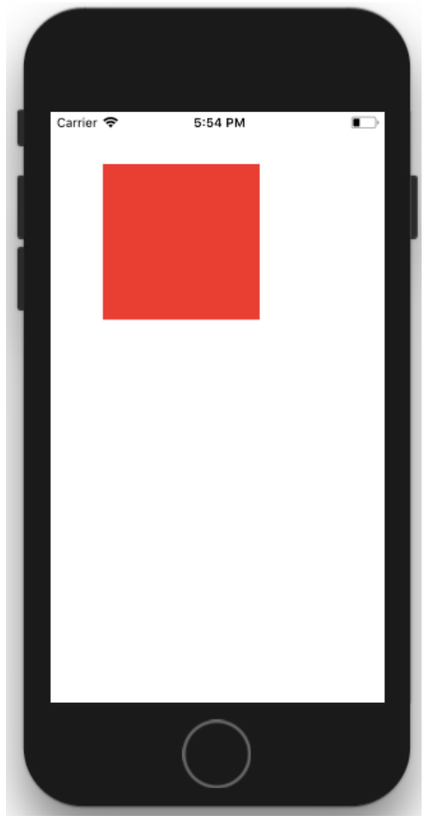
O complemento à essa window principal são as vistas, denominadas **views,** sendo estes objetos do tipo **UIView.** Desta forma podemos dizer que um aplicativo é composto por uma única **window** que internamente pode conter múltiplas **views.**

Trabalhando com códigos

Podemos alocar objetos do tipo view através da codificação do objeto, utilizando as instancias da classe UIView. Vamos a um exemplo da criação de um objeto:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let objeto = UIView(frame: CGRect(x: 50, y: 50,
        width: 300, height: 300))

        self.view.addSubview(objeto)

        objeto.backgroundColor = UIColor.red

    }
}
```

Observe que através do código também trabalhamos com a dimensão, posição e cores do objeto.



#Dica!

Objetos criados exclusivamente com códigos não serão exibidos no editor de interface gráfica do Xcode, mas podem ser visualizados diretamente no simulador.

É importante salientar que a classe **ViewController** é aquela que controla o que acontece com a sua respectiva view, no caso de um projeto **Single View.** Já o método **viewDidLoad** começa a atuar assim que o objeto view referente aquela ViewController é carregada.

Propriedades

No exemplo anterior trabalhamos algumas propriedades provenientes de dois tipos diferentes, `CGRect` e `UIColor`.

Quando definimos a posição e dimensão do objeto, utilizamos as propriedades `x`, `y`, `width` e `height` de `CGRect`.

Já para a cor de fundo, utilizamos a propriedade `backgroundColor` do tipo `UIColor`.

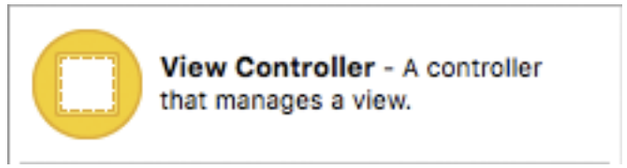
Principais métodos

Temos a disposição alguns métodos que podem ser utilizados em conjunto com objetos do tipo View:

```
.addSubview(UIView)
//Adiciona um objeto UIView no container

.removeFromSuperview()
//Remove o objeto da superview e da responder chain

.viewWithTag(Int)
//Retorna o objeto UIView definido pela tag especificada

init(frame: CGRect)
//Inicializa e retorna um novo objeto a partir do CGRect
especificado
```

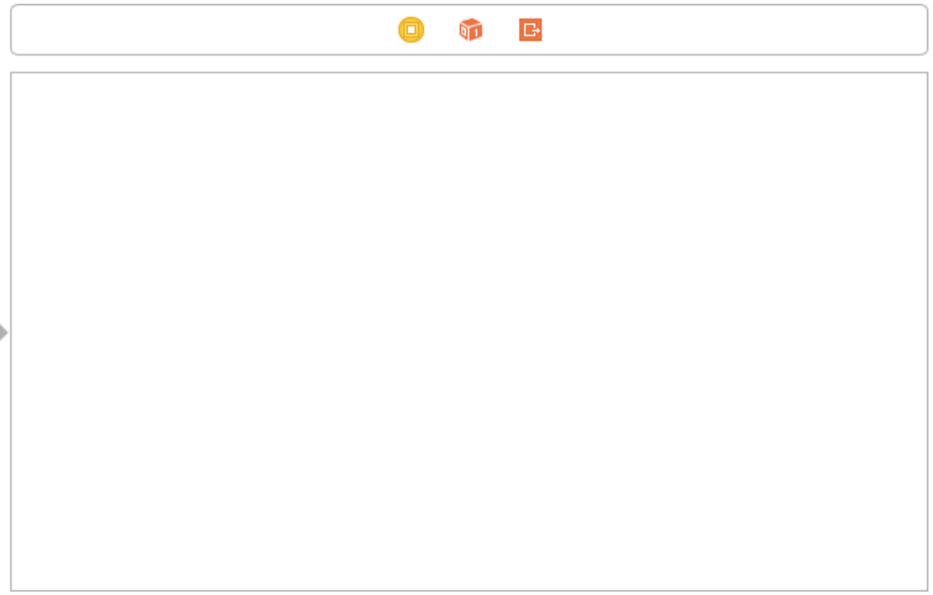
UIViewController

Herança: NSObject > UIResponder > UIViewController



Uma vez que a classe UIView representa, além dos objetos gráficos, a própria tela de um aplicativo, precisamos de uma classe que faça o controle destes elementos. Nesse momento entra em ação a **UIViewController.**

Os objetos do tipo **UIViewController** são responsáveis por gerenciar um conjunto de views, desde a sua inicialização até o seu término, ou seja, desde o momento que uma UIView é carregada ou descarregada na tela.



#Dica!

Não se esqueça que objetos de texto, botões ou imagens são filhos de UIView, desse modo, quando falamos que precisamos de um UIViewController para cuidar de UIViews, isso também inclui os objetos citados.

Objetos UIViewController raramente são utilizados sozinhos, porém podemos ter diversos objetos desse tipo dentro de uma aplicação para iOS. Todo UIViewController possui pelo menos uma view principal.

A classe **UIViewController** também serve como base para outros tipos de controllers, tais como **UITableViewController** utilizado para a criação de tabelas, **UICollectionViewController** para a criação de coleções de objetos, ou ainda para **UINavigationController** que é muito útil quando precisamos criar barras de navegação.

Definindo tamanho e orientação

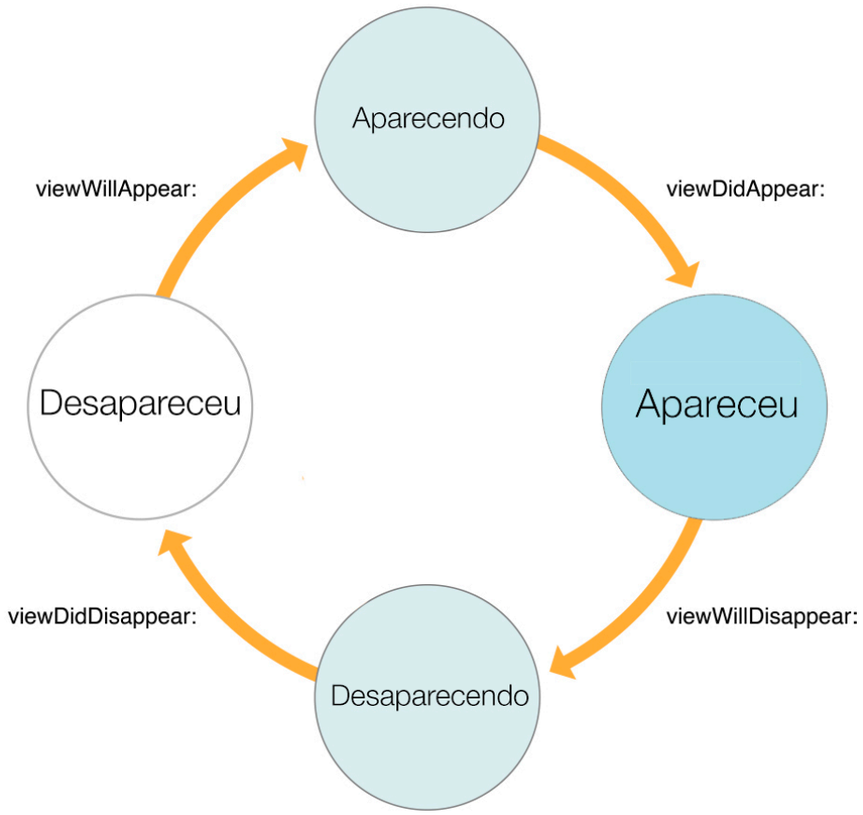
Podemos definir um tamanho base e um esquema de orientação para um UIViewController. Ambas opções estão disponíveis no painel **View As,** disponível na base do Storyboard corrente:

Métodos Relacionados ao gerenciamento da vista

Quando a visibilidade das suas vistas mudam, um controlador de vistas chama automaticamente os seus próprios métodos para que subclasses possam responder à mudança.

Existem quatro momentos que controlam essas mudanças. Quando a vista está sendo carregada, quando a vista está na tela, quando ele está desaparecendo, e quando está fora da tela.

A figura a seguir ilustra a estrutura de funções que controlam esses momentos:




```
override func viewWillAppear(_ animated: Bool) {
   //Método disparado quando a view irá aparecer
}

override func viewDidAppear(_ animated: Bool){
   //Método disparado quando a view de fato aparece
}
```

```
override func viewWillDisappear(_ animated: Bool) {
   //Método disparado quando o elemento vai desaparecer
}

override func viewDidDisappear(_ animated: Bool) {
   //Método disparado quando o elemento já desapareceu
}
```

Outros métodos

Além dos métodos que gerenciam as vistas, temos alguns outros que complementam o trabalho com UIViewControllers.

```
override func didReceiveMemoryWarning(){
      super.didReceiveMemoryWarning()

   //Método é disparado quando o aplicativo recebe um viso
   de memória do iOS

}


func performSegue(withIdentifier identifier: String, sender:
Any?){

   //Método que dispara um Segue para um objeto identificado
   no arquivo storyboard

}
```

```
func prepare(for segue: UIStoryboardSegue, sender: Any?){
   //Método que notifica que um segue será executado.
}

func viewDidLoad(){
   super.viewDidLoad()
   //Método disparado depois que a view foi carregada em
   memória.
}

func present(_ viewControllerToPresent: UIViewController,
animated flag: Bool, completion: (() -> Void)? = nil){
   //Método que exibe uma ViewController podendo conter
   animação
}
```

#Dica!

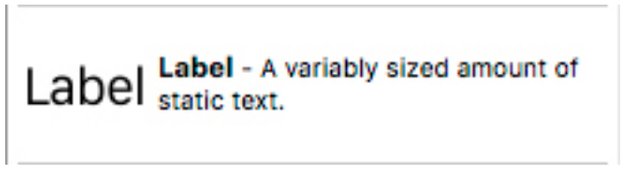
Os métodos **didReceiveMemoryWarning** e **viewDidLoad** já estarão incorporados ao código do arquivo **ViewController.swift** já na criação do projeto.

Capítulo 17

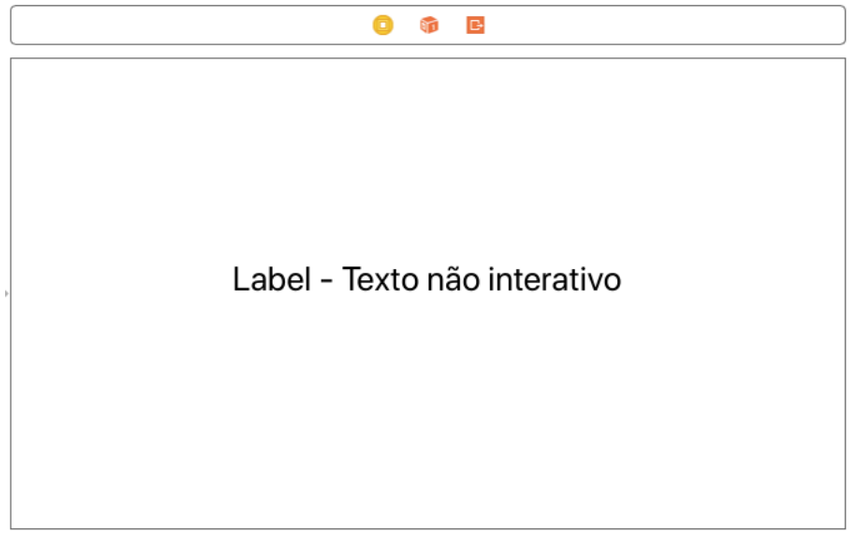
Objetos básicos do UIKit

UILabel

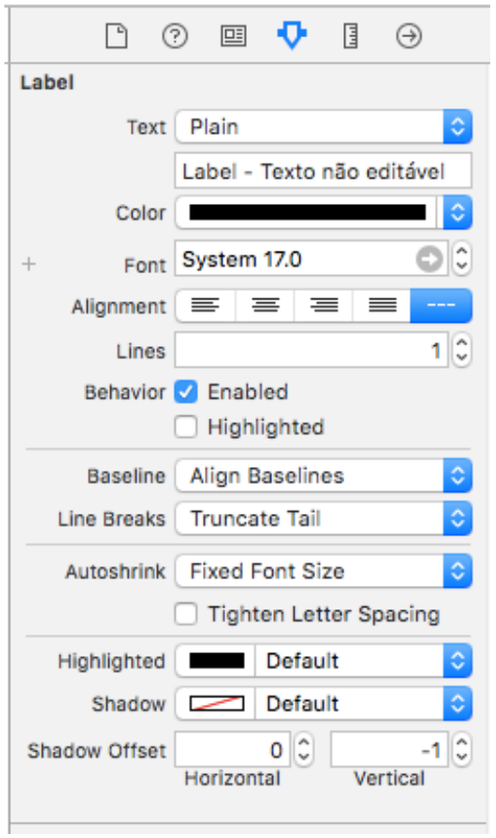
Herança: NSObject > UIResponder > UIView > UILabel

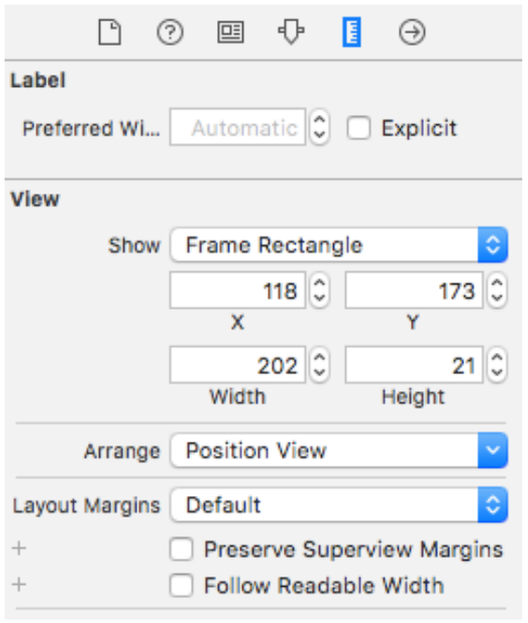


O objeto `UILabel` é responsável por exibir **textos não editáveis** em uma ou múltiplas linhas. Esses textos ajudam a identificar partes da interface do usuário.



Esse tipo de objeto é altamente configurável e podemos editar a cor do texto, o tipo de fonte, a sombra, entre outras propriedades no painel **Attributes,** e seu posicionamento e dimensões no painel **Size.**





É importante destacar que objetos **Labels** são amplamente utilizados nas aplicações.

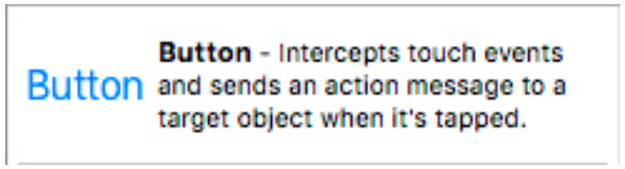
Vários objetos possuem **Labels** internos, como por exemplo o **UIButton** e o **UITableViewCell.**

Principais propriedades

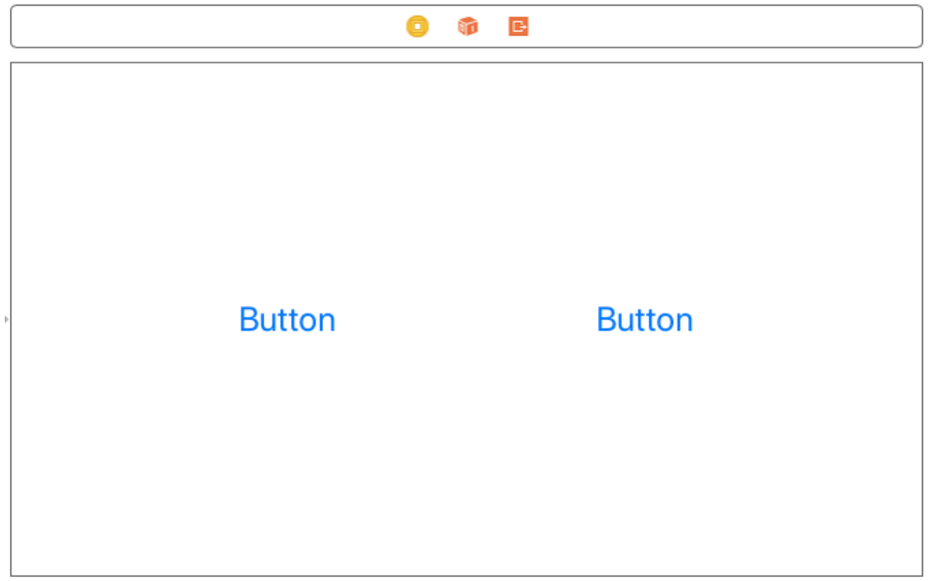
Propriedade	Tipo	Descrição
font	UIFont	Define / Retorna a fonte usada
minimumScaleFactor	CGFloat	Define / Retorna o tamanho mínimo da fonte
numberOfLines	Int	Define / Retorna o número máximo de linhas
text	String	Define / Retorna o texto exibido
textAlignment	NSTextAlignment	Define / Retorna o tipo de alinhamento do texto
textColor	UIColor	Define / Retorna a cor do texto

UIButton

Herança: NSObject > UIResponder > UIView > UIControl > UIButton



Recurso utilizado para colocar botões em suas aplicações. Um objeto `UIButton` é capaz de interceptar um evento de toque e disparar métodos **IBAction** que veremos adiante.



Podemos entender os objetos **UIButton** como gatilhos que podem executar métodos que realizarão uma ou varias tarefas determinadas pelo programador.

A classe **UIButton** também fornece propriedades para definirmos o título, a imagem e outros elementos referentes a sua aparência, posição e dimensões, da mesma forma como fizemos com Labels e Views.

É possível definir estas propriedades para cada estado do objeto, como quando o botão é clicado, por exemplo.

Principais propriedades

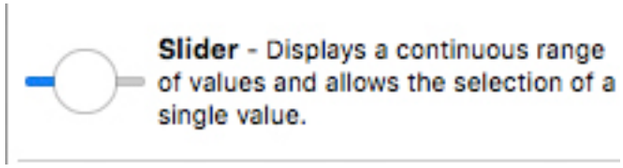
Propriedade	Tipo	Descrição
currentTitle	String?	Retorna o título atual
currentTitleColor	UIColor	Retorna a cor atual do título
titleLabel	UILabel?	Retorna o objeto label que exibe o título
showsTouchWhenHighlighted	Bool	Gera um efeito visual no centro do botão quando o mesmo é tocado
currentTitleShadowColor	UIColor?	Retorna a cor atual da sombra do título
currentBackgroundImage	UIImage?	Retorna a imagem de fundo exibida no botão

Principais métodos

```
.setImage(_ image: UIImage?, for state: UIControlState)

//Método que atribui uma imagem a um estado específico


.setBackgroundImage(_ image: UIImage?, forState:
UIControlState)

//Método que atribui uma imagem de fundo a um estado específico


.setTitle(_ title: String?, forState state:
UIControlState)
//Método que define o titulo para um estado específico

.setTitleColor(_ color: UIColor?, forState state:
UIControlState)
//Método que define a cor do titulo para um estado específico
```

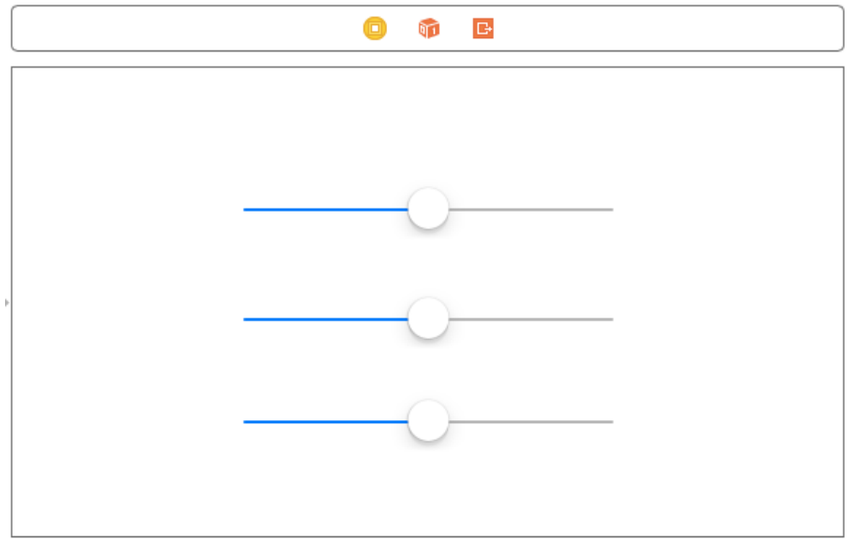
UISlider

Herança: NSObject > UIResponder > UIView > UIControl > UISlider



O objeto visual `UISlider` é responsável por exibir e selecionar um valor à partir de um intervalo pré-definido pelo usuário.

Seu visual é uma barra horizontal, onde se encontra um indicador, tecnicamente chamado de **Thumb**, que nos permite selecionar o valor desejado.



Principais propriedades

Propriedade	Tipo	Descrição
tintColor	UIColor!	Define/Retorna a cor do menu
minimumValue	Float	Define o valor mínimo para o slider
maximumValue	Float	Define o valor máximo para o slider
value	Float	Define com qual valor o slider será iniciado para o slider

#Dica!

O **intervalo padrão** de um **Slider** varia de **0.0** até **1.0.** Os valores mínimos e máximos podem ser definidos pelo programador tanto por código como também no painel **Attributes.**

Principais métodos

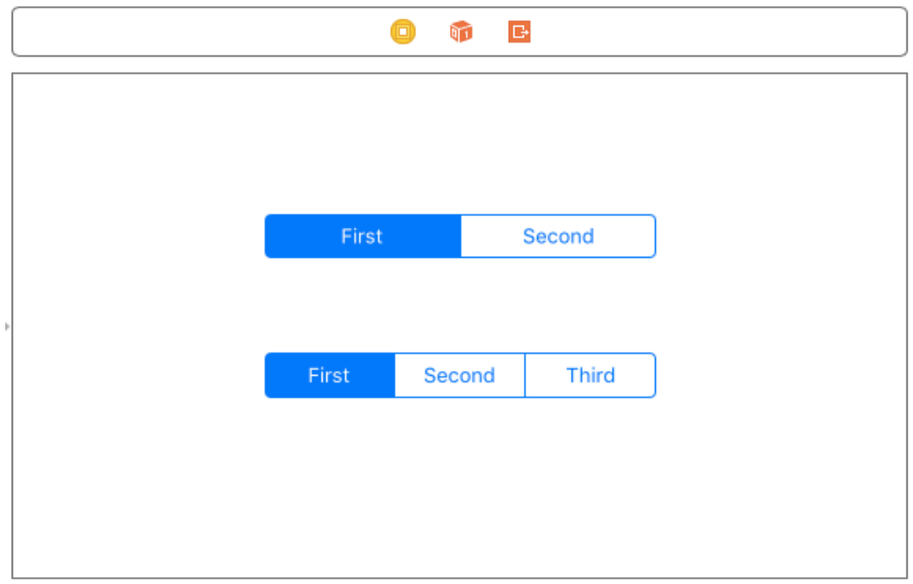
```
.setValue(_ value: Float, animated: Bool)

//Esta função permite definir manualmente um valor para o
estado inicial do slider



.setThumbImage(_ image: UIImage?, for state: UIControlState)
//Permite definir uma imagem customizada para o thumb do
Slider em um estado específico
```

Seção 17-4
UISegmentedControl

Herança: NSObject > UIResponder > UIView > UIControl > UISegmentedControl

O objeto `UISegmentedControl` é responsável por exibir em um único conjunto horizontal segmentos que irão atuar como botões separados, permitindo a cada um deles disparar ações (métodos) diferentes.

Esse tipo de elemento pode exibir um texto ou ainda uma imagem, que serão automaticamente ajustadas para o tamanho definido para o elemento.



Principais propriedades

Propriedade	Tipo	Descrição
numberOfSegments	Int	Retorna a quantidade de segmentos.
selectedSegmentIndex	Int	Define/Retorna o índice do segmento selecionado
tintColor	UIColor!	Define/Retorna a cor do menu

Pro padrão, um segmented control funciona como uma estrutura de **radio buttons**, ou seja, apenas um dos botões pode estar ativo por vez. A escolha é baseada em **índices,** e assim como em um **array,** o valor é iniciado em zero.

Caso queira iniciar um segmented control com nenhum dos elementos selecionados, temos que utilizar a propriedade `selectedSegmentIndex`, e definir pra ela o valor **-1.**

Principais métodos

```
.setImage(_ image: UIImage?, forSegmentAt segment: Int)

//Define uma imagem a ser utilizada em um seguimento indicado
pelo índice
```

```
.setTitle(_ title: String?, forSegmentAt segment: Int)

//Define um título a ser utilizado em um seguimento indicado
pelo índice



.titleForSegment(at segment: Int) -> String?

//Método que retorna o título do segmento da posição definida



init(items: [Any]?){

//Método que define, inicializa e retorna um menu de acordo
com o Array de itens fornecido

}
```

```
.removeSegment(at segment: Int, animated: Bool)

//Método que remove o segmento da posição definida.
Essa operação pode ser animada



.setEnabled(_ enabled: Bool, forSegmentAt segment: Int)

//Método que habilita ou desabilita o segmento da posição
definida.



.isEnabledForSegment(at segment: Int) -> Bool

//Método que retorna se o segmento da posição definida está
ou não habilitado
```

IBOutlets, IBActions e IBOutletCollections

Quando criamos elementos de interface gráfica direto no editor, sem a utilização de linhas códigos para a sua criação, não temos nenhum link desses elementos com os arquivos de código.

Esse link pode ser feito através dos recursos de **Interface Builder (IB)** chamados de **Outlets** e **Actions.**

É importante saber que a **IBOutlet** e **IBAction** são macros em **Objective-C,** e são declaradas pelo arquivo **UINibDeclarations.h** da seguinte forma:

```
#define IBAction void
#define IBOutlet
```

Em Swift, **IBAction** e **IBOutlet** são **constantes** e estão declaradas dentro do framework **UIKit.**

Utilizando IBOutlet

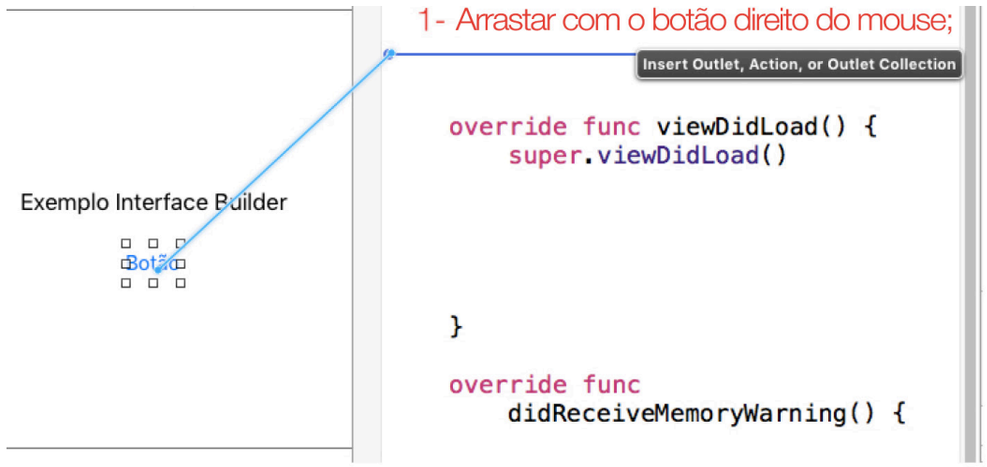
Podemos entender um **IBOutlet** como um canal de ligação entre o objeto gráfico e o código.

A partir dele podemos alterar as propriedades do objeto, como por exemplo o texto de uma label, a posição de um slider ou até mesmo a cor de fundo de uma view.

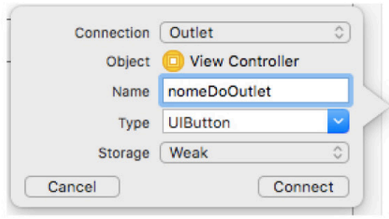
Sua sintaxe é bastante simples como podemos observar no exemplo abaixo:

```
@IBOutlet weak var exemploBotao: UIButton!
```

Não temos a necessidade de fazer a entrada da sintaxe manualmente. Podemos simplesmente arrastar o botão direito o objeto a ser declarado, e soltá-lo na área do código a ser utilizado:



2- Nomear o objeto



Utilizando IBAction

IBAction é utilizada para que um objeto visual dispare um **método** declarado no seu código. Sua sintaxe em Swift é a seguinte:

```
@IBAction func chamarUmMetodo (_ sender: AnyObject)
```

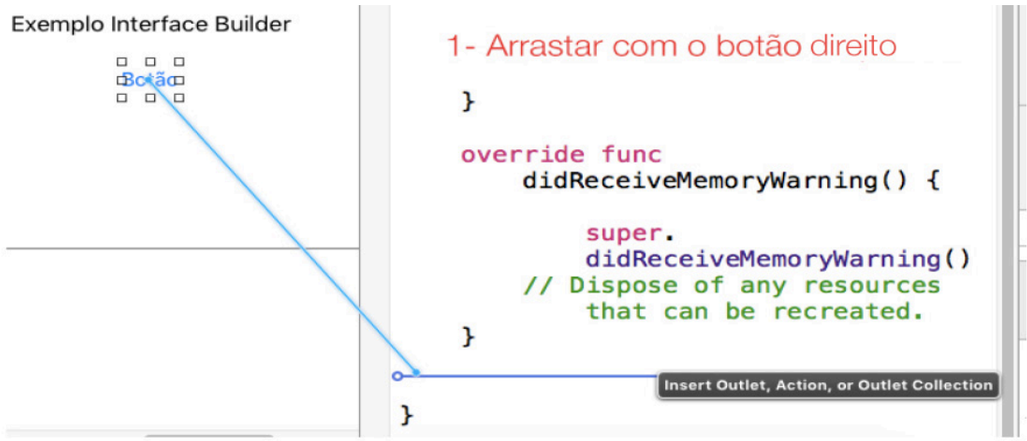
Podemos dizer que `chamarUmMetodo` é o método propriamente dito, e `sender` é o objeto visual que disparou o método.

Podemos também definir o tipo do `sender`, indicando o objeto diretamente, como `UIButton` ao invés de `AnyObject` por exemplo.

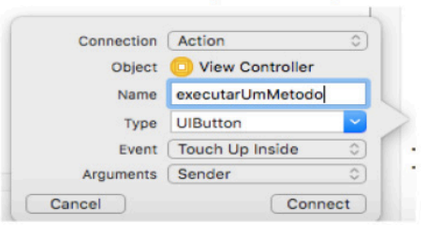
Se necessário, podemos passar um argumento extra do tipo `UIEvent`, que indica qual foi o tipo de evento o método disparou, veja a seguir um exemplo:

```
@IBAction func chameMeuMetodo(_ sender: AnyObject, forEvent
event: UIEvent)
```

Da mesma forma, como fizemos com os Outlets, podemos definir uma Action de um objeto ao arrastá-lo com o botão direito e soltá-lo na área do código desejada:



2- Escolher o tipo de Connection, nomear e escolher o tipo de objeto



#Dica!

É muito comum criarmos Outlets por engano quando queremos na verdade criar Actions. Fique de olho no campo **Connection** no momento da ação para que esse tipo de acidente não ocorra.

Coleções de Outlets - IBOutletCollection

Podemos condicionar em uma mesma linha de código diversos Outlets que formarão na verdade um **array** que contem uma coleção de **Outlets.**

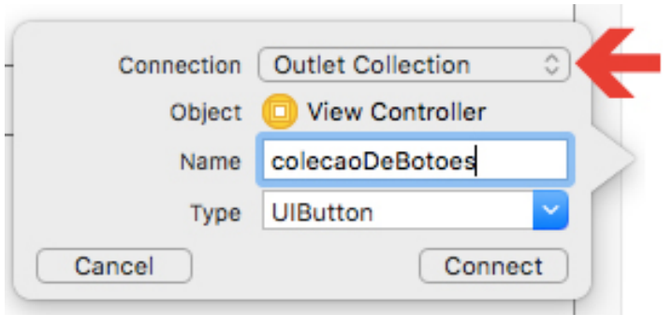
Essa é uma pratica bastante útil quando desejamos configurar propriedades de elementos de interface do mesmo tipo utilizando algum tipo de enumerador.

Devemos declarar um tipo IBOutletCollection da seguinte forma:

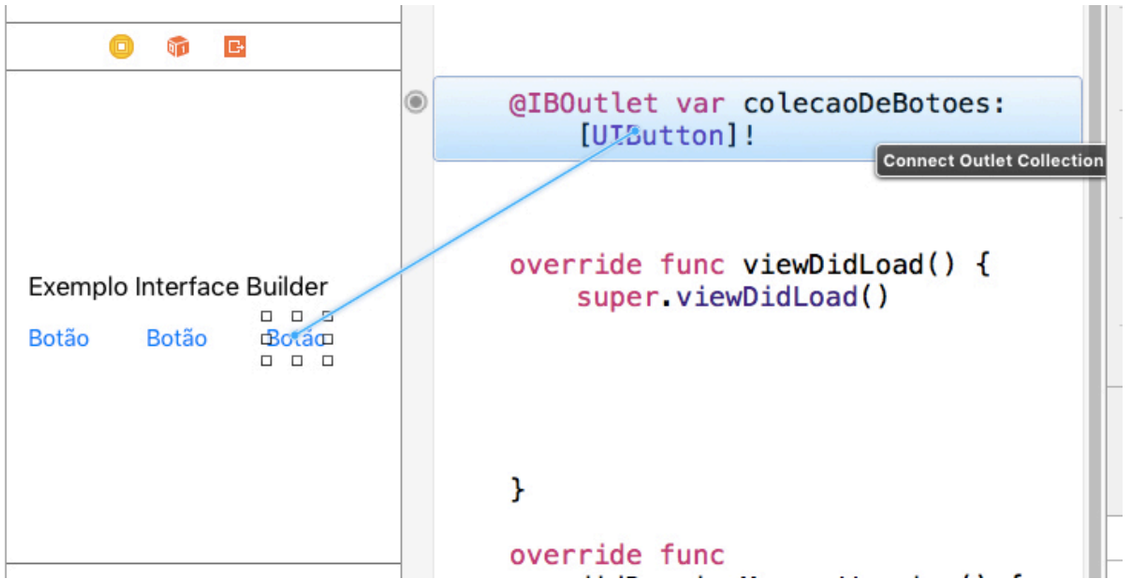
```
@IBOutlet var labels: [UILabel]!
```

Observe que utilizamos **labels** como exemplo, mas pode-se utilizar o recurso com diversos tipos de objetos de interface.

Para criarmos um **IBOutletCollection,** podemos proceder da mesma forma como fizemos com Outllets simples, porém devemos indicar o tipo de conexão a ser realizada:



Em seguida devemos arrastar também com o botão direito, os outros objetos que farão parte da coleção e soltá-los em cima do código do array criado no passo anterior:



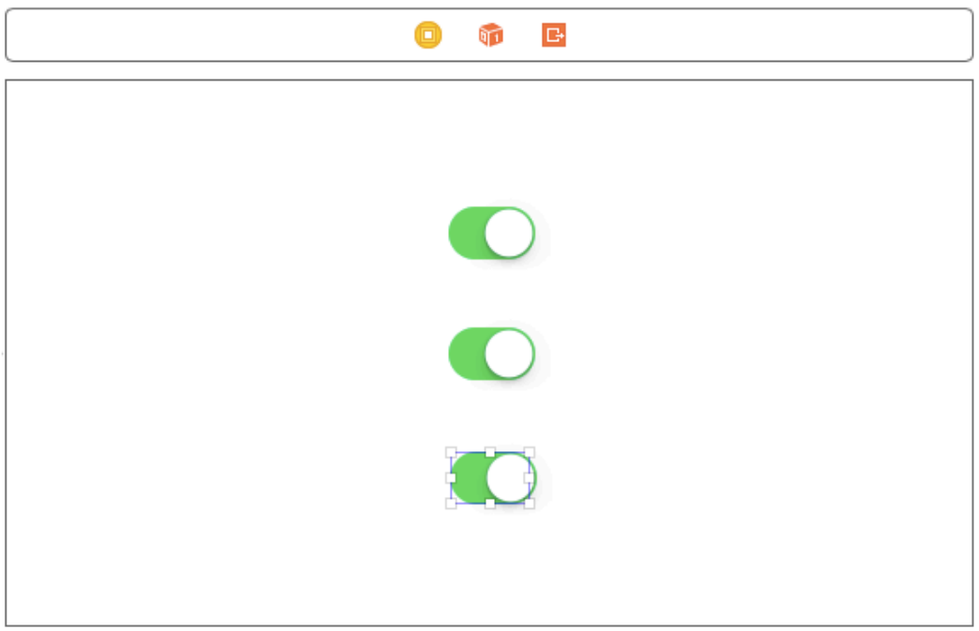
Capítulo 18

UISwitch

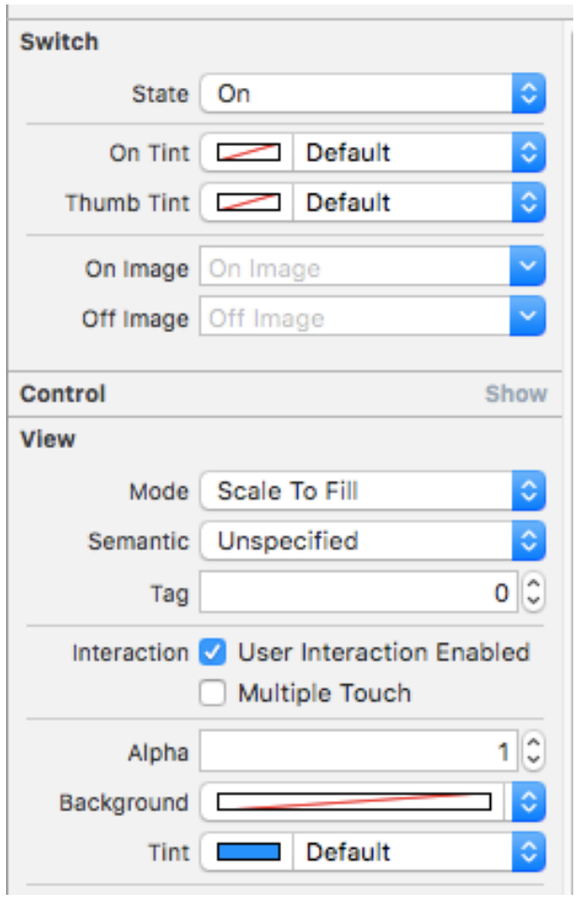
UISwitch

Herança: NSObject > UIResponder > UIView > UIControl > UISwitch

Utilizamos a classe `UISwitch` para criar e gerenciar **botões de Liga/Desliga** utilizados em definições tais como Airplane Mode e Bluetooth por exemplo. Estes objetos são conhecidos como interruptores **(Switchers).**



Você pode personalizar a aparência do objeto mudando a cor usada no botão central, bem como os arredores do objeto. Podemos ter cores diferentes para os estados ligado/desligado.



#Dica!

O objeto central do switch é chamado de Thumb.

Principais propriedades

Propriedade	Tipo	Descrição
isOn	Bool	Define se o objeto será ligado ou desligado
onTintColor	UIColor?	Define a cor do estado ligado
tintColor	UIColor!	Define a cor do estado desligado
thumbTintColor	UIColor?	Define a cor do objeto central

Principais métodos

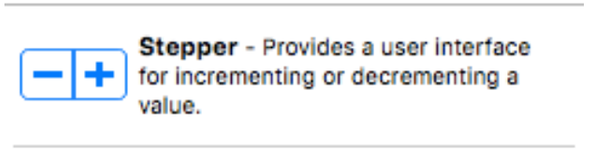
```
init(frame: CGRect)

//Retorna um objeto inicializado


.setOn(_ on: Bool, animated: Bool)

//Define o estado do switch (ligado ou desligado) com a opção
de animar a ação
```

Capítulo 19

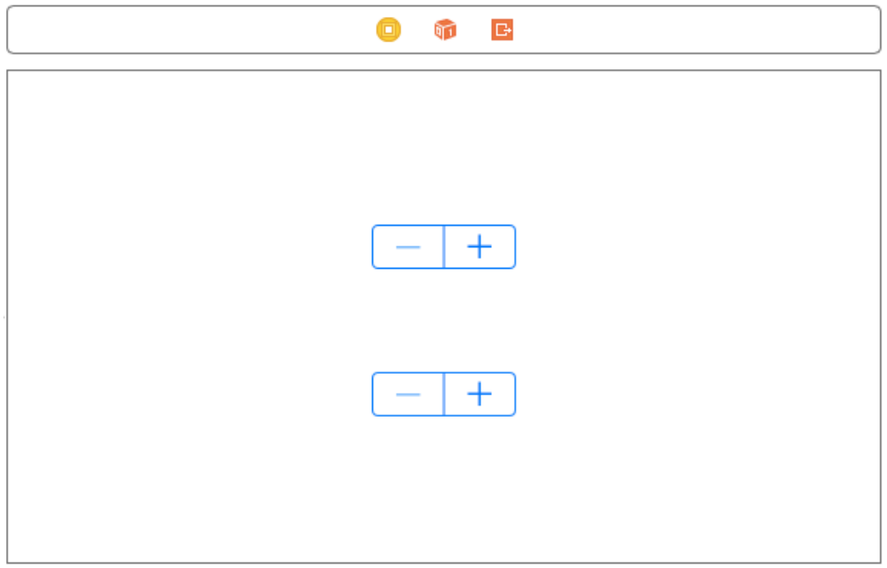
UIStepper

UIStepper

Herança: NSObject > UIResponder > UIView > UIControl > UIStepper

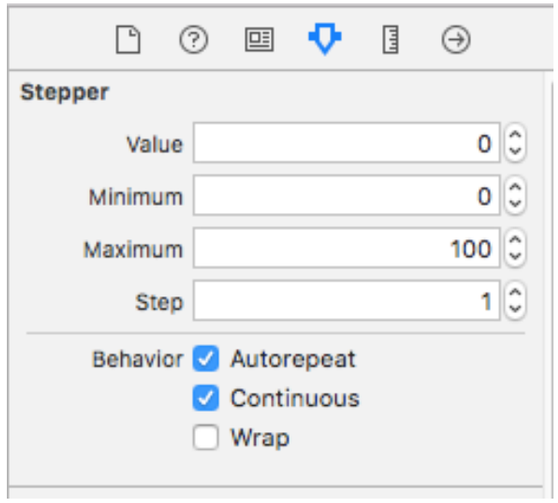


Um `UIStepper` fornece uma interface para **aumentar ou diminuir** um valor. O objeto conta com dois botões, um com um **sinal de menos (-)** e um com um **sinal de mais (+).**



Podemos definir o comportamento de um `UIStepper` para **auto repetição** (padrão), pressionando e segurando um de seus botões para variar o valor repetidamente. A taxa de variação depende de quanto tempo o usuário continua pressionando o controle, e os **intervalos** também são definidos pelo programador.

Os valores **mínimo** e **máximo** podem ser definidos de acordo com a necessidade do objeto.



Principais propriedades

Propriedade	Tipo	Descrição
isContinuous	Bool	Define se a alteração do valor do objeto respeitará o pressionamento contínuo do botão
minimumValue	Double	Define o valor mínimo a ser atingido
maximumValue	Double	Define o valor máximo a ser atingido
stepValue	Double	Define o intervalo de valor a ser utilizado
value	Double	Define o valor inicial

Principais métodos

```
.setDecrementImage(_ image: UIImage?, for state:
UIControlState)

//Define uma imagem como símbolo de decremento



.decrementImage(for state: UIControlState) -> UIImage?

//Retorna a imagem que está sendo usada como símbolo de
decremento



.setIncrementImage(_ image: UIImage?, for state:
UIControlState)

//Define uma imagem como símbolo de incremento



.incrementImage(for state: UIControlState) -> UIImage?

//Retorna a imagem que está sendo usada como símbolo de
incremento
```

Capítulo 20

Type Casting

Seção 20-1

Type Casting

A utilização do **Type Casting** é a melhor maneira de descobrir a origem dos tipos. Com a utilização de **casting** poderemos detectar qual foi o tipo utilizado para o instanciação do objeto. Esta é uma maneira muito eficaz na pesquisa da hierarquia das classes.

Iremos aprender como utilizar os operadores `is` e `as` para a verificação dos valores atribuídos aos tipos.

Com o primeiro **(`is`)** podemos fazer a verificação do tipo e utilizar o recurso em contagens de verificação. Já com o segundo (`as`) podemos ter o retorno de um objeto com uma nova conversão.

Nas próximas seções vamos aplicar os conceitos em pequenas aplicações e testar suas funcionalidades.

Operador is - Verificando o tipo

O operador `is` checa o tipo de uma instância de um determinado tipo. A verificação com `is` retorna `true` no caso da instância ser do mesmo tipo que o verificado, caso contrário, retorna `false`.

Confira o exemplo com a aplicação adiante:

```
//Superclasse Função
class Funcao {
   var nome : String

   init(umNome : String){
      self.nome = umNome
   }
}

//Subclasse Engenheiro baseada em Função
class Engenheiro: Funcao {
   var area : String

   init(umNome : String, umaArea : String){
      self.area = umaArea
      super.init(umNome : umNome)
   }
}

//Subclasse Arquiteto baseada em Função
class Arquiteto: Funcao {
   var ramo : String

   init(umNome: String, umRamo: String){
      self.ramo = umRamo
      super.init(umNome : umNome)
    }
}

//Instâncias Engenheiros e Arquitetos em um Array:
let todasFuncoes = [
Engenheiro(umNome: “João”, umaArea: “Civil”),
Engenheiro(umNome: “José”, umaArea: “Elétrica”),
Engenheiro(umNome: “Mario”, umaArea: “Hidráulica”),
Arquiteto(umNome: “Oscar”, umRamo: “Concreto”),
Arquiteto(umNome: “Santiago”, umRamo: “Metálica”),
Arquiteto(umNome: “Vilanova”, umRamo: “Espaços”)
]

//Controles para checagem de tipo
var quantidadeEngenheiros = 0
var quantidadeArquitetos = 0
```

```
//Acrescentando elementos aos controles de checagem
for profissional in todasFuncoes{
   if(profissional is Arquiteto){
      quantidadeArquitetos += 1
   }

   if(profissional is Engenheiro){
      quantidadeEngenheiros += 1
   }
}

//Impressão dos resultados em console
print("Número de Engenheiros.:
\(quantidadeEngenheiros)")
print("Número de Arquitetos.:
\(quantidadeArquitetos)")
```

Teremos impresso o resultado em console conforme solicitado, com as devidas contagens e separações.

Operador as - Downcasting

O operador `as` tem a finalidade provocar um **Downcasting**, ou seja, com ele podemos chegar no objeto fonte da instância desejada. Este operador pode ser utilizado de duas formas:

Na primeira podemos utilizar o operador sozinho **(`as`!),** o que irá forçar o **casting.** Na segunda forma **(`as`?),** o sistema tentará fazer o casting, mas caso falhe irá retornar um objeto nulo **(nil).**

Como exemplo, vamos continuar trabalhando com a estrutura da aplicação construída na seção anterior:

```
for funcionario in todasFuncoes{
  if let engenheiro = funcionario as? Engenheiro{
     print("Nome: \(engenheiro.nome) \nÁrea: \
     (engenheiro.area)\n")

  }else if let arquiteto = funcionario as? Arquiteto{
           print("Nome: \(arquiteto.nome) \nÁrea:
    \(arquiteto.ramo)\n")
  }
}
```

A utilização do operador **`as`?** em conjunto com a palavra reservada `let` proporciona uma maneira mais segura para descobrirmos a origem da instância de um tipo.

Capítulo 21

Gerenciamento Automático de memória com ARC

Seção 21-1

ARC - Automatic Reference Counting

ARC, ou **Automatic Reference Counting,** gerencia de maneira eficaz a utilização da memória nos dispositivos da Apple quando programamos com a Swift. Com o **ARC,** o programador não precisará preocupar-se com o gerenciamento de variáveis alocadas em memória.

Toda vez que criamos uma nova instância de uma classe, o **ARC** aloca na memória uma informação sobre aquela instância. A memória contem informações sobre o tipo da instância, assim como o valor de qualquer propriedade associada com aquela instância.

Quando a instância não é mais necessária, o **ARC** libera a memória alocada para aquela instancia, permitindo que ela seja utilizada para outros propósitos.

#Dica!

Programando no **Playground**, o ARC tem o uso limitado quando desejamos observar como as variáveis são desalocados da memória.

Vamos ao exemplo

```
class Jogador {
   var nome: String
   var idade: Int
   var sexo: Character

   init(nome: String, idade: Int, sexo: Character){
      self.nome = nome
      self.idade = idade
      self.sexo = sexo

      print(“Dados do jogador foram inicializados”)
   }

   deinit {
      print(“Foram retirados os dados do Jogador”)
   }
}

var jogador1 : Jogador?
jogador1 = Jogador(nome: “Jose Silva”, idade: 20,
sexo: “M”)

jogador1 = nil
```

Utilizamos a palavra reservada `init` para apresentar o processamento quando um tipo é instanciado. O método `init`, também conhecido como método inicializador, é chamado assim que o objeto é definido, podendo inclusive receber parâmetros para atribuir valores para as propriedades da classe.

O método `deinit` é chamado assim que o objeto é desalocado, podendo executar um bloco de código para liberar memória, ou executar alguma outra funcionalidade.

Com o **ARC,** a linguagem de programação Swift rastreará e gerenciará a utilização da memória pelo aplicativo.

Não existe necessidade de gerenciamento mais detalhado por parte do programador pois o gerenciamento automático sempre será feito pelo **ARC.**

Apêndice 1

Caderno de Exercícios

Code Style para Swift

Exercício 1:

Utilizando conceitos de **indentação** e **CamelCase** escreva alguns exemplos.

a - Um exemplo com abertura e fechamento de **chaves;**

b - Um exemplo com abertura e fechamento de **colchetes;**

c - Um exemplo com abertura e fechamento de **parenteses;**

d - Seu nome em **UpperCamelCase;**

e - o nome da Quaddro Treinamentos em **lowerCamelCase.**

Exercício 1:

Utilizando os conceitos de **comentários,** crie uma sequência de três comentários com linhas únicas, cada uma das linhas deve conter um nome, uma cor e um número.

Exercício 2:

Utilizando os conceitos de **comentários,** crie quatro **marcos** no seu código, um para cada ponto cardeal.

Exercício 3:

Utilizando a função **print,** crie uma aplicação que imprima em console o nome do seu filme preferido.

Exercício 4:

Determine para uma **variável** o nome de um país:

a - Imprima em console o conteúdo dessa variável;

b - mude valor dessa variável para outro nome de país;

c - Imprima novamente o conteúdo da variável.

Exercício 5:

Crie constantes para o mês de novembro, onde:

a - Seja declarado de forma implícita, onde o valor conste o número pelo qual esse mês é representado no calendário;

b - Seja declarado de forma explícita, onde o tipo seja levado em consideração e o valor seja o número pelo qual esse mês é representado no calendário;

c - Seja declarado de forma explícita, onde o tipo seja levado em consideração e o valor seja o nome do mês por extenso;

Exercício 6:

Utilizando variáveis e interpolação, crie um sistema de saudação que imprima essas informações em console:

`Olá, meu nome é ???, tenho ??? anos. Sou um profissional da área de ??? e meu e-mail para contato é ???.`

Cada valor indicado por ??? deverá ter sua própria variável.

Operadores

Exercício 1:

Crie uma aplicação que imprima em console dois números escolhidos e mostre as operações matemáticas básicas utilizando esses números, veja o exemplo:

```
***** Operações Matemáticas *****


Os números escolhidos foram:
Número A = 20     Número B = 5

Somar  = 25

Subtrair = 15

Multiplicar = 100

Dividir = 4


***** Fim das operações *****
```

Exercício 2:

Crie uma aplicação que imprima em console dois números escolhidos e mostre as relações possíveis entre eles, veja o exemplo:

```
***** Relações entre os Números *****


Os números escolhidos foram:
Número A = 10     Número B = 5

Número A é igual ao Número B: false

Número A é menor que Número B: false

Número A é maior que Número B: true

Número A é menor ou igual ao Número B: false

Número A é maior ou igual ao Número B: true

Número A é diferente do Número B: true


***** Fim das operações *****
```

Exercício 1:

Criar uma aplicação que ordene os planetas de acordo com a sua distância com relação a terra, com as seguintes propriedades:

- Nome
- Distância

Através de uma variável de controle, fazer com que o sistema exiba as informações relativas a um planeta. Veja um exemplo de exibição impresso em console:

```
***** Controle de Planetas *****

Número de controle: 3

Planeta:
Nome: Vênus
Distância da terra: 149.600.000 Km

***** Fim das operações *****
```

Condicionais

Exercício 1:

Dados três valores para um produto em três lojas diferentes, crie uma aplicação que imprima em console qual dos três valores é o mais barato, e ao final exiba todos os valores utilizados. Vamos ao exemplo:

```
***** Comparação de Preços *****

Preço da loja AAA é o mais em conta!

Preço na loja AAA: R$ 59,90.
Preço na loja BBB: R$ 69,90.
Preço na loja CCC: R$ 68,00.
```

Exercício 2:

Crie uma aplicação capaz de identificar se uma letra é vogal ou consoante. Atente para a capitalização das letras. Veja o exemplo de exibição em console:

```
***** Indicador de Vogal *****

Letra indicada: A

Letra A é Vogal.
```

Controle de Fluxo

Exercício 1:

Crie uma aplicação que imprima em console um contador de vai de 50 até 100 contando de 10 em 10.

```
***** Contador *****

O valor é: 50
O valor é: 60
O valor é: 70
O valor é: 80
O valor é: 90
O valor é: 100
```

Exercício 2

Crie uma aplicação que imprima em console os meses do ano e seu respectivo indicador numérico:

```
          ***** Meses *****
Jan é o mês 1              Jul é o mês 7
Fev é o mês 2              Ago é o mês 8
Mar é o mês 3              Set é o mês 9
Abr é o mês 4              Out é o mês 10
Mai é o mês 5              Nov é o mês 11
Jun é o mês 6              Dez é o mês 12
```

Exercício 3:

Crie ama aplicação que imprima em console um contador regressivo de 100 até 0 que conte de 10 em 10.

```
Faltam 100
Faltam 90
Faltam 80
Faltam 70
Faltam 60
Faltam 50
Faltam 40
Faltam 30
Faltam 20
Faltam 10
```

Exercício 1:

Crie uma função que imprima em console o valor de dois números e a soma de ambos:

```
***** Função para soma de dois valores *****

Primeiro valor informado: 10
Segundo valor informado: 20

A soma dos dois valores é: 30
```

Exercício 2:

Crie uma função que imprima em console o valor de múltiplos números e a soma de todos eles:

```
***** Função para soma de múltiplos valores *****

Os números indicados foram: [1, 2, 3, 10]

O resultado da soma dos valores é 16
```

Operadores Customizados

Exercício 1:

Crie um operador customizado **prefix,** usando os caracteres **<->,** que acrescente a cada valor **+2.**

Exercício 2:

Crie um operador customizado **postfix,** usando os caracteres **--,** que sempre retorne a **metade** do valor quando utilizado.

Exercício 1:

Utilizando **enum,** crie uma aplicação que utilize controles de movimento de 0 até 3 para definir uma das quatro direções, esquerda, direita, cima e baixo. Quando mudarmos o numero de controle, em console teremos a direção que escolhermos. Veja o exemplo:

```
0 = Movimento para esquerda

1 = Movimento para direita

2 = Movimento para cima

3 = Movimento para baixo
```

Optional Binding

Exercício 1

Crie uma aplicação que imprima em console o número de série de um produto quando informado. Caso esse número seja nulo, o sistema deverá informar que o número de série não foi fornecido:

```
Produto com o número de série: 555888/NYXPTO

Número de série não informado
```

Classes

Exercício 1:

Crie uma **classe** para inclusão de contatos em uma agenda telefônica, com funções para impressão das informações em console. Veja o exemplo:

```
***** Agenda Telefônica *****

Nome: Michelangelo Buonarroti
Residencial: 5555-5555
Comercial: 6666-6666
Celular: 92222-2222

Nome: Rafael Sanzio
Residencial: 8888-8888
Comercial: 7777-7777
Celular: 93333-3333
```

Herança e polimorfismo

Exercício 1:

Crie uma classe para **Ambiente**, onde tenha as propriedades **largura, comprimento, altura.** Estabeleça um método que imprima esses valores, e forneça a **área em m²** conforme o exemplo:

```
O comprimento do ambiente é 0.0m
A largura do ambiente é 0.0m
A altura do ambiente é 0.0m
O ambiente tem 0.0m² de área
```

Agora vamos herdar a classe **Ambiente** em duas novas classes, uma chamada **Cozinha,** e outra chamada **Dormitório.**

Para **Cozinha,** teremos a propriedade **tipo de pia** (mármore, granito, inox, etc), e um método para **imprimir** essa informação.

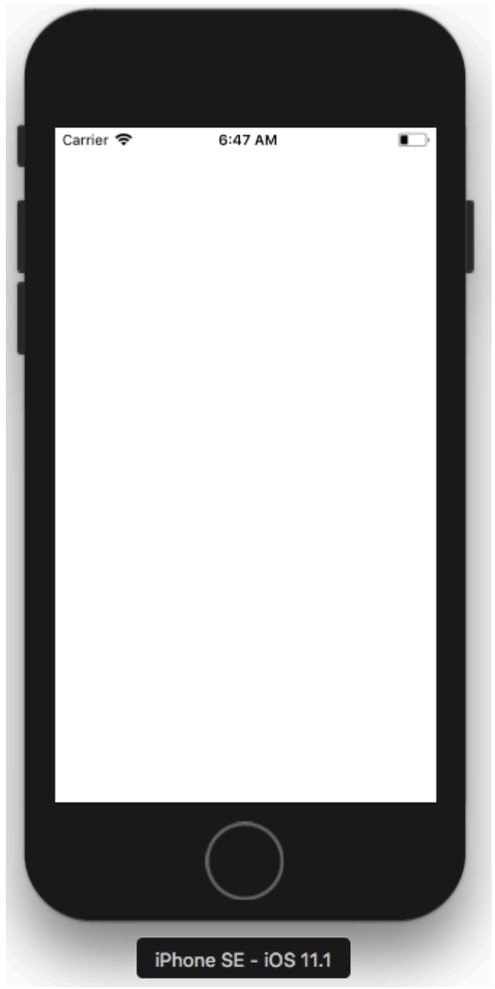
Para **Dormitório,** teremos a propriedade que confirma ou não a presença de **ponto de internet,** e um método para **imprimir** essa informação.

Como resultado final, devemos ter algo parecido com o exemplo:

```
Sala de estar
O comprimento do ambiente é 3.0m
A largura do ambiente é 3.0m
A altura do ambiente é 2.7m
O ambiente tem 9.0m² de área

Cozinha
Pia de Aço Inox
O comprimento do ambiente é 3.75m
A largura do ambiente é 3.25m
A altura do ambiente é 2.7m
O ambiente tem 12.1875m² de área

Dormitório 1
Ponto de internet = true
O comprimento do ambiente é 3.88m
A largura do ambiente é 4.25m
A altura do ambiente é 2.7m
O ambiente tem 16.49m² de área
```

Exercício 1

Utilizando os conceitos relacionados os capítulos 14 e 15, crie um novo projeto Single View, utilizando a linguagem Swift, que rode em um simulador de iPhone SE.



Objetos Views

Exercício 1

Utilizando o objeto grafíco UIView, posicione no storyboard dois elementos com as dimensões de 100x100. Para o primeiro objeto, utilize a cor de fundo vermelha, e para o segundo a cor amarela.

Exercício 2

Trabalhando com códigos, aloque em seu projeto um objeto UIView, que tenha 200 de comprimento e 100 de altura, e esteja posicionado a 50 em x e 100 em y.

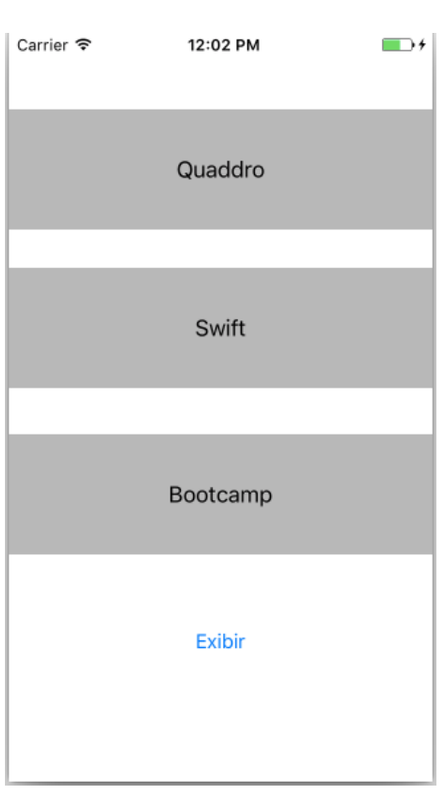
Para esse objeto, defina a cor de fundo verde.

Objetos básicos do UIKit

Exercício 1

Crie uma aplicação que contenha três objetos **labels** e um **buttom.** Cada vez que o usuário clicar no botão, uma label vazia recebe seu respectivo texto, conforme o exemplo:

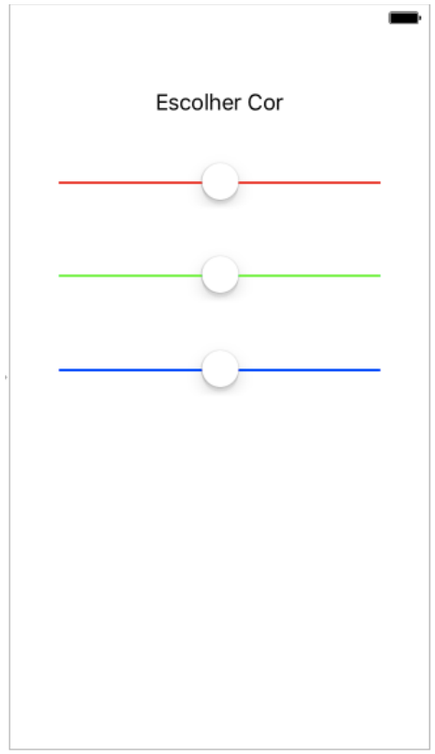
- “Quaddro”
- “Swift”
- “BootCamp”

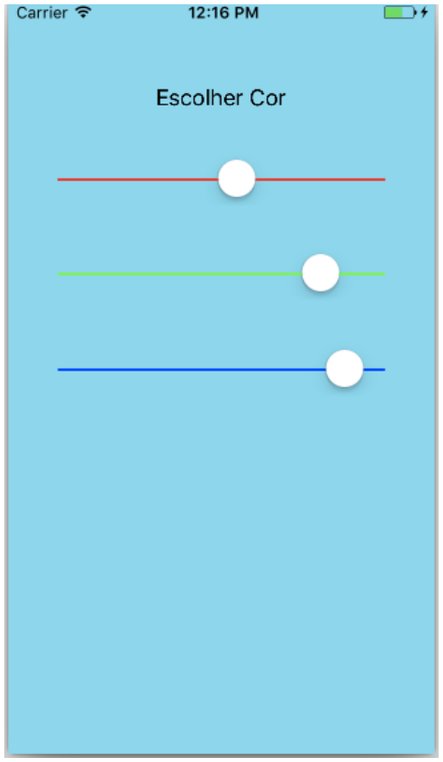


Exercício 2

Crie uma aplicação que contenha três **sliders**. Cada slider deve controlar um valor relacionado a escala de cores **RGB.**

Ao interagir com os sliders a **cor de fundo da view controller** deve se alterar de acordo com seus valores.



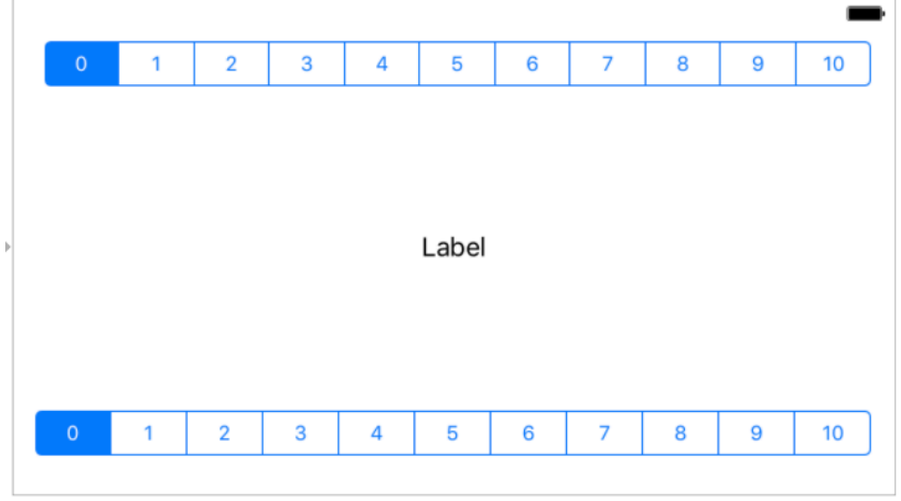


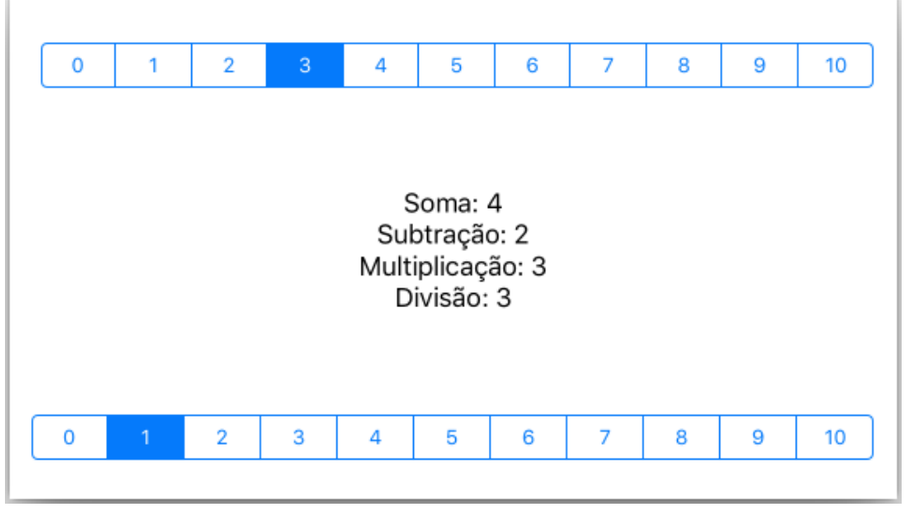
Exercício 3

Crie uma aplicação que contenha dois **segmenteds controls** e uma **label,** onde os segmenteds controls exibem valores de **0 a 10.**

Sempre que houver uma seleção deve ser exibido na label a seguinte relação entre os valores dos segmenteds:

- Soma
- Subtração
- Multiplicação
- Divisão

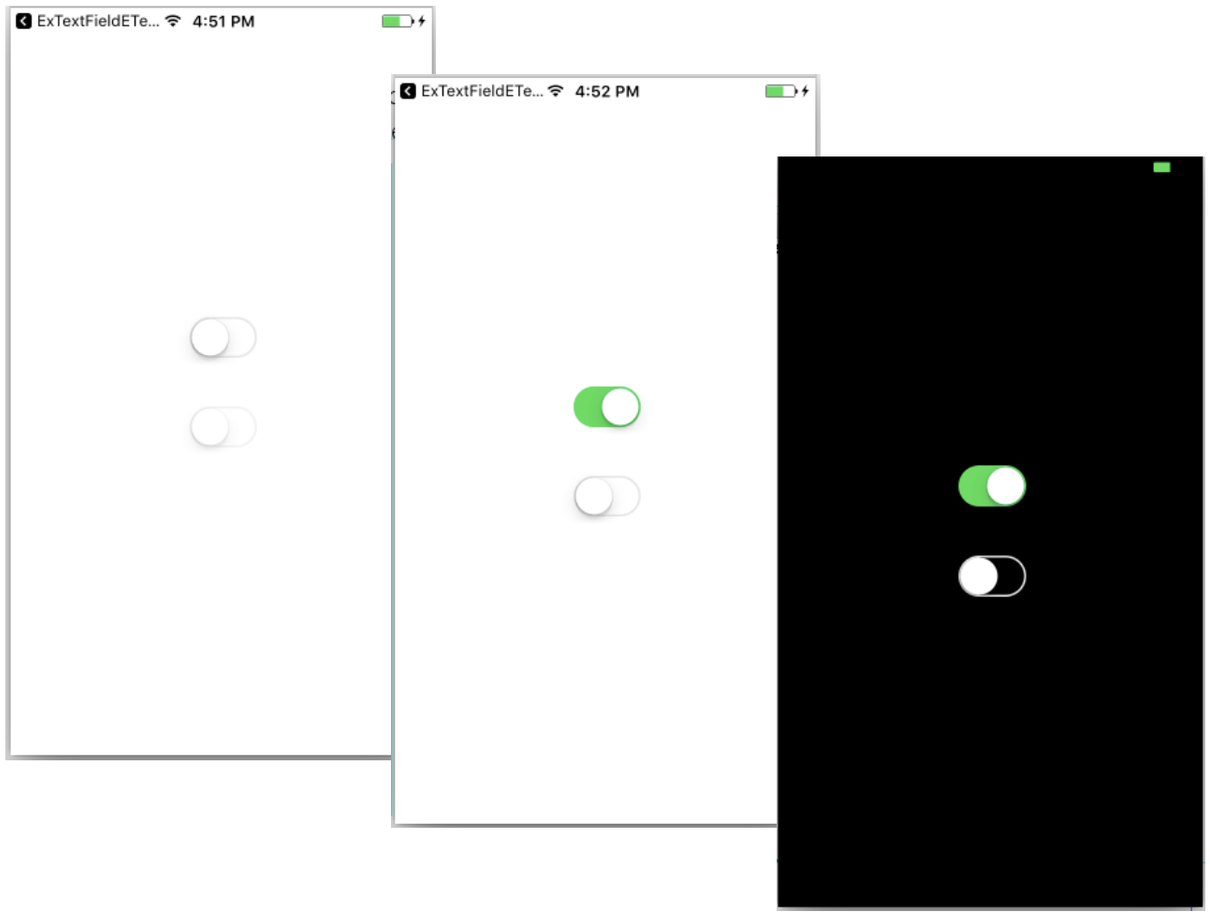




UISwitch

Crie uma aplicação que contenha dois **switches**, o primeiro deverá habilitar ou desabilitas a interação com o segundo.

O segundo por sua vez, servirá para alternar o background da view entre branco e preto de acordo com sua seleção.



Exercício 1

Crie uma aplicação que contenha um **stepper** e exiba em um objeto **label** os números primos existentes entre 1 e 100 de acordo com a seleção do usuário (**- ou +**), sendo os números primos de 1 até 100: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.

Type Casting

Exercício 1:

Dada uma lista com dez itens de mídia, sendo 5 delas álbuns musicais e 5 filmes, crie estrutura necessária para classificar, separar, contar e imprimir essa contabilidade conforme o exemplo:

Galeria:
Filme (nome: Casablanca, diretor: Michael Curtiz)
Filme (nome: Cidadão Kane, diretor: Orson Welles)
Filme (nome: Avatar, diretor: James Cameron)
Filme (nome: Psicose, diretor: Alfred Hitchcock)
Filme (nome: E.T, diretor: Steven Spielberg)

Música (nome: The Wall, artista: Pink Floyd)
Música (nome: Thriller, artista: Michael Jackson)
Música (nome: Blue Suede Shoes, artista: Elvis Presley)
Música (nome: Help!, artista: The Beatles)
Música (nome: Innuendo, artista: Queen)

Exemplo a ser impresso em console:

```
Biblioteca de mídia contém 5 filmes e 5 álbuns
musicais.
```

Utilizando typealias

```
class Filme {
   var titulo: String
   var ano: Int

   init(titulo: String, ano: Int) {
      self.titulo = titulo
      self.ano = ano
   }

   func dados() {
      var saida: String =
      "----------------------------------------"
      saida += "\nFilme:\(titulo) \t Ano:[\(ano)]"
      print(saida)
   }
}

var bttf: [Filme] = [
Filme(titulo: "De Volta para o Futuro I" ,
ano: 1985),
Filme(titulo: "De Volta para o Futuro II" ,
ano: 1989),
Filme(titulo: "De Volta para o Futuro III",
ano: 1990)
]

print("\nTotal de Filmes da Trilogia BTTF : \(bttf.
count) filmes\n")
for filme in bttf {
   filme.dados()
}

//Utilizando o typealias para clonar a estrutura
typealias Cinema = Filme
typealias Shopping = Filme
typealias Diversao = Filme

bttf.append(Cinema (titulo: "De Volta para o Futuro
IV", ano: 2085))

bttf.append(Shopping (titulo: "De Volta para o
Futuro V" , ano: 2089))

bttf.append(Diversao (titulo: "De Volta para o
Futuro VI", ano: 2090))

print("\nTotal de Filmes da Hexalogia BTTF : \
(bttf. count) filmes\n")

for filme in bttf {
   filme.dados()
   print("Tipo gerado: \(filme)")
}
```

Recebendo parâm. e retornando tupla

Vamos ao exemplo de uma função que receberá 3 parâmetros: valor, de e para e irá retornar uma tupla com o valor convertido da temperatura e qual o simbolo que representa o valor convertido:

```
    func  conversorTemperatura  (valor:  Double,  de:
String,
    para: String) -> (convertido: Double, temperaturaDe
:
    String, temperaturaPara: String) {
        var retorno: Double = 0.0

//Elaborando um dicionário com as temperaturas
var temperaturas: Dictionary<String,String> = [
“C”:”Celsius”, “F”:”Fahrenheit”, “K”:”Kelvin” ]

if de == “F” {
    if (para == “C”) {
        retorno = (valor - 32) / (5 / 9)
    }else if (para == “K”) {
        retorno = (valor - 32) * (5 / 9) + 273.15
    }
}
else if de == “C” {
    if (para == “F”) {
        retorno = valor * (9 / 5) + 32
    } else if (para == “K”) {
        retorno = valor +
    }
}
 else {
    if (para == “F”) {
        retorno = ((valor - 273.15) * 5 / 9) + 32
    }
    else if (para == “C”) {
        retorno = valor - 273
    }
}

  return (retorno, temperaturas[de]!, temperaturas[
   para]!)
}

//Capturando retorno da função através de tupla
var valor = 36.5
var de = “C” //Celsius
var para = “F” //Fahrenheit
```

```swift
//Exemplo 1
let (retornoValor, retornoDe, retornoPara) =
conversorTemperatura(valor, de, para)

print("A temperatura \(valor) em \(retornoDe) ficaria
\(retornoValor) em \(retornoPara)")


//Exemplo 2
let (a, b, c) = conversorTemperatura(100.0, "C",
"K")

print("A temperatura \(100.0) em \(b) ficaria \(a)
em\(c)")
```