APP WORLD

# VOE COM SWIFT

3B

## LISTA DE FIGURAS

## LISTA DE CÓDIGOS-FONTE

## SUMÁRIO

# 1 VOE COM SWIFT

Chegou a hora de conhecer a sintaxe de uma nova linguagem nativa para desenvolvimento mobile, depois de Kotlin agora aprenderá mais uma! *Swift* é a linguagem de desenvolvimento da Apple para todos os seus *devices*. Vamos trabalhar com essa nova linguagem de programação.

Além da sintaxe, você aprenderá todos os recursos que o *Swift* oferece ao desenvolvedor.

## 1.1 A linguagem SWIFT

### 1.1.1 Apresentação

Em 2014, em sua conferência anual para desenvolvedores (WWDC: Apple Worldwide Developers Conference), a Apple introduziu ao mundo sua mais nova linguagem de programação, a linguagem *Swift*.

Atualmente na versão 5.7, *Swift* é o resultado do que há de mais novo em pesquisas envolvendo linguagem de programação, combinado com décadas de experiência na construção dos softwares para as plataformas Apple. *Swift* entrega uma maneira incrível de desenvolver aplicativos, seja para iPhones/iPads/iPods, Apple TV, Macs, Apple Watch e até mesmo Servidores. Cem por cento orientada a objetos, que foi construída de modo a ser uma linguagem atual, segura, rápida e fácil de se trabalhar, com uma sintaxe limpa e clara e seguindo conceitos modernos de programação, como, por exemplo, a inferência dos tipos, gerenciamento automático de memória e a ausência da necessidade de utilização de ponto e vírgula, dentre outros, o que resulta em uma linguagem poderosa e divertida de se trabalhar.

### 1.1.2 Principais características

Vamos dar uma olhada nas principais características da Swift, o que a torna uma das linguagens de programação mais modernas da atualidade:

- **Inferência de Tipo:** Ao declarar uma variável, se vamos atribuir um valor a essa variável no momento da sua declaração, não precisamos definir explicitamente o seu tipo, pois a linguagem faz a inferência do tipo com base no valor atribuído, ou seja, interpreta o tipo do valor que está sendo atribuído àquela variável e o define automaticamente.
- **Ausência de ponto e vírgula:** Não é necessário o uso de ponto e vírgula para identificar o término de uma instrução em *Swift.* Isso é feito através da quebra de linha, portanto, novos comandos devem iniciar em uma nova linha. Apesar de não ser necessário, é possível utilizar ponto e vírgula em *Swift,* caso o desenvolvedor assim deseje.
- **Tipos fortes**: Uma vez definido o tipo de uma variável, este não poderá ser alterado ao longo do código.
- **Programação funcional**: Existem muitos métodos que se baseiam em padrões de programação funcional, como, por exemplo, map, filter e reduce.
- **Tuplas:** Estrutura dos dados muito útil e versátil. Bastante utilizada quando se deseja retornar múltiplos valores em um método, por exemplo.
- **Generics:** É possível trabalhar com tipos e métodos genéricos, ou seja, que aceitam qualquer tipo, sujeitos a requerimentos definidos pelo programador.
- **Closures:** Blocos autocontidos das funcionalidades que podem ser passados como parâmetros ou retornos de métodos.
- **Optionals**: Maneira segura de trabalhar com tipos que podem ser nulos.

## 1.2 Playground

### 1.2.1 Ambiente de estudos: Playground

Quando a Apple lançou a linguagem *Swift*, trouxe uma novidade muito interessante para os desenvolvedores: um ambiente de desenvolvimento do zero para estudo da linguagem. Desta vez, a IDE utilizada para desenvolvimento de aplicativos, o Xcode, veio com um recurso muito interessante e útil para que os programadores pudessem, de uma forma bem direta, treinar a linguagem *Swift* e aprender todos os seus recursos. Esse ambiente é chamado *Playground*, e é parte integrante do Xcode, a IDE de desenvolvimento de Apps e Software da Apple. Através dele, os desenvolvedores podem experimentar trechos de código sem a necessidade de criar um projeto de App para iPhone e rodar no simulador ou celular para testar seus códigos.

O *Playground* é um ambiente REPL (Read-Eval-Print-Loop), que fica constantemente (*loop*) lendo (*read*), avaliando ou interpretando (*eval*) o código e apresentando o resultado dessa interpretação (*print*). Com isso podemos, em tempo real e sem a necessidade de compilação, testar trechos de código *Swift.* É o ambiente perfeito para o aprendizado da linguagem.

Para que possamos utilizar o *Playground,* é preciso abrir o Xcode (que, lembrando, foi instalado no capítulo anterior), e na barra de menus do macOS, escolher a opção **File > New > Playground** ou a partir do atalho OPTION + SHIFT + COMMAND + N. Na tela seguinte, é apresentado um painel onde é possível escolher um template para o seu arquivo *Playground.* Quando desejamos produzir um arquivo com códigos prontos para determinados cenários, escolhemos um dos quatro templates que se encontram mais à baixo, ou seja, **Game** (para que o arquivo traga exemplo de código utilizando o SpriteKit, *engine* da Apple para desenvolvimento de jogos), **Map** (exemplo de uso de Mapas) e **Single View** (código com uma simples tela de um aplicativo). Como vamos utilizar o *Playground* somente para aprendizado da linguagem *Swift* em si, vamos escolher o template **Blank**, que é um template praticamente sem código, ideal para iniciarmos nossos estudos. Portanto, certifique-se de que a opção **iOS** esteja marcada na parte superior, escolha **Blank** e pressione o botão **Next**.

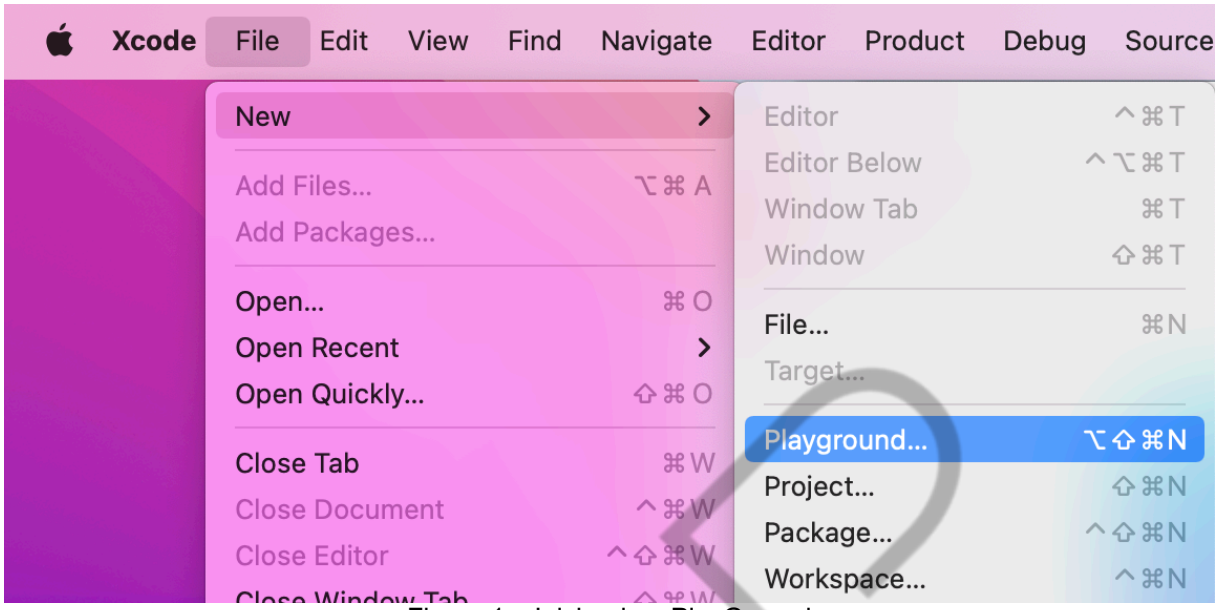


Figura 1 – Iniciando o PlayGround
Fonte: Elaborado pelo autor (2022)

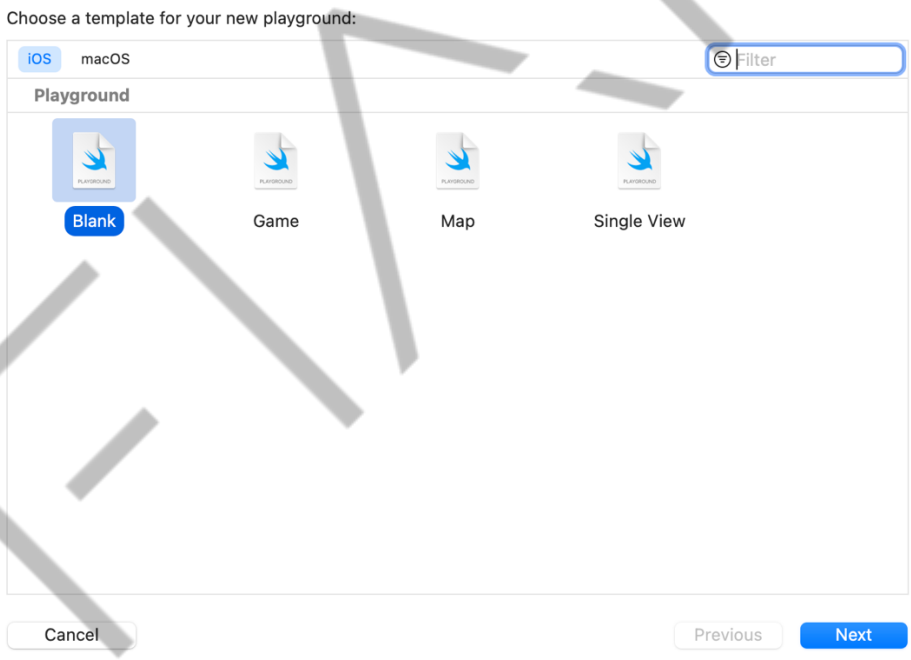


Figura 2 – Seleção de template
Fonte: Elaborado pelo autor (2022)

Na tela que se segue, escolha um nome para seu arquivo *Playground* na opção **Save As:**, selecione uma pasta onde deseja salvar o arquivo e clique em **Create**. Após ter feito isso, estará com o ambiente *Playground* aberto, pronto para receber seus códigos.

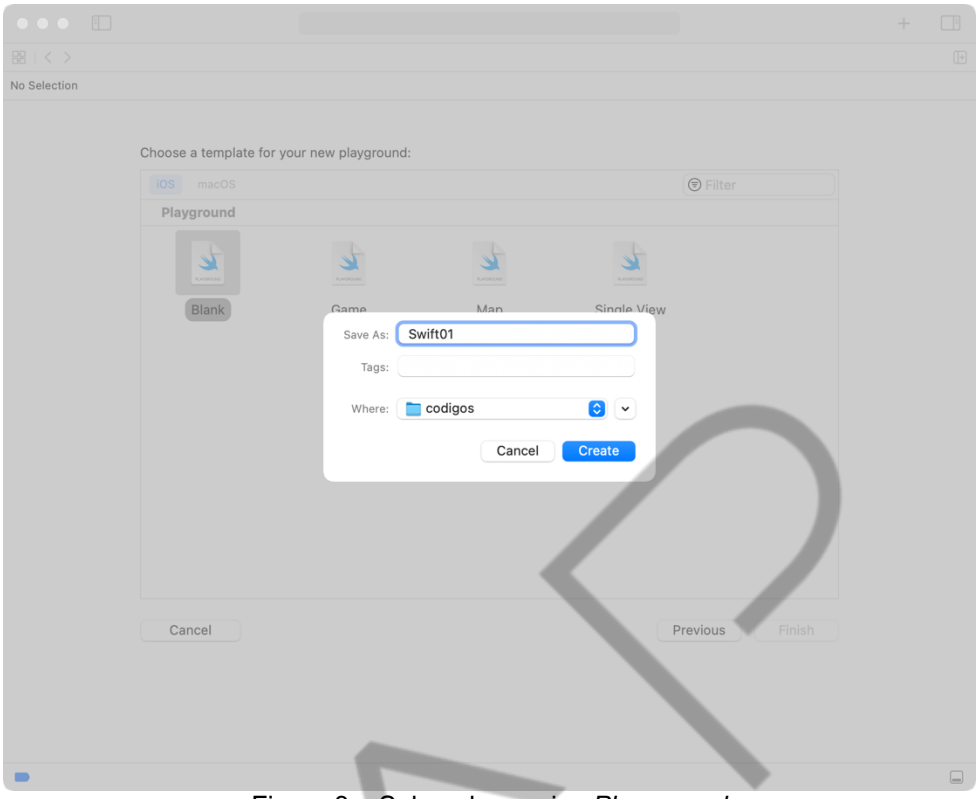


Figura 3 – Salvando arquivo *Playground*
Fonte: Elaborado pelo autor (2022)


Figura 4 – *Playground*
Fonte: Elaborado pelo autor (2022)

### 1.2.2 Conhecendo o *Playground*

O ambiente *Playground* é basicamente dividido em três áreas principais. Existem mais áreas e painéis, que fazem parte do Xcode como um todo, porém, em

se tratando de *Playground* e para o nosso propósito neste capítulo, que é o estudo da linguagem *Swift*, iremos focar em apenas três dessas áreas.

A parte central é onde iremos criar nossos códigos, ou seja, nossas variáveis, funções, classes etc. É aqui que você efetivamente programa seu código *Swift.*

Como o *Playground* é um ambiente REPL, estará constantemente lendo e avaliando seu código, e o resultado dessa análise é apresentado na área que se encontra ao lado direito. Você notou que, ao criarmos um template Blank, nosso arquivo *Playground* foi criado com duas simples linhas de código, que são um comando de importação de *framework* e a criação de uma simples variável chamada **greeting** que está recebendo a String "**Hello, playground**". Note que esse código é automaticamente avaliado e o seu resultado é apresentado no painel ao lado, ou seja, o conteúdo da variável **greeting** é mostrado no painel que fica do lado direito. Caso não esteja rodando automaticamente, você precisa configurar para que isso ocorra clicando e segurando na seta (azul apontando para baixo) ao lado do botão Run/Stop e depois escolha Automatically Run.

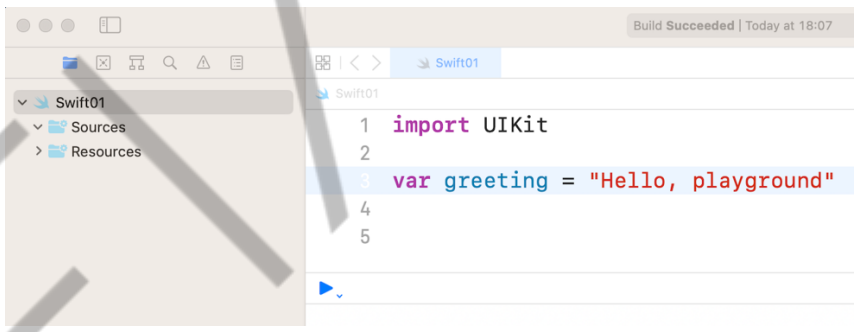


Figura 5 – Local para configurar Run automático
Fonte: Elaborado pelo autor (2023)

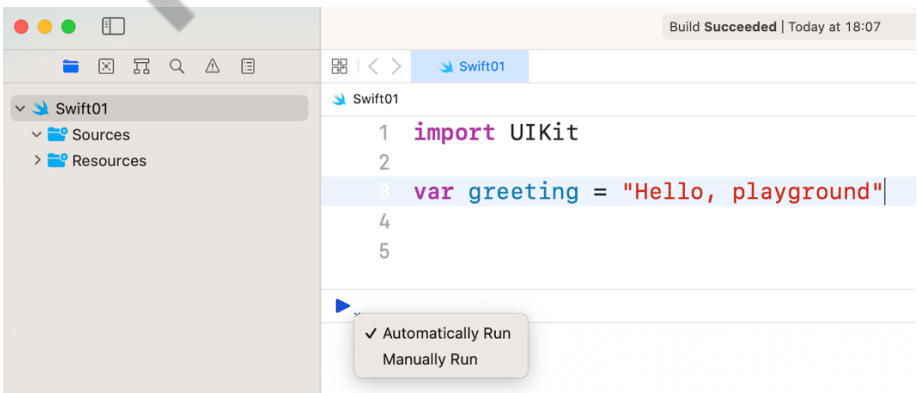


Figura 6 – Configurando Automatically Run
Fonte: Elaborado pelo autor (2022)

Qualquer código que você escreva será interpretado em tempo real e, caso possua algum retorno ou alimente alguma variável, esse valor será automaticamente apresentado na área que se encontra do lado direto do *Playground.*

Existe uma terceira área bem útil, chamada **Debug Area**, que é usada quando desejamos imprimir alguma mensagem definida por nós mesmos. Essa área é especialmente útil quando estamos depurando um aplicativo, pois em desenvolvimento de Apps, não temos um ambiente REPL como o *Playground,* e fazemos uso da **Debug Area** para imprimir o resultado de um método, o valor de uma variável ou qualquer mensagem que nos ajude a analisar o que está acontecendo com nosso App. Apesar de o *Playground* mostrar o resultado no painel à direita, é a **Debug Area** que nos informará sobre os possíveis erros que estejam ocorrendo com o código, além, é claro, de imprimir as mensagens que solicitamos. Essa área não vem aberta por padrão, porém é apresentada automaticamente sempre que utilizamos o comando **print**, usado para imprimir algum conteúdo na **Debug Area**.

Vamos fazer um pequeno exercício, digite logo abaixo da declaração da variável **greeting** o comando print(**greeting**). Com esse comando, estamos solicitando que o Xcode (lembre-se, o *Playground* é um ambiente interno do Xcode) imprima na **Debug Area** o conteúdo da variável **greeting**, que sabemos que é “Hello, *playground*”. Observe que, após alguns segundos, o *Playground* interpretará o seu código, abrirá a **Debug Area** e apresentará o resultado desse comando.

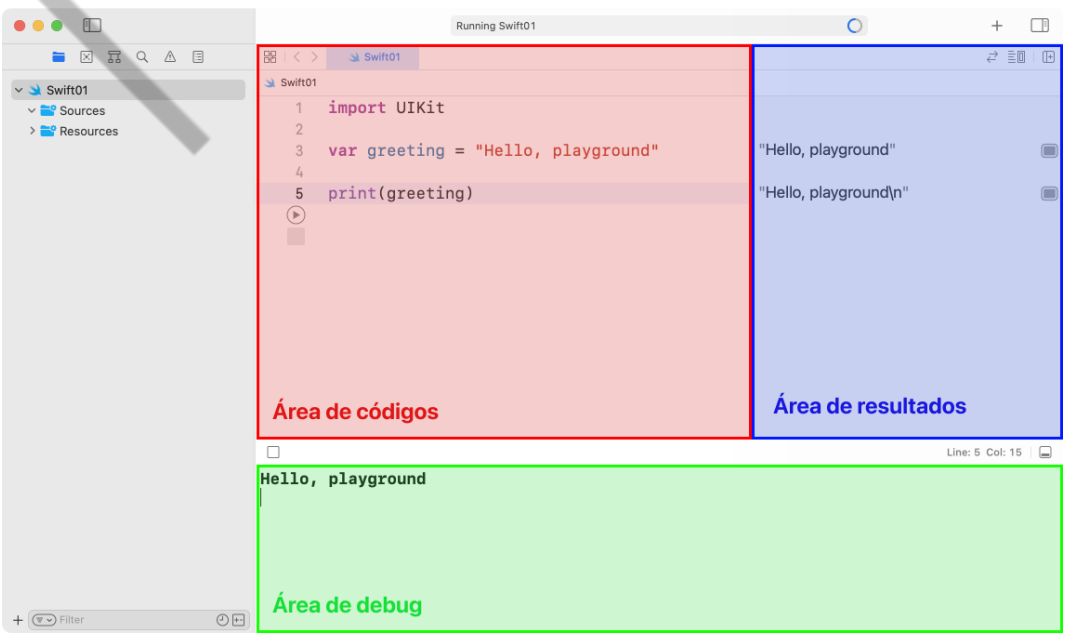

Figura 7 – Áreas do *Playground*
Fonte: Elaborado pelo autor (2022)

### 1.2.3 Interpretando códigos automaticamente/manualmente

Apesar do recurso de avaliação/interpretação e impressão automática do código ser muito útil, caso seu código comece a ficar muito grande ou execute operações complexas, isso pode se tornar um problema, pois a cada pequeno trecho de código escrito o *Playground* varre todo o seu código, desde o início, e começa o *loop* de interpretação e impressão, o que pode tornar o uso muito lento e, em alguns casos, até mesmo travar o Xcode. Uma dica que gostaria de deixar aqui é: evite deixar muito código no mesmo arquivo *Playground*, para que o processo de análise e impressão não fique lento. Faremos isso ao longo do curso, ou seja, frequentemente sugeriremos que você produza um novo *Playground* para o aprendizado dos novos comandos. Caso seu código seja enorme, mas precise estar presente no mesmo arquivo, uma técnica para tornar o desenvolvimento mais rápido é desabilitar a opção de análise automática. Isso é feito clicando novamente no mesmo local apresentado anteriormente e segurando o botão em forma de seta azul que se encontra acima da **Debug Area** e, ao abrir o menu, selecionar a opção **Manually Run**. Note que a seta passa a fica transparente, o que indica que o Xcode interpretará seu código caso clique nessa seta. Vamos deixar essa opção na forma como está, que é **Automatically Run**, para que possamos ver em tempo real o resultado dos nossos códigos.

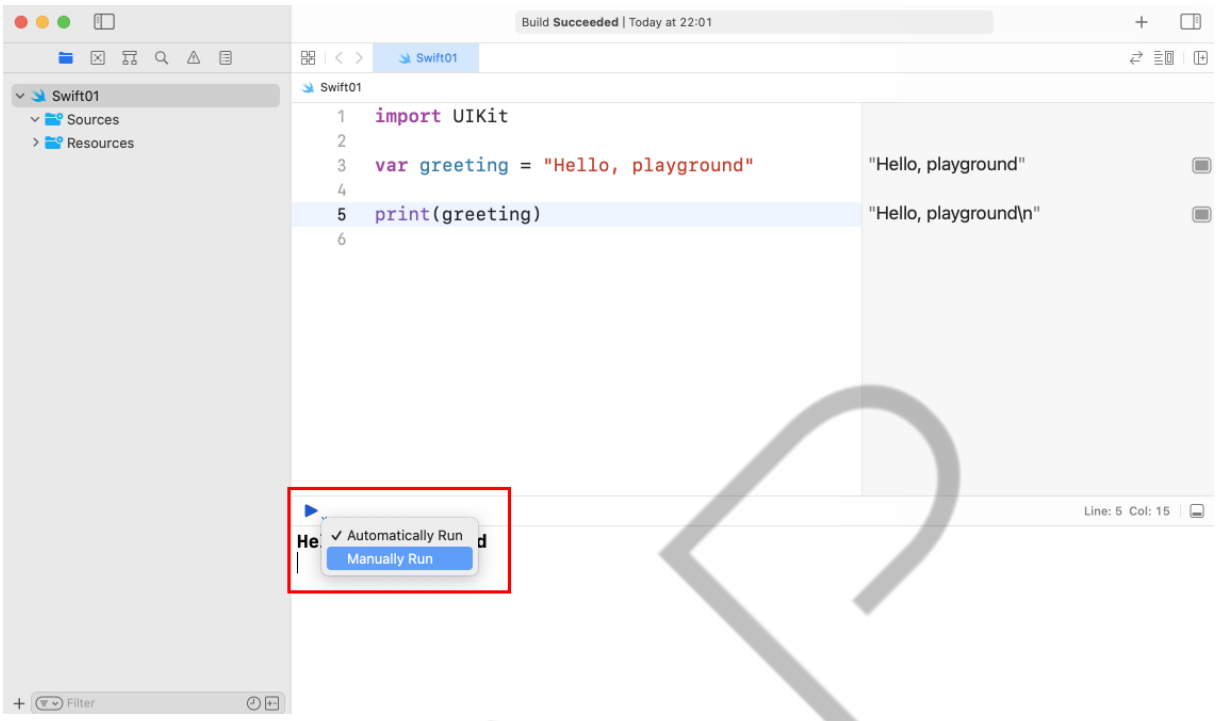


Figura 8 – Automatically/Manually Run
Fonte: Elaborado pelo autor (2022)

## 1.3 Comentários e variáveis

### 1.3.1 Comentários

Toda linguagem de programação possui um recurso que permite ao desenvolvedor deixar um comentário, seja para documentar um trecho de código ou para lembrá-lo de revisar algo que tenha feito. Também usamos comentários quando desejamos que certo trecho do nosso código não seja interpretado, porém, ao mesmo tempo, não queremos excluí-lo, pois podemos precisar desse código no futuro. Em *Swift*, existem dois tipos de comentários: os comentários de uma linha e os comentários de múltiplas linhas. Para criarmos um comentário de uma linha, usamos os caracteres *//*, e para comentários de múltiplas linhas, iniciamos o comentário com */** e finalizamos com ***/*.

Apague todo o conteúdo do seu arquivo *Playground* e insira os códigos abaixo.

```
var age1 = 20
var age2 = 30


///Sum é a soma da age1(primeira idade) e age2(segunda idade)
var sum = age1 + age2
age2 = 50

//Esta linha e a linha abaixo não serão executadas
//sum += age2

/*
//Todo esse bloco de código não será executado
var age3 = 15
sum = age2 + age3

 /*
 //Aqui temos um comentário de múltiplas linhas dentro
 //de outro comentário de múltiplas linhas
 age3 = 35
 sum += age3
 */

print(sum)
sum += 20
*/

print(sum)
```

Código-fonte 1 – Comentários
Fonte: Elaborado pelo autor (2019)

Neste caso, o resultado impresso será 50, que é a soma de age1 (20) com age2 (30). Note que nenhuma das linhas que se encontram comentadas será executada.

Uma particularidade da linguagem *Swift* com relação aos comentários é que ela permite que o desenvolvedor insira um comentário de múltiplas linhas dentro de outro comentário de múltiplas linhas, sem indicar erro. Boa parte das linguagens de programação não saberia interpretar esse tipo de código e, ao encontrar o fim do comentário interno, iria tratar a sequência como um erro.

Também podemos documentar uma propriedade ou um método com comentários, para isso colocaremos os caracteres ///. Feito isso podemos abrir o *Quick Help* com a tecla de atalho OPTION + CLICK na propriedade sum e ver o sumário

dessa variável. Note que logo após a criação das variáveis age1 e age1 foi utilizado \\\ para antes do cálculo para explicar o conteúdo da variável sum.

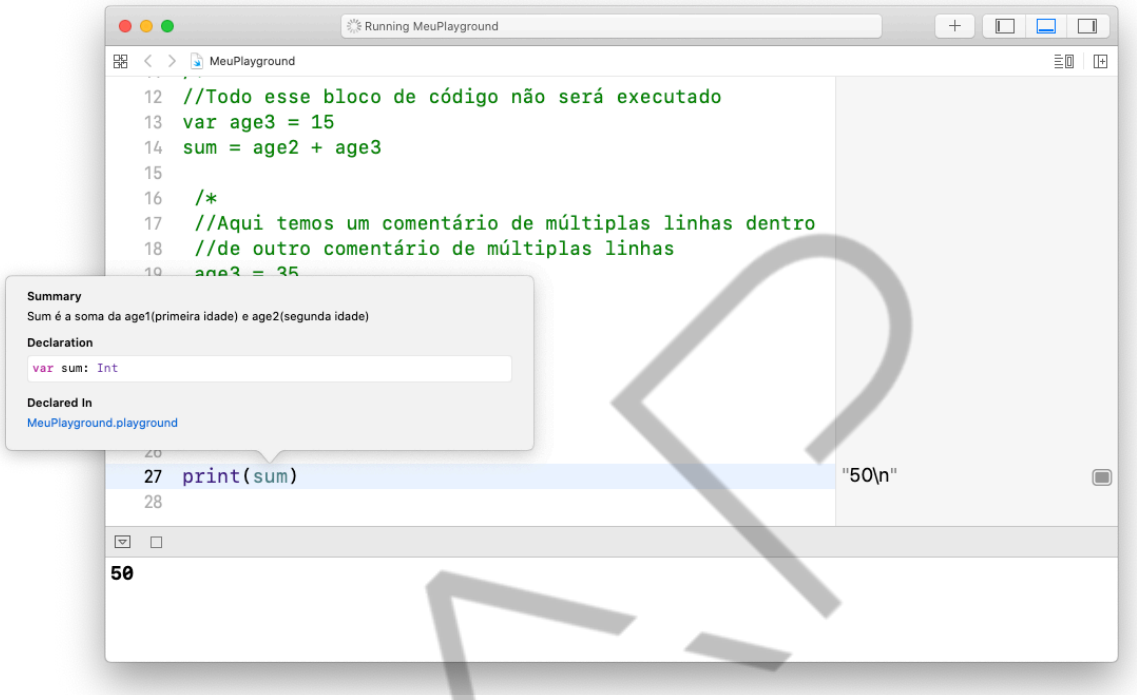


Figura 9 – *Quick Help* em uma variável
Fonte: Elaborado pelo autor (2020)

### 1.3.2 Variáveis e constantes

A criação de variáveis em *Swift* é feita através do uso da palavra reservada **var**, que indica o início da declaração de um objeto cujo valor pode mudar ao longo do código (objeto mutável, ou variável). Para criarmos uma variável, iniciamos com a palavra reservada **var**, seguida do nome da variável, dois pontos, o tipo da mesma e a atribuição do valor (ou inicialização da variável).

Uma das características da linguagem *Swift* é que ela trabalha com o que chamamos de **Inferência de Tipo**, ou seja, não é obrigatória a definição do tipo de uma variável, caso inicialize a mesma durante a sua declaração. Vejamos o código abaixo:

```
//Criando variável com tipo explícito
var name : String = "Steve Jobs"

//Criando variável utilizando inferência de tipo
var name2 = "Steve Wosniak"
```

Código-fonte 2 – Variáveis
Fonte: Elaborado pelo autor (2019)

Não há diferença alguma para a linguagem se criarmos uma variável como foi feito para name ou da maneira usada para name2. Na primeira forma, definimos explicitamente que o tipo da variável é *String*, e na segunda maneira a própria linguagem atribuiu *String* à variável, com base no conteúdo atribuído a ela (que é um conteúdo do tipo *String*).

Um detalhe que devemos atentar em *Swift* é que toda e qualquer variável precisa ter um valor atribuído antes de sua utilização, seja atribuído na sua própria definição ou então através do construtor da classe onde a variável se encontra. Falaremos mais sobre isso quando entrarmos no assunto Classes e Estruturas, porém, até lá, todas as variáveis que criaremos aqui sempre estarão com um valor atribuído.

Variáveis são criadas começando com caracteres minúsculos e, caso sejam compostas de mais de uma palavra, usamos lowerCamelCase, ou seja, a primeira palavra se inicia em letra minúscula, porém cada nova palavra se inicia em maiúscula.

Antes de prosseguirmos, um detalhe sobre nosso conteúdo: todo código-fonte criado aqui usará nomes de variáveis, métodos, classe etc. em **inglês**. Utilizamos essa abordagem, pois no mercado de trabalho a grande maioria dos desenvolvedores programam utilizando códigos escritos em inglês, além de ser uma linguagem universal em programação, fica fácil de trabalhar com times de qualquer lugar do mundo, algo bem comum nos dias de hoje, principalmente em empresas multinacionais. Mas não se preocupe, cada trecho de código criado será explicado de modo que, caso não entenda muito o **inglês**, você consiga entender o que será criado.

Continuando, digamos que desejamos produzir uma variável que será usada para armazenar o primeiro nome de uma pessoa, e outra para armazenar o último nome. A maneira correta para criar essas variáveis seria:

```
//Usando lowerCamelCase. Cada nova palavra, começa em
maiúscula
var firstName = "John"
var lastName = "Appleseed"
```

Código-fonte 3 – Utilizando lowerCamelCase
Fonte: Elaborado pelo autor (2019)

Variáveis não podem começar com números e não podem conter espaço entre as palavras, porém *Swift é* uma das poucas, senão a única, linguagem a oferecer a possibilidade de trabalhar com emojis como nome de variáveis. Obviamente não faremos isso em um projeto real, pois não é algo intuitivo, porém a Apple disponibilizou essa *feature* na linguagem com o intuito de trazer pessoas cada vez mais novas para o mundo da programação, inclusive crianças, e nada melhor do que permitir o uso de emojis para tornar o aprendizado mais divertido. Então, em *Swift,* é perfeitamente possível fazer algo como:

```
var 🐶 = "Billy"
var 😈 = "capetinha"
var 💩 = "cocô"
print("O meu cachorro se chama", 🐶, "e é um verdadeiro", 😈,
"pois faz", 💩, "na casa toda")
```

Código-fonte 4 – Utilizando emojis
Fonte: Elaborado pelo autor (2019)

**DICA:** Para utilizar um emoji, aperte as teclas Command + Control + Barra de Espaço que o painel de emojis aparecerá e você poderá escolher qual emoji utilizar. Só não são válidos os emojis que, em seu código interno, começam com um número.

**DICA:** Quando usamos o comando print, podemos separar o conteúdo do que será impresso usando vírgulas. Nesse caso, imprime o conteúdo separando cada variável por espaço.

Também é possível produzir diversas variáveis na mesma linha, separando por vírgula cada variável, porém utilizando **var** somente uma vez.

```
//Criando 3 variáveis em uma única linha
var x = 12, y = "Oi", z = 3.3
```

Código-fonte 5 – Múltiplas variáveis na mesma linha
Fonte: Elaborado pelo autor (2019)

Para criarmos constantes, ou seja, objetos que não modificam seu valor ao longo do código (objetos imutáveis), a sintaxe é quase idêntica a de produção de variáveis, porém com o uso da palavra reservada **let** no lugar de **var**.

```
//Criando constantes
let pi = 3.141592
let earthGravity = 9.81

//Note que não é possível alterar o valor de uma constante
earthGravity = 10.01  //ERRO
```

Código-fonte 6 – Criando constantes
Fonte: Elaborado pelo autor (2019)

**DICA:** Sempre procure utilizar constantes em seu código, a menos que realmente precise modificar o objeto futuramente. O acesso às constantes na memória é mais rápido do que o acesso às variáveis. Durante o processo de programação dentro do Xcode, para todas as variáveis encontradas no seu código que não tiverem seu conteúdo alterado, será sugerida a mudança da declaração para let.

### 1.4 Tipos

Existem certos tipos predefinidos em *Swift* que são utilizados com muita frequência. Algumas linguagens costumam chamar esses tipos de tipos primitivos, porém essa nomenclatura não é usada em *Swift.* Dentre essa lista de tipos (que na verdade são Structs, assunto que será abordado mais para frente), nós temos tipos para trabalhar com números inteiros, números com casas decimais, valores booleanos (que aceitam verdadeiro ou falso), entre outros. Vamos listar abaixo os principais.

### 1.4.1 Int (Inteiros)

Quando precisamos representar em *Swift* um número que não possui casas decimais, utilizamos o tipo **Int** (inteiro). Esse tipo utiliza 64 bits de memória para ser representado, ou seja, pode variar de -9223372036854775808 a 9223372036854775807.

Este é o tipo que *Swift* infere quando atribuímos um valor inteiro a uma variável. Existem variações do tipo Int, que são Int8 (8 Bits), Int16 (16 Bits), Int32 (32 Bits) e Int64 (64 Bits), porém o mais utilizado é o **Int.**

Em casos em que o valor que você deseja utilizar não aceite números negativos, como, por exemplo, a idade de uma pessoa, pode-se utilizar a variante **UInt**, que vem de Unsigned Int, ou seja, inteiro sem sinal. Nesse caso, essa variável somente aceitaria números positivos. Da mesma forma que o Int, o UInt também possui variações (UInt8, UInt16, UInt32 e UInt64).

```
var value1 = 500          //Aqui, a inferência é para Int
var value2 : Int = 500    //Idêntico à linha acima
var myAge : UInt8 = 39    //Somente valores positivos

//Forma de mostrar o valor máximo aceito pelo tipo
//Válido para todos os tipos de Int (Int8, Int16, Int32,
UInt8, etc)
print(Int.max)

//Forma de mostrar o valor mínimo aceito pelo tipo
print(Int.min)
```

Código-fonte 7 – Utilizando Inteiros
Fonte: Elaborado pelo autor (2019)

### 1.4.2 Double e Float (números com casas decimais)

Para representar números que possuem casas decimais, utilizamos **Double** (que é o tipo inferido automaticamente quando atribuímos um número com casas decimais a uma variável) e **Float**. A diferença entre **Double** e **Float** é que **Double**

ocupa 64 bits na memória, podendo trabalhar com números maiores, e **Float** utiliza 32 bits de memória.

```
var balance = 1500.75 //Double inferido automaticamente
var sallary : Double = 1200.50 //Double explícito

//Para usarmos Float, precisamos explicitar o tipo
var temperature : Float = 35.9
```

Código-fonte 8 – Utilizando Inteiros
Fonte: Elaborado pelo autor (2019)

### 1.4.3 String e Character

Blocos de texto são representados em *Swift* pelo tipo String, que é definido por um ou mais caracteres entre aspas. Este é o tipo que é inferido quando atribuímos texto a uma variável. É possível utilizar o tipo Character quando necessitamos ocupar o espaço de apenas um único caractere, porém vale ressaltar que, para atribuir um Character a uma variável, é necessário definir de forma explícita o tipo Character na variável, pois do contrário ela será String, mesmo que o texto contenha apenas 1 caractere.

```
var module: String = "Introdução ao Swift"
var schoolName = "FIAP"

//Note que letter, na linha abaixo, é uma String
//Devido à inferência de tipo
var letter = "A"

//Para usarmos Character, precisamos definir explicitamente
var gender : Character = "M"
```

Código-fonte 9 – String e Character
Fonte: Elaborado pelo autor (2019)

Quando precisamos inserir em uma String algum caractere reservado pela linguagem (por exemplo, o caractere “que define o início/fim de uma String), usamos o caractere \. Através dele, podemos utilizar caracteres reservados, bem como outros

conjuntos especiais, como, por exemplo, o carriage return (simulando a tecla ENTER), uma tabulação, entre outros.

```
var text = "Este texto \"será quebrado\" em \nduas linhas"
//Resultado:
//Este texto "será quebrado" em
//duas linhas

//O \t gera uma tabulação
var text2 = "Nota:\t 10"
//Resultado:
//Nota:     10
```

Código-fonte 10 – Caractere especial \
Fonte: Elaborado pelo autor (2019)

Em *Swift,* é possível criar Strings utilizando uma técnica chamada Interpolação de Strings. É muito comum precisarmos atribuir a uma String a combinação de muitas variáveis, inclusive, com tipos diferentes de dados. Por exemplo, supomos que tenhamos uma variável que represente a nota do aluno (studentGrade), o nome (studentName) e o resultado de sua avaliação (aprovado ou reprovado). Podemos criar uma variável chamada message que conterá uma String informando todos os dados dos alunos e a sua avaliação. Por exemplo: “O aluno João tirou 8.5 e está aprovado”. Para criarmos essa String, precisamos juntar informações de três variáveis e, para isso, utilizamos a técnica de interpolação.

Isso é feito colocando, na String principal, as variáveis dentro de parênteses, precedidos pelo caractere \. Veja no exemplo abaixo:

```
let studentGrade = 8.5
let studentName = "João"
let result = "aprovado"

let message = "O aluno \(studentName) tirou \(studentGrade) e
está \(result)"
print(message)
//Resultado:
//O aluno João tirou 8.5 e está aprovado
```

Código-fonte 11 – Interpolação de Strings
Fonte: Elaborado pelo autor (2019)

Uma outra forma de trabalhar com Strings é utilizando o formato Multiline String Literal, em que podemos criar uma String com múltiplas linhas e a tabulação necessária para a sua exibição, sem a necessidade de caracteres de escape como \t ou \n para quebrar linhas ou criar tabulações.

```
let nationalAnthem = """
Ouviram do Ipiranga as margens plácidas
De um povo heroico o brado retumbante.

   Joaquim Osório Duque-Estrada
"""
//O resultado impresso será idêntico a String criada.
```

Código-fonte 12 – Strings multilinhas
Fonte: Elaborado pelo autor (2019)

### 1.4.4 Bool (Booleanos)

Booleanos são tipos simples, que ocupam apenas 1 bit de memória e aceitam apenas 2 estados, 0 ou 1. Em *Swift,* booleanos são definidos pelo tipo Bool e aceitam true (verdadeiro) ou false (falso).

```
var isApproved = true //Boolean inferido automaticamente
var firstTime : Bool = false  //Boolean explícito
```

Código-fonte 13 – Booleanos
Fonte: Elaborado pelo autor (2019)

### 1.4.5 Tuple (Tupla) – Criando e acessando elementos

Existe em *Swift* um tipo muito útil quando precisamos, em determinados casos, compor dois ou mais tipos em um só. Um dos casos mais utilizados é quando, por exemplo, precisamos definir que uma função precisa retornar mais de um tipo ao mesmo tempo. Estamos falando da **tupla**, que é um tipo composto formado por dois ou mais tipos distintos.

Vamos ao exemplo abaixo, no qual criamos uma variável String que contém o nome da rua, o número e o CEP de um endereço.

```
let address = "Av. Paulista, 1106, 01311-000"
```

Código-fonte 14 – Endereço em String
Fonte: Elaborado pelo autor (2019)

Seria muito complicado imprimir para o usuário somente o nome da rua, ou somente o endereço, por exemplo, afinal de contas toda a informação está contida dentro de uma única String. É nesse caso que entra em cena a **tupla**. Como podemos fazer uso da **tupla** para nos auxiliar?

Antes vamos analisar a sintaxe de criação de uma **tupla.** O tipo de uma **tupla** é definido pela utilização de parênteses e, dentro dos parênteses, separamos por vírgulas todos os tipos que estarão compostos dentro da mesma. No nosso exemplo, temos o nome da rua, o número e o CEP, que podemos representar por uma String (para o nome da rua), um **Int** (para o número) e outra String (para o CEP). Esse tipo (essa **tupla**) seria representado da seguinte forma: (String, Int, String).

Ou seja, para criarmos nossa **tupla**, trocamos o código acima pelo seguinte:

```
let address: (String, Int, String) = ("Av. Paulista", 1106,
"01311-000")
print("""
  Logradouro: \(address.0)
  Número: \(address.1)
  CEP: \(address.2)
""")
```

Código-fonte 15 – Tupla
Fonte: Elaborado pelo autor (2019)

Note que alimentamos a **tupla** com os respectivos valores associados a cada tipo, também separados por vírgulas e entre parênteses.

Para acessarmos um elemento da **tupla,** utilizamos o índice desses elementos, sendo que o primeiro se encontra no índice 0, o segundo no índice 1 e assim sucessivamente, ou seja, se precisarmos acessar o número do endereço, utilizaremos

**address.1**, pois o número do endereço é o segundo componente da nossa **tupla** (índice 1).

Uma forma ainda mais interessante de utilização de **tupla** é criar uma nomeada, ou seja, definirmos nomes para cada um dos elementos da **tupla.** Dessa forma, além de ser possível acessar cada um dos elementos por meio do seu índice, também é possível acessá-los através de seu nome, o que torna muito mais prático e intuitivo o seu uso.

Altere o código para que fique desta forma:

```
let address: (street: String, number: Int, zipCode: String) =
("Av. Paulista", 1106, "01311-000")
print("""
  Logradouro \(address.street)
  Número: \(address.number)
  CEP: \(address.zipCode)
""")
```

Código-fonte 16 – Tuplas nomeadas
Fonte: Elaborado pelo autor (2019)

Note que cada um dos elementos agora tem um nome associado, que definimos respectivamente de street (rua), number (número) e zipCode (CEP). O acesso através de índice continua valendo, porém fica muito mais prático utilizar pelo nome, certo?

Para finalizar, é muito comum precisarmos decompor a **tupla** em outras variáveis. Imagine que queira criar uma variável chamada **streetName**, que conterá o nome da rua, e uma variável **streetNumber**, que deverá conter o número do endereço. Você poderia produzir essas duas variáveis da seguinte maneira.

```
let address: (street: String, number: Int, zipCode: String) =
("Av. Paulista", 1106, "01311-000")

let streetName = address.street
let streetNumber = address.number
```

Código-fonte 17 – Atribuindo tupla a variáveis
Fonte: Elaborado pelo autor (2019)

Porém, existe uma maneira mais simples e direta de fazermos isso. Essa técnica é chamada Decomposição de **Tupla** e consiste em criar variáveis e atribuir os elementos da **tupla** em uma única instrução. Isso é feito da seguinte maneira:

```
let address: (street: String, number: Int, zipCode: String) =
("Av. Paulista", 1106, "01311-000")

let (streetName, streetNumber, _) = address
print("O número da rua é \(streetNumber)")
```

Código-fonte 18 – Decomposição de Tupla
Fonte: Elaborado pelo autor (2019)

Note que a linguagem sabe que variável atribui cada valor da **tupla** com base na sua posição. Dessa forma, a primeira variável (streetName) receberá o primeiro elemento da **tupla** e a segunda variável (streetNumber) receberá o segundo elemento da **tupla**. Como não queremos produzir uma terceira variável, utilizamos o caractere _, que é usado quando desejamos indicar à linguagem que não queremos elaborar uma variável para aquele caso. É importante ressaltar que sempre se deve utilizar a mesma quantidade dos elementos que a **tupla** contém, usando o _ nos casos em que não deseja que a variável seja criada. Por exemplo, se quiséssemos desenvolver uma variável para o nome da rua e outra para o CEP, ficaria desta forma:

```
let address: (street: String, number: Int, zipCode: String) =
("Av. Paulista", 1106, "01311-000")

let (streetName, _, zipCode) = address
print("O CEP é \(zipCode)")
```

Código-fonte 19 – Decomposição de Tupla
Fonte: Elaborado pelo autor (2018)

### 1.4.6 Optionals

Um dos tipos mais utilizados em *Swift* é o Optional, que permite que determinada variável possa ser nula (ou seja, não contenha valor) ou contenha um valor encapsulado (embrulhado, ou *wrapped*).

Em *Swift*, toda variável precisa ter algum valor atribuído no momento do seu uso. Isso é feito seja atribuindo um valor no momento da definição da variável, ou através do construtor da classe em que essa variável foi definida. Porém, é comum termos cenários nos quais nós não temos o valor de uma variável no momento de sua criação. Imagine o caso de uma classe criada para representar uma pessoa, e dentre as propriedades de uma pessoa está a sua carteira de motorista. Você não concorda que nem todo mundo possui carteira de motorista? Porém, todos podemos tirar algum dia.

Portanto, temos um cenário no qual essa variável (que iremos chamar de driverLicense) pode perfeitamente não possuir um valor no momento de construirmos o objeto que representará essa pessoa. Assumindo que essa variável seja do tipo String, como poderemos defini-la? Se formos definir como var driverLicense : String, estaremos criando uma variável do tipo String e a linguagem nos obrigará a alimentá-la no construtor ou na sua definição.

Para que seja possível alcançarmos o que desejamos, essa variável precisa ser uma String Optional, e definimos isso colocando o sinal **?** após o nome do tipo, ou seja, nossa variável ficará assim (apague todo o código do seu arquivo ou crie um novo para continuar, apenas para evitar que o Xcode fique lento):

```
var driverLicense : String?
print(driverLicense)
//Resultado: nil
```

Código-fonte 20 – Criando Optional
Fonte: Elaborado pelo autor (2019)

Dessa maneira, estamos dizendo à linguagem que desejamos desenvolver uma variável que será um Optional de String, ou seja, essa variável pode ou não conter uma String. No exemplo acima, como não atribuímos um valor à variável, seu conteúdo é nulo, que em Swift é definido pela palavra nil (not in list). Perceba que é este o resultado que surge quando tentamos imprimir o conteúdo da variável.

Agora mude o código para o seguinte:

```
var driverLicense : String?
driverLicense = "6789877"
print(driverLicense)
//Resultado: Optional("6789877")
```

Código-fonte 21 – Criando Optional
Fonte: Elaborado pelo autor (2019)

Você notou que o resultado não foi 6789877 e sim Optional (“6789877”)? Isso acontece porque, ao criarmos um Optional, estamos criando na verdade um tipo que, quando não possui valor, retorna **nil**, porém, quando possui, empacota o valor dentro de um container. Para que possamos extrair esse valor, esse conteúdo, devemos **desembrulhar** (fazer o *unwrap*) esse Optional.

Existem diversas formas de desembrulhar um Optional, algumas seguras, outras nem tanto. Vamos começar pela mais simples, porém menos segura, delas. Insira o sinal ! logo após o nome da sua variável no comando print, deixando o seu código da seguinte forma.

```
var driverLicense : String?
driverLicense = "6789877"
print(driverLicense!) //Desempacotando
//Resultado: 6789877
```

Código-fonte 22 – Desembrulhando Optional de forma insegura
Fonte: Elaborado pelo autor (2019)

O caractere !, quando utilizado após uma variável Optional, faz o *unwrap* (desembrulhar) desse Optional, devolvendo o valor. Foi exatamente isso que aconteceu no código acima, porém, como mencionei anteriormente, essa forma não é segura, pois, ao utilizá-la, você precisa ter certeza de que existe algum valor nessa variável. Caso não haja nenhum valor, desembrulhar um Optional dessa forma fará com que seu código dê erro e, quando isso acontecer, seu aplicativo vai simplesmente fechar nas mãos do usuário. Experimente comentar a linha onde atribuímos o valor à variável e veja o que acontece. (Obs.: O *Playground* não fechará, porém, se esse código estivesse rodando em um app, aconteceria um *crash*, ou seja, um erro fatal e seu app seria encerrado).

```
var driverLicense : String?
//driverLicense = "6789877"
print(driverLicense!)
//Resultado: Fatal error: Unexpectedly found nil while
//unwrapping an Optional value
```

Código-fonte 23 – Desembrulhando Optional de forma insegura
Fonte: Elaborado pelo autor (2019)

Veja que o *Playground* informa, na **Debug Area**, ocorreu um erro fatal, pois foi encontrado **nil** ao desembrulhar um Optional. Se fosse no mundo real, isso seria o mesmo que entrar em um elevador sem que ele estivesse lá!

E como desembrulhar um Optional de maneira segura? Uma das técnicas mais utilizadas é chamada de Optional Binding e consiste em tentar desembrulhar o Optional atribuindo-o a uma nova variável, que será criada somente caso a operação tenha sucesso, ou seja, exista valor naquele Optional. Isso é feito da seguinte forma (substitua todo o código anterior pelo código abaixo):

```
var driverLicense : String?
//driverLicense = "6789877"
if let license = driverLicense {
    print("A carteira de motorista é \(license)")
} else {
    print("Esta pessoa não possui carteira de motorista")
}
//Resultado: Esta pessoa não possui carteira de motorista
```

Código-fonte 24 – Optional Binding
Fonte: Elaborado pelo autor (2019)

Note que o comando **if let license = driverLicense** na verdade tenta desembrulhar o Optional driverLicense e atribuí-lo à variável license. Se isso der certo, essa variável será criada e poderá ser utilizada dentro do primeiro bloco, delimitado pelo primeiro conjunto de { }. Caso contrário, o código pula para o bloco do **else** e **executa** o comando definido lá. O uso do **else** é opcional, ou seja, não é obrigatório implementá-lo, se você não quiser tomar alguma atitude quando o Optional for nulo. Experimente descomentar a linha onde atribuímos o valor à variável driverLicense e

veja que o código agora cairá no primeiro bloco, imprimindo a mensagem A carteira de motorista é 6789877.

Vale ressaltar que a variável license criada somente existirá dentro do bloco, ou seja, se à frente no código você precisar imprimir novamente o valor da carteira de motorista, deverá efetuar um novo Optional Binding.

Outra forma segura de desembrulhar uma variável opcional é o comando **guard**, assim como o **if let,** o **guard** cria uma variável se o valor opcional for nulo, para que possamos usar de maneira segura, sem acontecer um crash no seu programa.

Como vantagem ao if let, o guard pode ser usado em todo o escopo que ele foi criado. No código abaixo, vemos que podemos chamar a variável *license* mesmo fora do bloco {}.

```
func check(driverLicense: String?) {
    guard let license = driverLicense else {
        print("Esta pessoa não possui carteira de motorista")
        return
    }
    print("A carteira de motorista é \(license)")
}

//check(driverLicense: nil)
//Resultado: Esta pessoa não possui carteira de motorista

check(driverLicense: "6789877")
//Resultado: A carteira de motorista é 6789877
```

Código-fonte 25 – Optional Binding
Fonte: Elaborado pelo autor (2020)

Também é possível criar um Optional que nasce implicitamente desembrulhado, ou seja, pode conter nulo, porém, se estiver com algum valor, este valor poderá ser acessado sem a necessidade de se utilizar Optional Binding. O nome dessa técnica é **Implicitly Unwrapped Optional**, porém, deve-se tomar muito cuidado ao utilizá-la, pois caso esqueça de alimentar a variável com um valor e chamar algum método desse Optional, seu App quebrará. Implemente o código abaixo:

```
var alias : String!
//a linha abaixo imprime Meu apelido é nil
print("Meu apelido é \(alias)")
alias = "Pelé"
//Imprime Meu apelido é PELÉ
print("Meu apelido é \(alias.uppercased())")
```

Código-fonte 26 – Implicitly Unwrapped Optional
Fonte: Elaborado pelo autor (2019)

Essa técnica é tão perigosa quanto desembrulhar com !, ou seja, evite utilizá-la, ok? Comente a linha em que o apelido é atribuído à variável e veja que seu código indicará erro fatal.

Uma última forma de desembrulhar Optionals é utilizando um operador chamado **Nil-Coalescing Operator** (operador de coalescência nula). Esse operador é binário (ou seja, opera em 2 operandos) e, ao utilizá-lo, colocamos do lado esquerdo uma variável Optional e do lado direito o valor que desejamos utilizar, caso a variável seja nula. Dessa forma, se o Optional possuir um valor, será desembrulhado, do contrário, será utilizado o valor padrão fornecido.

Para exemplificar, vamos utilizar uma variável Optional de Int (Int?) que conterá a idade de uma pessoa e atribuiremos essa idade a uma outra variável utilizando o operador de coalescência nula. Veja que, no exemplo abaixo, a variável age2 recebe o valor 0, pois nesse momento a variável age não possui valor, porém a variável age3 recebe 27, que foi o valor atribuído à age imediatamente antes.

```
var age: Int?
let age2 = age ?? 0
print(age2) //0
age = 27
let age3 = age ?? 0
print(age3) //27
```

Código-fonte 27 – Nil-coalescing operator
Fonte: Elaborado pelo autor (2018)

Quando usamos Optionals e desejamos executar algum método dessa variável, a maneira mais segura de fazê-lo é utilizando a técnica **Optional Chaining**, que consiste em utilizar ? após o Optional, antes de chamar o método. O próprio Xcode

faz isso por você, se você tentar chamar um método de um Optional. Dessa maneira, conseguimos executar alguma operação naquela variável, caso possua valor, e o código não é executado, caso seja nula, sem acarretar nenhum tipo de erro. Veja que, no código abaixo, no primeiro print não acontece um erro, apenas imprime nil, enquanto no segundo, executa a operação (o método uppercased() retorna uma String em caixa-alta). Ele ainda retorna um Optional, porém podemos fazer um Optional Binding para recuperar o valor. O importante é notar que não quebramos o App na primeira execução, quando weekDay era nulo.

```
var weekDay: String?
print(weekDay?.uppercased())   //nil
weekDay = "Segunda"
print(weekDay?.uppercased())   //Optional("Segunda")
```

Código-fonte 28 – Optional Chaining
Fonte: Elaborado pelo autor (2019)

## 1.5 Coleções

Toda linguagem de programação contém uma forma de agregarmos muitas variáveis em uma só, produzindo uma coleção de tipos. As principais coleções em *Swift* são Arrays, Dictionaries e Sets.

### 1.5.1 Array (Matriz)

Uma das principais coleções em *Swift* é o **Array**, uma coleção ordenada de elementos de mesmo tipo, ou seja, não é possível misturar tipos dentro de um Array. Sendo assim, se definimos um Array de Int, todos os elementos dessa coleção devem necessariamente ser Int, não podendo ter uma String, um Double, ou qualquer outro tipo nessa coleção que não seja Int. Veremos agora algumas formas de criarmos Array em *Swift*. Crie um novo *Playground* e implemente o código abaixo:

```
//Criando um Array de Strings vazio
var emptyArray : [String] = []

//Criando um Array de Strings e alimentando valores na
criação
var shoppingList : [String] = ["Leite", "Pão", "Manteiga",
"Açúcar"]

//Usando inferência
var inferredShoppingList = ["Leite", "Pão", "Manteiga",
"Açúcar"]

//Testando se um Array está vazio
if shoppingList.isEmpty {
    print("A lista de compras está vazia")
} else {
    print("A lista de compras NÃO está vazia")
}

//Recuperando o total de elementos do Array
print("Nossa lista de compras possui \(shoppingList.count)
itens")
//Resultado: Nossa lista de compras possui 4 itens
```

Código-fonte 29 – Arrays
Fonte: Elaborado pelo autor (2019)

Veja que o Array possui uma propriedade chamada isEmpty, um Bool que nos informa se o Array está ou não vazio. Também possui uma propriedade que retorna o total de itens, a propriedade count. Essas propriedades estão presentes em todas as coleções usadas em *Swift*.

**DICA:** Em *Swift*, uma String também é uma coleção de caracteres, ou seja, se quisermos saber o total de letras presentes em uma String, basta utilizarmos a propriedade count. Faça o teste e comprove!

Para recuperarmos algum elemento de um Array, devemos acessá-lo através do uso de subscript, ou seja, definindo entre colchetes o índice onde se encontra esse elemento dentro do Array. O mesmo pode ser usado para modificar um elemento de um Array.

Para inserir itens, podemos usar o método **append** ou até mesmo utilizar o operador **+**, somando um novo Array ao Array atual. Outra técnica é utilizar o método **insert(_ newElement: String, at i: Int)**, que adiciona um elemento em uma

determinada posição do Array, porém, devemos ter certeza de que aquela posição existe antes de inserir o elemento. Vejamos abaixo alguns exemplos:

```
//Criando um Array de Strings e alimentando valores na
criação
var shoppingList : [String] = ["Leite", "Pão", "Manteiga",
"Açúcar"]

//Adicionando novos elementos utilizando append
shoppingList.append("Arroz")
shoppingList.append("Feijão")

//["Leite", "Pão", "Manteiga", "Açúcar", "Arroz", "Feijão"]


//Utilizando operador + para somar dois Arrays
var secondShoppingList : [String] = ["Ovos", "Suco"]
var finalShoppingList : [String] = shoppingList +
secondShoppingList

//["Leite", "Pão", "Manteiga", "Açúcar", "Arroz", "Feijão",
"Ovos", "Suco"]



//Inserindo um novo item no índice 2
finalShoppingList.insert("Chocolate", at: 2)

//["Leite", "Pão", "Chocolate", "Manteiga", "Açúcar",
"Arroz", "Feijão", "Ovos", "Suco"]
```

Código-fonte 30 – Manipulando Arrays
Fonte: Elaborado pelo autor (2022)

Além de inserir itens, podemos também removê-los, o que é possível através de vários métodos e formas, conforme veremos abaixo. Também note o uso do método **contains**, que serve para verificar se um determinado elemento encontra-se no Array, e do método **index(of element: String)**, que devolve o índice de determinado elemento, caso esteja presente na lista (esse método devolve um Int?, ou seja, se o elemento não for encontrado, retornará nil).

```
//Removendo o elemento na posição 3 e atribuindo a uma
variável
let banana = shoppingList.remove(at: 3)
```

```
//Removendo o primeiro e o último elementos
let milk = shoppingList.removeFirst()
let pear = shoppingList.removeLast()

//Removendo os 2 últimos e os 2 primeiros
//shoppingList.removeLast(2)
//shoppingList.removeFirst(2)
//Açúcar, Café

//Verificando se a lista contém um elemento
print(shoppingList.contains("Café"))     //true

//Pesquisando o índice de um elemento
//Note que o retorno desse método é um Int?, ou seja
//Para trabalharmos com este índice, precisamos
//desembrulhá-lo
if let coffeeIndex = shoppingList.firstIndex(of: "Açúcar") {
    print("O índice do Açúcar no Array é \(coffeeIndex)")
}
```

Código-fonte 31 – Manipulando Arrays
Fonte: Elaborado pelo autor (2019)

### 1.5.2 Dictionary (Dicionário)

Outra coleção bastante utilizada em *Swift* são os dicionários. Diferentemente dos Arrays, um Dicionário é uma coleção não ordenada de elementos de mesmo tipo. Como trata-se de uma coleção não ordenada, precisamos de uma outra forma para acessar os itens do Dicionário, pois usar subscript não é possível. É por isso que o dicionário armazena seus elementos vinculando-os a uma chave, semelhante a um dicionário convencional do mundo real, que nada mais é que uma coleção de significados associados a uma chave (no caso, a palavra em si).

Para criarmos um dicionário, primeiro definimos o tipo associado à sua chave e depois o tipo dos seus elementos. Isso é feito utilizando colchetes, separando os tipos da chave e dos elementos pelo sinal de dois pontos. Vamos pôr em prática através do código abaixo (lembre-se sempre de desenvolver um novo arquivo ou limpar o atual, caso seu Xcode comece a demorar para processar o código).

```
//Criando um dicionário cuja chave é uma String
//e os valores (elementos) são String também
var states : [String: String] = ["PA" : "Pará", "BA" :
"Bahia", "SP" : "São Paulo", "RJ" : "Rio de Janeiro"]

//Criando um dicionário vazio
var emptyStates : [String: String] = [:]

//Verificando se o dicionário está vazio
if states.isEmpty {
    print("O dicionário está vazio")
}

//Recuperando o valor atribuído a uma chave
let rj = states["RJ"]
print(rj)   //Optional("Rio de Janeiro")

//Um dicionário sempre devolve Optional, por isso
//precisamos desembrulhar seu conteúdo para utilizar
if let rj = states["RJ"] {
    print(rj)   //Rio de Janeiro
}
```

Código-fonte 32 – Dicionários
Fonte: Elaborado pelo autor (2019)

É importante perceber que sempre que tentamos recuperar um elemento de um Dicionário, este nos retorna um Optional, de modo que sempre temos que desembrulhar o Optional para trabalhar com o valor. Se ao recuperar um conteúdo vinculado a uma chave em um Dicionário nós obtemos um Optional, fica simples deduzir como fazemos para excluir um elemento de um Dicionário, certo? Exatamente, basta atribuirmos nil a uma respectiva chave para eliminarmos aquele conteúdo. E para verificarmos se determinado elemento existe? Simples, verificando se o retorno deste nos entrega nil ou não. Vejamos abaixo como realizar essas operações.

```
//Inserindo elementos
states["MS"] = "Mato Grosso do Sul"
//Verificando se um elemento está presente no dicionário
if states["MS"] != nil {
    print("Existe Mato Grosso do Sul no dicionário")
}
//Duas formas de remover elementos
states["RJ"] = nil
states.removeValue(forKey: "BA")
```

Código-fonte 33 – Manipulando Dicionários
Fonte: Elaborado pelo autor (2019)

### 1.5.3 Set (Conjunto)

Set é uma coleção mais direta, já que não permite elementos repetidos, ideal para quando precisamos definir um conjunto de itens cujo valor não pode se repetir, como os alunos que estão em uma turma ou a nossa lista de filmes favoritos, por exemplo. Para criar um Set, precisamos definir o tipo explicitamente, pois a inferência nesse caso atribuirá conteúdo a um Array por padrão. Criamos um Set usando a palavra Set seguida do tipo, entre os sinais < e >.

Vamos aprender a trabalhar com Set através de alguns exemplos listados no código abaixo. Cada código será explicado através dos respectivos comentários.

```
//Criando um Set de Strings
var movies: Set<String> = [
    "Matrix",
    "Vingadores",
    "Jurassic Park",
    "De Volta para o Futuro",
]

//Criando um set vazio
var movies2 = Set<String>()

//Inserindo elementos
movies.insert("Homem-Aranha: De Volta para o Lar")
print(movies.count) //5

//Perceba que o código abaixo não altera a quantidade
//de itens do Set pois ele não aceita itens repetidos.
movies.insert("Homem-Aranha: De Volta para o Lar")
print(movies.count)

//Podemos saber se o item foi ou não inserido se atribuirmos
//a operação de inclusão a uma variável e verificarmos se
//a propriedade inserted é true
let result = movies.insert("Homem-Aranha: De Volta para o
Lar")
print(result.inserted)

//Removendo elementos
```

```
movies.remove("Homem-Aranha: De Volta para o Lar")
print(movies)   //["Vingadores", "De Volta para o Futuro",
"Matrix", "Jurassic Park"]

//Pecorrendo um Set
for movie in movies {
    print(movie)
}

//Verificando se determinado elemento está contido no Set
if movies.contains("Matrix") {
    print("Matrix está na minha lista de filmes favoritos!!")
}

//Vamos criar um novo Set para realizarmos algumas operações
var myWifeMovies: Set<String> = [
    "De Repente 30",
    "Mensagem para você",
    "Sintonia de Amor",
    "De Volta para o Futuro",
    "Jurassic Park"
]

//Abaixo, estamos criando um novo Set (filmes favoritos)
//que será composto pela intersecção dos filmes presentes em
//movies com os presentes em myWifeMovies
let favoriteMovies = movies.intersection(myWifeMovies)
print(favoriteMovies)
//["De Volta para o Futuro", "Jurassic Park"]

//Criando um Set com todos os filmes
let allMovies = movies.union(myWifeMovies)
print(allMovies)
//["De Repente 30", "Mensagem para você", "Vingadores", "De
Volta para o Futuro", "Jurassic Park", "Sintonia de Amor",
"Matrix"]

//Removendo um Set de outro
movies = movies.subtracting(myWifeMovies)
print(movies)
//["Vingadores", "Matrix"]
```

Código-fonte 34 – Sets
Fonte: Elaborado pelo autor (2019)

Perceba como Set é uma coleção extremamente poderosa e muito útil em determinados casos.

# 2 OPERADORES

A maioria dos operadores existentes nas diversas linguagens de programação também estão disponíveis em *Swift* e atuam basicamente da mesma forma. Temos os operadores unários (que atuam apenas em um operando), os operadores binários (atuam em dois operandos) e o operador ternário (atua em três operandos). Vamos categorizar os operadores de acordo com suas funcionalidades.

## 2.1 Operador de Atribuição (=)

Fizemos uso do operador de atribuição, o sinal de =, para atribuir um valor a uma variável ou constante.

```
var height : Double = 1.75
let pi : Double = 3.14
```

Código-fonte 35 – Operador de atribuição
Fonte: Elaborado pelo autor (2019)

## 2.2 Operadores Aritméticos (+, -, *, /, %)

Esses operadores são utilizados para a realização das operações aritméticas, como soma, subtração e módulo (resto de uma divisão).

```
var a = 12
var b = 3

var sum = a + b                 //15
var subtract = a - b            //9
var multiplication = a * b      //36
var division = a / b            //4
var módulus = a % b             //Resto da divisão: 0
```

Código-fonte 36 – Operadores aritméticos
Fonte: Elaborado pelo autor (2019)

Quando utilizamos os sinais de + ou - à frente de um valor, estamos trabalhando com o operador de forma unária, ou seja, não estamos operando dois valores, mas, sim, alterando o sinal de um valor.

```
let negativeNumber = -12                //-12
let positiveNumber = +negativeNumber  //12
```

Código-fonte 37 – Operadores de atribuição
Fonte: Elaborado pelo autor (2019)

### 2.3 Operadores Compostos (+=, -=, *=, /=, %=)

São a junção dos operadores aritméticos com o operador de atribuição, ou seja, efetuam a operação e atribuem o valor na variável ao mesmo tempo.

```
var a = 2
var b = 3
var newValue = 5

newValue += a    //7
newValue -= b    //4
newValue *= a    //8
newValue /= a    //4
newValue %= b    //Resto da divisão: 1
```

Código-fonte 38 – Operadores compostos
Fonte: Elaborado pelo autor (2019)

Uma observação: em *Swift*, nós não temos os operadores de incremento (++) e decremento (- -). No início, eles existiam, porém, foram removidos da linguagem em uma de suas revisões, entre as versões 2.0 e 3.0.

### 2.4 Lógicos (&&, ||, !)

Estes operadores executam uma operação lógica, ou seja, sempre retornam verdadeiro ou falso.

```
var yes = true, no = false

print(yes && no)    //false
print(yes || no)    //true
print(!yes)         //false
```

Código-fonte 39 – Operadores lógicos
Fonte: Elaborado pelo autor (2019)

### 2.5 Comparação (>, <, >=, <=, ==, !=)

Usados para comparar valores. Retornam booleano.

```
var a = 12, b = 3, c = 7, d = 3
print(a > b)    //true
print(a < b)    //false
print(b >= d)   //true
print(a <= c)   //false
print(b == d)   //true
print(b != d)   //false
```

Código-fonte 40 – Operadores de comparação
Fonte: Elaborado pelo autor (2019)

### 2.6 Ternário ( condição ? “resultado se true” : “resultado se false” )

Este operador é o único que opera em três operandos. Com ele, podemos avaliar uma condição e atribuir um valor, caso a condição seja verdadeira, e outro valor, caso seja falsa.

```
var grade = 7.5
var result = grade > 7.0 ? "aprovado" : "reprovado"
print(result)    //aprovado
```

Código-fonte 41 – Operador ternário
Fonte: Elaborado pelo autor (2019)

### 2.7 Coalescência nula (??)

Como vimos anteriormente, este operador serve para desembrulhar um Optional e atribuir seu valor a uma variável ou, caso não seja possível, atribuir um valor padrão à variável.

```
var age: Int?
let myAge = age ?? 0     //0
age = 25
let newAge = age ?? 0    //25
```

Código-fonte 42 – Operador ternário
Fonte: Elaborado pelo autor (2019)

### 2.8 Closed Range e Half Closed Range (…. e ..<)

Estes operadores desenvolvem um *range*, ou seja, um intervalo de valores. O **Closed Range** cria um intervalo fechado, onde entram o valor inicial e o valor final, o **Half Closed Range** produz um intervalo entre o valor inicial e o valor imediatamente anterior ao valor final. Usaremos bastante quando trabalharmos com estruturas de repetição (como **for**, por exemplo).

```
let numbers = 1...10
for number in numbers {
    print(number)    //Imprime de 1 a 10
}

let newNumbers = 1..<10
for number in newNumbers {
    print(number)    //Imprime de 1 a 9
}
```

Código-fonte 43 – Operadores Closed Range e Half Closed Range
Fonte: Elaborado pelo autor (2019)

# 3 ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

Toda linguagem precisa ter uma estrutura em que possamos tomar uma decisão e agir de acordo com aquela decisão, ou seja, podemos definir o fluxo de nosso código com base no resultado de uma análise. Chamamos essas estruturas de estruturas condicionais e, em *Swift*, a mais utilizada é a estrutura **if else**, que veremos em detalhes a seguir. Outro recurso existente em toda linguagem de programação é a possibilidade de executarmos o mesmo trecho de código diversas vezes, seja controlado por um range específico ou até que uma condição seja alcançada.

## 3.1 If – else – else if

Para tomarmos uma decisão, caso certa condição seja verdadeira, e outra, caso seja falsa, fazemos uso da estrutura if – else – else if, como nos exemplos abaixo.

```
let number = 11
if number % 2 == 0 {
    print("Ele é par")
} else {
    print("Ele é ímpar")
}

var temperature = 18
var climate = ""
if temperature <= 0 {
    climate = "Muito frio"
} else if temperature < 14 {
    climate = "Frio"
} else if temperature < 21 {
    climate = "Clima agradável"
} else if temperature < 30 {
    climate = "Um pouco quente"
} else {
    climate = "Muuuito quente"
}
```

Código-fonte 44 – if – else – else if
Fonte: Elaborado pelo autor (2019)

### 3.2 Switch

Vimos no exemplo acima que, no caso da temperatura, precisamos fazer uso de vários *else if* para percorrermos todos os cenários existentes. Em situações como essa, a estrutura *switch* é mais adequada, já que foi criada especificamente para validar uma série dos cenários possíveis.

Um detalhe, em *Swift,* o *switch* precisa ser exaurido, ou seja, você deve contemplar todos os possíveis cenários para aquela variável que está sendo validada. Isso fica relativamente simples quando estamos validando um enum (veremos enum mais à frente), porém em situações em que o cenário é muito amplo ou até mesmo infinito, fazemos uso da cláusula **default**, que é o cenário escolhido quando nenhum dos outros é verdadeiro. Vejamos em exemplos:

```
var number = 7
switch number % 2 {
case 0:
    print("\(number) é par")
default:
    print("\(number) é ímpar")
}

//Exemplo com vários cenários no mesmo case
let letter = "z"
switch letter {
case "a", "e", "i", "o", "u":
    print("vogal")
default:
    print("consoante")
}

//Exemplo com range de letras
switch letter {
case "a"..."f":
    print("Você está na turma 1")
case "g"..."l":
    print("Você está na turma 2")
case "m"..."r":
    print("Você está na turma 3")
default:
    print("Você está na turma 4")
}
```

```
//Range de números
let speed = 33.0
switch speed {
case 0.0..<20.0:
    print("Primeira marcha")
case 20.0..<40.0:
    print("Segunda marcha")
case 40.0..<50.0:
    print("Terceira marcha")
case 50.0..<90.0:
    print("Quarta marcha")
default:
    print("Quinta marcha")
}
```

Código-fonte 45 – switch
Fonte: Elaborado pelo autor (2019)

### 3.3 While / repeat while

Esta estrutura de repetição é utilizada quando desejamos que um certo trecho de código seja executado enquanto (**while**) uma condição for verdadeira. Ou seja, somente sairemos do laço de repetição no momento em que a condição for falsa, e justamente por isso é de extrema importância que, dentro do laço, exista algum fator que faça aquela condição tornar-se falsa, do contrário, cairemos no chamado “loop infinito”, quando um trecho de código é executado para sempre. A estrutura **repeat while** é semelhante, porém sempre executa o código uma vez antes de validar a condição.

```
import Foundation

//Usando while
var life = 10
while life > 0 {
    print("O jogador está com \(life) vidas")
    life -= 1
}

//Usando repeat while
var tries = 0
```

```
var diceNumber = 0
repeat {
    tries += 1
    diceNumber = Int(arc4random_uniform(6)+1)
} while diceNumber != 6
print("você tirou 6 após \(tries) tentativas")
```

Código-fonte 46 – while, repeat while
Fonte: Elaborado pelo autor (2019)

No código acima, utilizamos **while** para executar a operação de imprimir a quantidade de vidas do jogador enquanto for maior do que 0. Note como reduzimos a quantidade de vidas em cada loop, para garantir que, em um determinado momento, essa quantidade seja zerada e o laço seja interrompido.

No segundo exemplo, foi feito o uso do **repeat while,** pois desejávamos que a operação fosse executada pelo menos uma vez (jogada de um dado para ver se cai o número 6). Nesse código, utilizamos um método chamado arc4random_uniform, que recebe um valor (do tipo UInt32) e gera um número aleatório que vai de 0 até o valor menos 1, ou seja, no nosso caso, de 0 a 5, e depois somamos com 1, para que tenhamos de 1 a 6. O retorno desse método também é um UInt32 e, por isso, transformamos em **Int** (utilizando o construtor de **Int**() passando o valor dentro do construtor), para podermos atribuir esse valor à variável **diceNumber** que, por inferência, é do tipo **Int.** Lembre-se de que *Swift* é uma linguagem fortemente tipada, o que significa que não podemos efetuar operações entre tipos diferentes e, nesse caso, como o tipo retornado pelo método é UInt32, não poderia ser atribuído a um **Int**.

Outro detalhe, esse método não está incluído nos métodos padrões da linguagem *Swift*. Está definindo em outro *framework* (*framework* é um conjunto de classes, métodos, tipos etc.), chamado Foundation, e é por isso que precisamos importar esse *framework*, para que nosso código tenha acesso ao arc4random_uniform. Essa importação é feita utilizando o comando import.

### 3.4 For in

Sem dúvida, a estrutura de repetição mais utilizada em *Swift* é o **for in**. Através dessa estrutura, podemos iterar (percorrer) uma coleção e recuperar todos os seus valores, o que é ideal quando necessitamos percorrer um Array ou um Dicionário, por

exemplo. Seu funcionamento é bem simples, utiliza **for**, seguido pelo nome da variável, que receberá cada elemento da coleção ou sequência, **in** e em seguida a coleção ou sequência que desejamos percorrer. Vamos analisar na prática, por meio dos diversos exemplos abaixo:

```
//Percorrendo um Array
let students = [
    "João Francisco",
    "Pedro Henrique",
    "Gustavo Oliveira",
    "Janaina Santos",
    "Francisco José"
]
for student in students {
    print("O aluno \(student) veio na aula de hoje!")
}

//Percorrendo uma sequência (range)
for day in 1...30 {
    print("Estou no dia \(day)")
}

//Note abaixo que uma String também é uma coleção
let name = "Steven Paul Jobs"
for letter in name {
    print(letter)
}

//Vejamos como percorrer um dicionário,
//imprimindo sua chave e valor. Neste dicionário
//a chave é String e o valor é Int
let people = [
    "Paulo": 25,
    "Renata": 18,
    "Kleber": 33,
    "Eric": 39,
    "Carol": 36
]

//A variável person, abaixo, é uma tupla que recebe a chave
//(key) e o valor (value) de cada elemento do dicionário
for person in people {
    print(person.key, person.value)
}

//Se quisermos, podemos inclusive decompor
```

```
//a tupla em variáveis
for (name, age) in people {
    print(name, age)
}

//Podemos quebrar a execução de um laço usando
//o comando break
let grades = [10.0, 9.0, 8.5, 7.0, 9.5, 5.0, 22.0, 6.5, 10.0]
for grade in grades {
    print(grade)
    if grade < 0.0 || grade > 10.0 {
        print("Nota \(grade)é inválida")
        break
    }
}
```

Código-fonte 47 – For in
Fonte: Elaborado pelo autor (2019)

Vimos acima, diversas formas de trabalho com o **for in**, seja em Arrays, Dicionários ou até mesmo em Strings.

### 3.5 Enumeradores

Enumeradores (ou Enums) são tipos criados pelo programador que servem para se definir um tipo comum para um conjunto fechado de valores, permitindo que trabalhe com esses valores de uma maneira segura em seu código. É muito utilizado quando há um cenário no qual deve-se armazenar uma informação baseada em um conjunto limitado de possibilidades. Para criarmos um enum, utilizamos a palavra reservada **enum** seguida do nome dos enumerados (com inicial maiúscula) e, entre chaves, definimos todos os valores possíveis utilizando **case**. Abaixo, iremos criar um enum que serve para definir uma bússola, com quatro possíveis valores (norte, sul, leste e oeste):

```
//Definindo um enum
enum Compass {
    case north
    case east
    case west
    case south
}

//Criando uma variável do tipo Compass
var direction = Compass.north

//Como Swift trabalha com inferência de tipo, podemos usar
// somente .valor, caso o tipo seja definido explicitamente
var direction2: Compass = .south
print("Minha direção é \(direction)")
//Minha direção é north

//Enums são muito usados com switch para análise do valor
switch direction {
case .north:
    print("Estamos indo para o norte")
case .south:
    print("Estamos indo para o sul")
case .east:
    print("Estamos indo para o leste")
case .west:
    print("Estamos indo para o oeste")
}
//Estamos indo para o norte
```

Código-fonte 48 – enum
Fonte: Elaborado pelo autor (2019)

Você notou, no código anterior, que quando imprimimos o valor de um enum, imprime-se o próprio nome do valor (north, no nosso exemplo). Enums são muito utilizados com *Switch*, pois geralmente precisamos verificar qual valor possuem para tomarmos uma decisão.

Repare que, quando validamos um enum usando *Switch,* se nosso *switch* verificar todos os possíveis valores automaticamente, ele é exaurido, ou seja, não precisaremos mais utilizar o default, pois, nesse caso, todos os cenários foram cobertos, e essa é outra vantagem de usarmos enums.

### 3.5.1. Valores padrões

Em *Swift*, podemos definir o tipo de um enum e, além disso, atribuir um valor padrão (também chamado de *raw value*) a cada um dos casos.

Vejamos o exemplo abaixo:

```
//Enum que define as posições das poltronas em um avião
//Veja que é possível atribuir um valor padrão a cada uma
delas
enum SeatPosition: String {
    case aisle = "corredor"
    case middle = "meio"
    case window = "janela"
}
var passengerSeat = SeatPosition.window

//Para imprimir o valor padrão, usamos a propriedade rawValue
print(passengerSeat.rawValue)    //janela

//Enum de Int com valores padrões
enum Month: Int {
    case january = 1, february, march, april, may, june,
july, august, september, october, november, december
}

var currentMonth: Month = .june
print("Estamos no mês \(currentMonth.rawValue) do ano")
```

Código-fonte 49 – enum com valores padrões
Fonte: Elaborado pelo autor (2019)

Neste caso, ao criamos o enum SeatPosition, definimos valores padrões para cada um dos seus casos e, ao imprimir o enum, utilizamos a propriedade **.rawValue**, que nos dá acesso ao valor padrão. Dessa forma, mostraremos ao usuário uma mensagem amigável e em português.

Outro detalhe importante que demonstramos no segundo exemplo é que, ao desenvolvermos um enum de Int e atribuir um valor ao seu primeiro elemento, o *Swift* automaticamente atribui sequencialmente os valores restantes, ou seja, quando criamos um enum com valor. june (junho), ao imprimirmos o seu rawValue notamos que seu valor padrão era 6, que foi atribuído automaticamente. Outro detalhe, é

possível colocar todos os valores do enum em sequência, separando por vírgulas, utilizando um único case.

### 3.5.2 Valores associados

Uma das características mais interessantes do enum em Swift é a possibilidade de associarmos valores a cada um dos seus casos. Dessa forma, quando formos definir uma variável como sendo um enum, podemos neste momento associar um valor a este enum. Cada caso do enum pode ter um valor associado de qualquer tipo, que é diferente de quando temos um valor padrão em que todos os tipos são iguais.

Vamos ao exemplo no qual precisamos criar um enum que represente medidas, e terá 3 valores, weight (medida de peso), age (idade) e size (medida de tamanho, que contém uma largura e uma altura). Veja que, nesse caso, não adianta definirmos um tipo para nosso enum, pois cada uma dessas medidas trabalha com tipos diferentes, uma vez que peso é definido por um Double, age por um Int e size pode ser definido por uma tupla contendo largura (Double) e altura (Double).

Além disso, a ideia é que, na criação da variável do tipo enum, nós possamos definir qual o seu caso (weight, age ou size) e, nesse momento, definirmos qual o seu valor, e não termos um valor padrão. Vejamos como isso é possível através do exemplo abaixo:

```
//Criando um enum que representa medidas (Measure) e que
//possui valores associados (Associated Values)
enum Measure {
    case weight(Double) //neste caso, associaremos um Double
    case age(Int)       //aqui, um Int
    case size(width: Double, height: Double) //Uma tupla
}

//Criando uma variável Measure com o valor age (idade)
//e definindo seu valor associado
let ageMeasure: Measure = .age(33)

//Agora, uma com o valor weight e associando um Double
let weightMeasure: Measure = .weight(82.3)
```

```
//Abaixo, estamos associando uma tupla ao enum do tipo size
let sizeMeasure: Measure = .size(width: 0.6, height: 1.71)

//Para recuperar os valores, precisamos trabalhar
//usando switch, da seguinte forma
switch sizeMeasure {
case .weight(let weight):
    print("O seu peso é \(weight)")
case .age(let age):
    print("A sua idade é \(age)")
case .size(let width, let height):
    print("As medidas são \(width)m de largura \(height)m de
altura")
}
//As medidas de tamanho são 0.6m de largura e 1.71m de altura
```

Código-fonte 50 – enum com valores associados
Fonte: Elaborado pelo autor (2019)

Veja que, com isso, podemos associar qualquer valor ao enum criado, desde que obviamente seja do tipo definido àquele caso. Esse recurso é muito poderoso e abre várias possibilidades quando trabalhamos com enums.

## 3.6 Funções e closures

### 3.6.1 Funções

Muitas vezes, ao longo do desenvolvimento de um App, nos deparamos com trechos das funcionalidades que precisam ser reutilizados em várias partes ao longo do código. Criamos esses blocos de código através do uso das funções, que são trechos dos comandos que executam operações definidas, podem receber valores (parâmetros) para se trabalhar e também podem retornar um resultado, que podem ser armazenados em uma variável ou constante.

#### 3.6.1.1 Criando funções

Vamos dar uma olhada, nos exemplos abaixo, em como criamos funções em *Swift:*

```
//Sintaxe para criação de funções
/*
 func nomeDaFuncao(parâmetro: Tipo) -> TipoDeRetorno {
    //Códigos
    return TipoDeRetorno
 }
*/

//Exemplo de uma função simples, sem parâmetros e sem retorno
func printHello() {
    print("Hello!!!!")
}
printHello()    //Hello!!!

//Função que aceita parâmetro
func say(message: String) {
    print(message)
}
say(message: "Vamos criar funções em Swift")

//Função que aceita mais de um parâmetro e que retorna algo
func sumNumbers(a: Int, b: Int) -> Int {
    return a + b
}
var result = sumNumbers(a: 10, b: 15)
print(result)   //15
```

Código-fonte 51 – Funções
Fonte: Elaborado pelo autor (2019)

Você percebeu que, quando vamos chamar uma função, precisamos escrever os parâmetros na sua chamada, ou seja, ao chamarmos a função **sumNumbers**, tivemos que escrever os valores de ambos os parâmetros (a e b).

É possível modificarmos o nome que um parâmetro terá na chamada da função (nome externo), de modo que seja diferente do nome que há no uso da função (nome interno). Isso é muito útil para tornar aquela função mais legível para quem vai utilizá-la, criando nomes que façam mais sentido interna e externamente.

Além disso, é possível omitir completamente o nome dos parâmetros na sua chamada, tornando a função mais enxuta e simples de usar. Vale lembrar que sempre

devemos prezar pela clareza e legibilidade no nome da função e de seus parâmetros, criando funções que, ao serem utilizadas, deixem bem claro o que fazem e como são usadas. Vejamos alguns exemplos:

```
//A função abaixo serve para darmos uma mensagem de
//boas-vindas:
func say(welcome message: String, to person: String) {
    print("\(message) \(person)!")
}
say(welcome: "Seja bem-vindo", to: "aluno")
//Seja bem-vindo aluno!

//Removendo o nome externo do primeiro parâmetro
func say(_ message: String, to person: String) {
    print("\(message) \(person)!")
}
say("Olá", to: "Fabiana")
//Olá Fabiana!

//Removendo todos os parâmetros externos
func sumNumbers(_ number1: Int, _ number2: Int) -> Int {
    return number1 + number2
}
print(sumNumbers(3, 7)) //10
```

Código-fonte 52 – Nomes internos e externos de parâmetro
Fonte: Elaborado pelo autor (2019)

Note que na primeira função utilizamos nomes diferentes, pois, durante a leitura, faz mais sentido utilizarmos *welcome* e *to* como nomes externos e *message* e *person* como nomes internos, para que a leitura da função faça mais sentido. Na verdade, como faria ainda mais sentido a ausência do nome *welcome* no primeiro parâmetro, criamos uma segunda versão da função sem o nome externo desse parâmetro, utilizando o caractere _. Fizemos o mesmo na segunda função, que não tem nenhum parâmetro com nome externo.

### 3.6.1.2 Utilizando funções como argumento

Em *Swift*, as funções são o que chamamos em programação de **First-Class Citizen**. Esse termo é utilizado para qualquer entidade que suporte todas as

operações disponíveis a um objeto, como, por exemplo, atribuir essa entidade a uma variável, usá-la como parâmetro e retorno de uma função etc.

Como são first-class citizens, as funções podem ser utilizadas como argumento de outras funções, assim como um **Int**, uma String ou qualquer outra estrutura.

Implementem os exemplos abaixo para treinarmos o uso das funções como parâmetro. Neste exemplo, criaremos quatro funções que possuem a mesma assinatura (ou seja, aceitem a mesma quantidade e tipo de parâmetros e devolvem o mesmo tipo). Usaremos essas funções como parâmetro de uma outra função.

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func subtract(_ a: Int, _ b: Int) -> Int {
    return a - b
}
func divide(_ a: Int, _ b: Int) -> Int {
    return a / b
}
func multiply(_ a: Int, _ b: Int) -> Int {
    return a * b
}



//A função abaixo utiliza outra função como parâmetro
func applyOperation(_ a: Int, _ b: Int, operation: (Int, Int)
-> Int) -> Int {
    return operation(a, b)
}

let result = applyOperation(10, 20, operation: multiply)
print(result)   //200
```

Código-fonte 53 – Nomes internos e externos de parâmetro
Fonte: Elaborado pelo autor (2019)

A função applyOperation aceita três parâmetros, os parâmetros a e b, ambos do tipo **Int**, e um terceiro parâmetro chamado operation, cujo tipo é uma função. Nesse caso, definimos o tipo com base na assinatura das funções que serão aceitas, ou seja,

no nosso exemplo o parâmetro operation aceita qualquer função que trabalhe com 2 parâmetros **Int** e que retorne um **Int.**

Internamente, o método applyOperation simplesmente executa a função passada utilizando os parâmetros a e b como parâmetros dessa função, ou seja, se a função for, por exemplo, a função sum, que faz a soma de dois números, applyOperation executará a operação de soma em seus parâmetros e retornará esse resultado. No exemplo, usamos a operação multiply com os valores 10 e 20 e obtivemos como resultado 200, que é a multiplicação de 10 por 20. Experimente utilizar as demais funções e veja o resultado mudar.

### 3.6.1.3 Retornando funções

Vejamos agora como desenvolver uma função que retorne outra. No próximo exemplo, criaremos uma função que recebe uma String como parâmetro, que irá conter o nome em português da operação que queremos receber, e retornará à função correspondente. Para facilitar, ao invés de utilizarmos **(Int, Int) -> Int** a fim de definir o tipo de função de retorno, vamos criar um “apelido” para esse tipo, de modo que fique mais simples de escrever.

É possível definir um apelido para qualquer tipo em *Swift,* utilizando a palavra reservada **typealias**, seguida do novo nome do tipo e atribuindo o tipo original. Ao fazer isso, ao invés de usarmos (Int, Int) -> Int, podemos usar o apelido.

```
//Criando um "apelido" para (Int, Int) -> Int
typealias Op = (Int, Int) -> Int

//Método que retorna outro método
func getOperation(_ operation: String) -> Op {
    switch operation {
    case "soma":
        return sum
    case "subtração":
        return subtract
    case "multiplicação":
        return multiply
    default:
        return divide
    }
}
```

```
//Abaixo, a variável operation receberá a função
//retornara pelo método getOperation
var operation = getOperation("subtração")
operation(30, 15)
```

Código-fonte 54 – Função que retorna outra função
Fonte: Elaborado pelo autor (2019)

Neste exemplo, atribuímos o retorno do método getOparation à variável Operation, ou seja, passou a ser também uma função (no exemplo, a função de subtração) e pode ser usada normalmente, por isso operation(30, 15) retorna 15.

### 3.7 Closures

Anteriormente, no método applyOperation, passamos uma das quatro funções criadas como parâmetro, porém e se quiséssemos criar uma nova função ao invés de utilizar uma criada anteriormente? É aí que entram as closures.

Closures são blocos autocontidos de código, semelhante às funções, contudo não possuem um nome que possa ser usado para chamá-la. São o que chamamos de funções anônimas, e são um recurso extremamente útil da linguagem *Swift.*

### 3.8 Utilizando closures

Vamos dar uma olhada, nos exemplos abaixo, em como produzimos closures em *Swift.* A sintaxe é bem semelhante à de uma função, com algumas pequenas mudanças, como a ausência da palavra func e do nome da função. Outras mudanças são o uso da chave inicial no início e não no fim da linha e o uso da palavra in no lugar onde ficava a chave inicial. Veja abaixo:

```
//Sintaxe de uma função
/*
 func nomeDaFuncao(parâmetro: Tipo) -> TipoDeRetorno {
    //Códigos
    return TipoDeRetorno
 }
 */

//Sintaxe de uma closure
/*
  {(parâmetro: Tipo) -> TipoDeRetorno in
    //Códigos
    return TipoDeRetorno
 }
 */
```

Código-fonte 55 – Sintaxe de função *versus* closure
Fonte: Elaborado pelo autor (2018)

Vamos fazer uma nova chamada do método applyOperation, porém dessa vez iremos criar a closure contendo o código que desejamos executar. Neste exemplo, faremos a operação de módulo (o resto de uma divisão). Deve manter no seu *Playground* o método applyOperation criado anteriormente (repetimos o mesmo no código abaixo).

```
func applyOperation(_ a: Int, _ b: Int, operation: (Int, Int)
-> Int) -> Int {
    return operation(a, b)
}



//Implementando uma closure que realiza a operação
//de módulo
applyOperation(14, 5, operation: {(x: Int, y: Int) -> Int in
    return x % y
})
```

Código-fonte 56 – Utilizando closure
Fonte: Elaborado pelo autor (2019)

Com essa closure, definimos o que o método applyOperation deverá fazer com os parâmetros, e nesse caso, desenvolvemos um código que retorna o resto da operação entre 14 e 5.

### 3.9 Forma simplificada

É possível utilizar uma closure em sua forma reduzida, ou simplificada, que reduz bastante a quantidade de código escrito, fazendo com que uma closure, se for simples, possa ser expressa em uma única linha.

Em suma, faremos uso das características de inferência e tipagem forte da linguagem *Swift*, em que a linguagem sabe exatamente o que receberá. Ainda usando nosso método applyOperation, vamos fazer uma análise: o parâmetro operation aceita apenas uma closure (ou função) que trabalhe com dois parâmetros do tipo **Int**, ou seja, se a linguagem espera esse tipo, podemos omitir essa informação (inclusive os parênteses) deixando o uso como da forma abaixo.

```
applyOperation(14, 5, operation: {x, y -> Int in
    return x % y
})
```

Código-fonte 57 – Simplificação 1
Fonte: Elaborado pelo autor (2019)

Se podemos retirar o tipo dos parâmetros, então podemos também tirar o tipo de retorno, que só pode ser **Int**. Veja como fica essa nova simplificação:

```
applyOperation(14, 5, operation: {x, y in
    return x % y
})
```

Código-fonte 58 – Simplificação 2
Fonte: Elaborado pelo autor (2019)

Bom, é sabido que sempre teremos que trabalhar com dois parâmetros, podemos também omitir essa informação, porém, como iremos referenciar esses parâmetros dentro da closure? Simples, a própria linguagem atribui tais parâmetros a

variáveis iniciadas por $, ou seja, o primeiro parâmetro passa a se chamar $0, o segundo $1 e assim sucessivamente. Aproveitando que iremos remover os parâmetros, podemos remover também o **in**. Veja como fica o código:

```
applyOperation(14, 5, operation: {
    return $0 % $1
})
```

Código-fonte 59 – Simplificação 3
Fonte: Elaborado pelo autor (2019)

Como o método esperado tem que retornar um **Int**, e em nossa closure há 1 linha de código, é óbvio que essa linha tem que ser a de retorno, certo? Nesse caso, podemos retirar essa informação e deixar nosso código ocupando apenas 1 linha.

```
applyOperation(14, 5, operation: {$0 % $1})
```

Código-fonte 60 – Simplificação final
Fonte: Elaborado pelo autor (2019)

Esta é a forma mais simplificada e reduzida de utilizarmos closures, e existem vários métodos em *Swift,* principalmente métodos presentes em coleções e sequências (Array, Dictionary etc.). Vamos conhecer alguns deles.

### 3.10 Map, filter e reduce

Muitas vezes precisamos efetuar operações em Arrays ou Dictionaries em que é necessário percorrer a coleção para extrair determinados elementos ou apenas modificá-los. Normalmente, a primeira ideia que nos vem à mente é utilizar o **for in** para iterar a coleção e realizarmos o desejado, porém existem vários métodos que utilizam closures para esses fins.

O **map** é um método presente em coleções que percorre a coleção e executa uma closure em cada um dos seus elementos, devolvendo a nova coleção gerada. No exemplo abaixo, temos um Array dos nomes e desejamos criar um novo Array contendo todos os nomes escritos em letras maiúsculas. Veja como é possível realizar essa operação utilizando map:

```
let names = ["João", "Paulo", "Henrique", "Ana", "Beatriz",
"Carla", "Caroline"]

//Aplicando map em names
let uppercasedNames = names.map({$0.uppercased()})
print(uppercasedNames)
//["JOÃO", "PAULO", "HENRIQUE", "ANA", "BEATRIZ", "CARLA",
"CAROLINE"]
```

Código-fonte 61 – Map
Fonte: Elaborado pelo autor (2019)

Criamos uma closure que retorna a versão em maiúsculas (usando o método de String **uppercased()**) de todos os nomes. Lembrando que o $0 nesse caso se refere ao parâmetro da closure que representa cada um dos nomes da coleção.

No próximo exemplo, queremos filtrar a nossa coleção e gerar um novo Array contendo apenas os nomes compostos por 5 ou menos letras, ou seja, agora vamos criar um novo Array contendo parte dos elementos do Array principal. Quando temos casos como este, utilizamos o método **filter** que, como o próprio nome sugere, filtra uma coleção, devolvendo outra com os elementos que foram filtrados.

```
//Aplicando filter em names
let filteredNames = names.filter({$0.count < 6})
print(filteredNames)
//["João", "Paulo", "Ana", "Carla"]
```

Código-fonte 62 – Filter
Fonte: Elaborado pelo autor (2019)

O método filter solicita que seja passada a função que servirá para filtrar os elementos. Essa função deve conter a lógica que será implementada em cada um dos elementos do Array e, caso essa lógica retorne true, aquele elemento deverá fazer parte do novo Array. Veja que, no nosso exemplo, nós criamos uma closure que verifica se a contagem dos caracteres de cada nome é menor que 6, ou seja, se tem 5 ou menos letras.

No nosso último exemplo, vamos criar um Array de Double que representa algumas movimentações feitas em uma conta corrente, ou seja, entrada e saída de valores (valores positivos representam entrada, valores negativos indicam saída). Como podemos saber o saldo final dessa conta?

Em *Swift*, temos um método que está presente em todas as coleções e serve para combinar, ou seja, juntar todos os valores presentes naquela coleção, segundo uma lógica estipulada por nós. No nosso exemplo, essa combinação deverá ser feita através da soma de todos esses valores, porém pode implementar a lógica que desejar. Vale ressaltar que, apesar de em nosso exemplo estarmos trabalhando com Double, podemos utilizar Arrays de qualquer tipo, mudando obviamente a lógica que implementar. Esse método é o **reduce**, e vamos ver abaixo como ficaria sua implementação.

```
//Utilizando Reduce
var transactions = [500.0, -45.0, -70.0, -25.80, -321.72,
190.0, -35.15, -100]

//Sintaxe do reduce
//func reduce<Result>(_ initialResult: Result, _
nextPartialResult: (Result, Double) throws -> Result)
rethrows -> Result

let balance = transactions.reduce(0.0, {$0 + $1})
print("Seu saldo é R$ \(balance)")
//Seu saldo é R$ 92.33
```

Código-fonte 63 – Reduce
Fonte: Elaborado pelo autor (2019)

O método reduce recebe dois parâmetros. O primeiro, chamado initialResult, contém o valor inicial da operação e o segundo, nextPartialResult, contém uma closure que receberá, a cada iteração, o resultado da operação e o elemento do Array. No nosso exemplo, definimos que o valor inicial seria 0 e que a cada iteração o valor será somando com o elemento do Array, ou seja, na primeira iteração teremos 0.0 + 500.0, na segunda 500.0 + -45.0, na terceira 455.0 + -70.0, e assim sucessivamente até chegar ao último elemento do Array.

### 3.11 Generics

Você deve ter notado, na sintaxe do método reduce, um tipo que não conhecemos, o tipo Result. Esse tipo realmente não é um tipo válido ou existente na linguagem, mas sim um tipo que foi criado na assinatura desse método e que serve para representar um tipo genérico, ou seja, uma indicação de que qualquer tipo pode ser utilizado naquele parâmetro. Perceba que o método exige que o mesmo tipo utilizado no parâmetro initialResult seja usado dentro da closure e também seja o tipo de retorno da função. Esse tipo foi definido logo após o nome do método (reduce**<Result>**) e isso indica que, nesse método, usaremos um tipo chamado de Result que pode ser representado por qualquer tipo existente ou criado por você.

O nome desse recurso é Generics e é um recurso muito poderoso em linguagens de programação, pois não ficamos limitados a um tipo específico quando criamos nossos métodos ou classes. Para exemplificar seu uso, vamos criar abaixo uma função que serve para trocar dois inteiros de lugar.

```
func swapInts(_ a: inout Int, _ b: inout Int) {
    let tempA = a
    a = b
    b = tempA
}
var int1 = 20
var int2 = 30
swapInts(&int1, &int2)
print(int1, int2)    //30, 20
```

Código-fonte 64 – Método swapInts
Fonte: Elaborado pelo autor (2019)

Antes, algumas explicações. Quando passamos um Int, String, Double etc. como parâmetro de uma função, esses parâmetros são constantes, ou seja, você não consegue alterá-los dentro do código, a menos que sejam marcados como inout, um modificador que indica que aqueles parâmetros serão passados por referência e não por cópia. Nesse caso, a e b não possuem uma cópia dos valores 20 e 30, mas apontam para o mesmo lugar na memória onde esses valores residem. Com isso, além de ser possível trocar os valores desses parâmetros, estamos trocando também

os valores das variáveis int1 e int2. Isso só é necessário pois **Int**, em *Swift,* não é uma classe e sim uma Struct, como aprendemos, e as Structs, quando passadas para uma função ou para uma variável, são passadas por cópia, que significa que uma cópia daquela struct é atribuída à nova variável ou aos parâmetros de uma função.

Voltando ao código, vimos que através do nosso método swapInts é possível trocar o valor de 2 variáveis do tipo **Int**, porém, e se eu quisesse utilizar o mesmo método em um Double, ou em uma String, por exemplo? Não podemos passar uma String como parâmetro de um método que só aceita **Int**, correto? São casos como esse que o uso de Generics nos ajuda a resolver nossos problemas. Vejamos como ficaria a criação de um método que troca dois valores de lugar, porém trabalhe com tipos genéricos.

```
//Método genérico
func swapValues<T>(_ a: inout T, _ b: inout T) {
    let tempA = a
    a = b
    b = tempA
}
var name1 = "Jaqueline"
var name2 = "Carlos"
swapValues(&name1, &name2)
print(name1, name2) //Carlos Jaqueline

var value1 : Double = 55.7
var value2 : Double = 28.9
swapValues(&value1, &value2)
print(value1, value2) //28.9 55.7
```

Código-fonte 65 – Generics
Fonte: Elaborado pelo autor (2019)

Para criarmos uma função que trabalhe com Generics, precisamos definir um nome que será dado ao tipo genérico utilizado. No nosso exemplo, utilizamos a letra T para representá-lo, e isto é feito definindo esse tipo entre <> logo ao lado do nome da função. Agora, o tipo T existe dentro de swapValues, foi utilizado para definir que os parâmetros a e b são do tipo T. Dessa maneira, podemos utilizar objetos de qualquer tipo como parâmetros de nossa função, não importa se são Strings, Doubles, ou qualquer outro que desejarmos.

# 4 CLASSES X STRUCTS

## 4.1 Definição e construção

Podemos construir nossos próprios tipos em Swift utilizando Struct, porém também é possível definir um tipo utilizando uma estrutura mais complexa e mais rica que o Struct, que são as classes.

Antes de aprendermos como criar classes ou structs em *Swift,* vamos descrever as semelhanças e diferenças entre as duas estruturas.

Classes e Structs possuem as seguintes características:

- Definem propriedades para armazenar valores.
- Definem métodos para fornecer funcionalidades.
- Definem inicializadores para configurar seu estado inicial.
- Podem ser estendidas para expandir suas funcionalidades além das presentes na sua implementação.

Porém, existem certas características que somente as classes possuem, que são:

- Trabalham com herança, o que permite que uma classe possa herdar as características de outra.
- Type casting, o que permite checar e interpretar uma classe como sendo outra.
- Desconstrutores que permitem que uma instância de uma classe possa limpar da memória os recursos que não estão sendo mais utilizados.
- Contagem de referência (*Reference counting*) permite que exista mais de uma referência à mesma instância de uma classe.

Para criarmos uma Struct, utilizamos a palavra reservada **struct**, seguida do nome da struct (iniciando em maiúscula) e sua implementação entre chaves (**{ }**). Veja o exemplo de criação de uma struct chamada RGBColor.

```
struct RGBColor {

    //Propriedades armazenadas
    var red : Int
    var green : Int
    var blue : Int

    //Métodos de classe
    func printColor() {
        print("""
        Red: \(red)
        Green: \(green)
        Blue: \(blue)
        """)
    }
}

//Instanciando a struct RGBColor
var rgbYellow = RGBColor(red: 255, green: 255, blue: 0)

//Acessando e Alterando uma propriedade de rgbYellow
rgbYellow.red = 254
```

Código-fonte 66 – Classes
Fonte: Elaborado pelo autor (2019)

Uma característica interessante com relação às structs é a não necessidade de criar métodos construtores (ou métodos inicializadores), que serão definidos automaticamente de acordo com as propriedades de classe indicadas. Caso seja necessária a criação de métodos construtores, eles podem ser inseridos sem quaisquer problemas, de acordo com a sintaxe que será exibida para a criação de construtores em uma classe. Para criarmos uma classe, utilizamos a palavra reservada **class**, seguida do nome da classe (iniciando em maiúscula) e sua implementação entre chaves (**{ }**). Veja o exemplo de criação de uma classe chamada Person:

```
class Person {

    //Propriedades armazenadas
    var name : String
    var age : Int = 0
    var married : Bool
```

```
    //Método construtor da classe
    init(aName: String, isMarried: Bool) {
        name = aName
        married = isMarried
    }

    //Métodos de classe
    func speak(sentence: String) {
        if age < 3 {
            print("gugu dada")
        } else {
            print(sentence)
        }
    }

    func introduce() -> String {
        return "Olá, meu nome é \(name)"
    }
}

//Instanciando a classe Person
let jobs = Person(aName: "Steve Jobs", isMarried: true)

//Alterando uma propriedade de jobs
jobs.age = 39
```

Código-fonte 67 – Classes
Fonte: Elaborado pelo autor (2019)

A classe acima (**Person**) representa uma pessoa e possui as propriedades name, married e age, que armazenam o nome, casado (se for true, significa sim) e a idade. Em uma classe ou estrutura, as propriedades que armazenam um conteúdo também são chamadas de propriedades armazenadas.

Toda classe necessita de um método construtor (ou método inicializador), para criar uma instância daquela classe (também chamado de objeto), utilizamos seu método construtor, o init.

**DICA:** Dentro das classes ou Structs, as funções passam a ser chamadas de métodos, e as variáveis são chamadas de propriedades.

O método construtor é o método que cria uma instância daquela classe e tem por obrigação alimentar qualquer propriedade que não tenha sido inicializada. No nosso exemplo, a propriedade age é a única que foi definida e inicializada com um valor (0). As demais (name e married) precisam ser inicializadas, e cabe ao método

construtor efetuar essa tarefa, e é por isso que o construímos solicitando dois parâmetros, name e married, que serão repassados às respectivas propriedades. Vale ressaltar que os nomes dos parâmetros devem se diferenciar dos nomes das propriedades, de forma a identificá-los no momento certo do código. Como exemplo podemos citar a propriedade **name**, que recebe como valor o parâmetro **aName.**

A nossa classe Person possui dois métodos, o **introduce()**, que serve para retornar a apresentação da pessoa, e **speak(sentence: String)**, que faz com que a pessoa fale algo. Nesse método, incluímos uma regra informando que uma pessoa com menos de 3 anos só fala **"gugu dada"**.

Para instanciarmos uma classe, criamos uma variável e atribuímos à mesma a chamada do método construtor da classe que desejamos. Para chamarmos o método construtor, usamos apenas **(** e **)** após o nome da classe (apesar de que também é possível utilizar **.init()** ), passando os valores dos parâmetros do método.

### 4.2 Propriedades computadas

Nossa classe possui três propriedades armazenadas, porém, em *Swift* podemos fazer uso do que chamamos de propriedade computada. Vamos imaginar que em nossa classe seria interessante ter uma propriedade chamada maritalStatus, que retornaria uma String contendo "casado" ou "solteiro", de acordo com o estado civil da pessoa. Poderíamos tranquilamente criar essa propriedade, mas você não concorda que essa informação pode ser recuperada a partir de married, e que criar uma nova propriedade faria com que tivéssemos que sempre alimentar duas propriedades para termos a mesma informação? Uma das soluções para esse problema seria criarmos um método que retornasse o estado civil em String, porém métodos são recursos que fazem mais sentido quando precisamos passar um parâmetro para obter um resultado, e no nosso caso, maritalStatus é uma definição de Person, é uma característica e é representada por propriedades.

Existe um recurso em *Swift* que nos permite ter uma propriedade que não armazena nenhum valor, apenas utiliza e trabalha um valor existente. A essas propriedades chamamos de propriedades computadas. Vejamos abaixo como

poderíamos criar a propriedade computada maritalStatus. Colocaremos apenas o código que será adicionado à classe Person. Insira o código logo abaixo da variável married:

```
class Person {

    //.....

    //Propriedade computada
    var maritalStatus: String {
        if married {
            return "casado"
        } else {
            return "solteiro"
        }
    }

    //......
}
```

Código-fonte 68 – Propriedades computadas
Fonte: Elaborado pelo autor (2019)

Agora se quisermos imprimir o estado civil da pessoa, podemos utilizar maritalStatus ao invés de married, pois fica mais fácil e legível ao usuário visualizar o estado civil como **"casado"** ou **"solteiro"** do que como **true** ou **false**. Neste exemplo, criamos uma variável computada somente de leitura, então, apesar de podermos utilizá-la para recuperar o estado civil da pessoa, não podemos usá-la para atribuir um novo estado civil, ou seja, se quisermos modificar o estado civil utilizando a variável maritalStatus, precisamos definir que ela é também de escrita. Isso é possível utilizando as palavras **get** e **set**, que definem o que acontece quando queremos recuperar um valor (get) e quando queremos atribuir um valor (set). Troque a implementação de maritalStatus pela implementação abaixo:

```
var maritalStatus: String {
    get {
        if married {
            return "casado"
        } else {
            return "solteiro"
```

```
            }
        }
        set {
            if newValue == "casado" {
                married = true
            } else {
                married = false
            }
        }
}
```

Código-fonte 69 – Propriedades computadas
Fonte: Elaborado pelo autor (2019)

Quando utilizamos set, temos acesso ao valor passado para a variável através da constante **newValue**. É isso que usamos para verificar o que foi passado e para realizar a devida alteração em married, que é nossa propriedade armazenada. Abaixo, uma utilização dessa nova propriedade.

```
let onePerson = Person(aName: "Paulo", isMarried: false)

//Alterando o estado civil utilizando maritalStatus
onePerson.maritalStatus = "casado"
print(onePerson.married, onePerson.maritalStatus)
//true casado
```

Código-fonte 70 – Propriedades computadas
Fonte: Elaborado pelo autor (2019)

### 4.3 Propriedades/métodos de classe

Todas as propriedades criadas na nossa classe Person são o que chamamos de **propriedades de instância**, o que significa que seu uso é possível através de uma instância da classe (através de um objeto). Podemos criar propriedades que não necessitam de uma instância para serem utilizadas e podem ser acessadas diretamente na classe. Tais propriedades são chamadas de **propriedades de classe**.

Na nossa classe Person, vamos desenvolver uma propriedade de classe que retorna a classe de animal à qual uma pessoa faz parte (mamífero). Como esta é uma informação referente à própria classe em si, ou seja, toda pessoa é um mamífero,

vamos criá-la como propriedade de classe, pois assim podemos saber de qual classe animal uma Person é sem a necessidade de atribuirmos a instância dessa classe.

Propriedades de classe são criadas utilizando a palavra reservada **static**, por isso costumam ser chamadas também de propriedades estáticas. Uma propriedade estática mantém seu valor se alterado ao longo do código, o que a torna muito útil em determinados cenários. Além das propriedades de classe, podemos ter métodos de classe, que, como as propriedades, podem ser utilizados sem a necessidade de uma instância, porém, para a criação de tais métodos, utilizamos a palavra reservada **class** antes da definição.

Ao longo do curso veremos que existem vários métodos e propriedades de classe dentro dos *frameworks* utilizados no desenvolvimento dos aplicativos, e nós faremos uso de vários em nossos projetos. Vejamos abaixo como desenvolver métodos e propriedades de classe em *Swift.* Tais propriedades e métodos serão inseridos na classe Person.

```
class Person {

    //.....

        //Propriedade de classe (estática)
    static let animalClass: String = "mamífero"



    //Método de classe
    class func getInfo() -> String {
        return "Esta pessoa é um \(Person.animalClass) que
possui nome, estado civil e idade"
    }


    //......
}

//Usando propriedades e métodos de classe
print(Person.animalClass)
//mamífero
print(Person.getInfo())
//Pessoa: mamífero que possui nome, sexo e idade
```

Código-fonte 71 – Propriedades e métodos de classe
Fonte: Elaborado pelo autor (2019)

Um método ou uma propriedade de classe são utilizados através da própria classe em si e não da instância, por isso que devemos chamá-lo na própria classe, como mostram os exemplos acima. Até mesmo internamente (como no caso do método **getInfo()**), precisamos referenciá-lo através da classe.

### 4.4 Herança

Uma das principais vantagens de se trabalhar com classes é que elas fazem uso de herança, ou seja, podem herdar as características de outra classe, que nesse caso chamamos de **classe mãe** ou **super**.

Em *Swift*, para definirmos que uma classe herda outra, utilizamos o sinal: (dois pontos) após o seu nome, seguido da classe a qual herdará. Caso a classe filha implemente uma nova propriedade e esta não tenha nenhum valor associado durante sua definição, devemos criar um novo construtor, que terá o papel de alimentar tanto esta quanto todas as propriedades não inicializadas da classe mãe. Nesse caso, devemos primeiro inicializar as propriedades da classe filha para depois inicializar as propriedades da classe mãe. Essa etapa é feita chamando o construtor da classe super, usando a palavra reservada super seguida do método construtor (init).

Vamos construir uma nova classe chamada **Student**, que irá representar um estudante e, como todo estudante é uma pessoa, vamos criá-la herdando da classe **Person**. Nosso estudante possui uma propriedade a mais, o **rm**, e em função disso necessitará de um construtor próprio. Veja a implementação abaixo:

```
class Student: Person {
    var rm: String

    init(aName: String, isMarried: Bool, aRm: String) {
        rm = aRm
        super.init(aName: aName, isMarried: isMarried)
    }
}
let oneStudent = Student(aName: "Monteiro Lobato", isMarried:
true, aRm: "1234")
print(oneStudent.name)
```

Código-fonte 72 – Herança
Fonte: Elaborado pelo autor (2019)

Observe que o método construtor de Student precisa solicitar todas as informações para não só criar um Student, como também criar um Person, cujo construtor é chamado pelo uso de **super**, que nos dá acesso direto à classe mãe.

### 4.5 Sobrescrita

As classes filhas podem modificar propriedades ou métodos das classes mãe utilizando uma técnica chamada sobrescrita (override). Em *Swift,* fazemos isso utilizando a palavra reservada **override** seguida da nova implementação do método ou propriedade que será modificado. Com essa técnica, uma classe não precisa necessariamente executar os métodos ou propriedades computadas da mesma forma que a sua classe mãe executa.

Vamos modificar o método **introduce()** fazendo com que agora ele, além de retornar a apresentação do estudante pelo seu nome, também informe o seu RM, e faremos uso do próprio método introduce() da classe mãe para recuperar essa informação inicial. Nossa classe Student ficará assim:

```
class Student: Person {
    var rm: String

    init(aName: String, isMarried: Bool, aRm: String) {
        rm = aRm
        super.init(aName: aName, isMarried: isMarried)
    }



    override func introduce() -> String {
        return "\(super.introduce()) e meu RM nesta escola é
\(rm)"
    }
}
let oneStudent = Student(aName: "Monteiro Lobato", isMarried:
true, aRm: "1234")
print(oneStudent.introduce())
```

```
//Olá, meu nome é Monteiro Lobato e meu RM nesta escola é
1234
```

Código-fonte 73 – Sobrescrita
Fonte: Elaborado pelo autor (2019)

### 4.6 Extension (extensão)

Quando definimos nossos próprios tipos (seja classe ou struct), criamos suas propriedades e seus métodos e assim concebemos a sua estrutura da maneira como desejamos, entretanto, existem cenários nos quais seria interessante se pudéssemos adicionar novas características a uma determinada classe ou structs que não foi criado por nós, ou até mesmo modificar as que criamos, porém fora da sua definição. Vamos a um exemplo prático, criaremos uma variável **name** que contém nosso nome. Se quisermos saber a quantidade das letras que nosso nome possui, podemos simplesmente acessar a propriedade **count** presente na String, correto? E se nós quiséssemos ter uma nova String contendo apenas as iniciais do nome? Seria bem legal se todas as String tivessem uma propriedade chamada initials que nos devolvesse isso, correto? Outro exemplo, digamos que desejasse saber o total das vogais presentes em um nome? Seria muito útil se uma String tivesse, por exemplo, um método chamado getVowelsCount() que nos devolvesse esse valor, o que você acha?

Bom, apesar de tais funcionalidades não existirem por padrão na String, isto pode ser conseguido utilizando extensões. É exatamente isso que faremos no exemplo abaixo:

```
import Foundation
let name = "Steven Paul Jobs"

extension String {
    var initials: String {
        let words = self.components(separatedBy: " ")
        let firstLetters = words.map({String($0.first!)})
        return firstLetters.joined()
    }

    func getVowelsCount() -> Int {
        var total: Int = 0
        let characterArray = Array(self)
```

```
        for letter in characterArray {
            let lowerLetter = String(letter).lowercased()
            switch lowerLetter {
            case "a", "e", "i", "o", "u":
                total += 1
            default:
                break
            }
        }
        return total
    }
}

print("O nome \(name) tem \(name.getVowelsCount()) vogais")
//O nome Steven Paul Jobs Brito tem 5 vogais

print("As iniciais de \(name) são \(name.initials)")
//As iniciais de Steven Paul Jobs são SPJ
```

Código-fonte 74 – Extensions
Fonte: Elaborado pelo autor (2019)

No código anterior, criamos uma extensão para o tipo String. Isto é feito utilizando a palavra reservada extension, seguida do tipo que queremos estender. Agora, qualquer método ou propriedade computada (não é possível criar propriedade armazenada através de extension) que forem definidos nesse escopo farão parte do tipo String e poderão ser utilizados em qualquer String criada no código.

Primeiro criamos uma variável computada chamada **initials** que cria um array (words) composto por todas as palavras presentes na String. Isso é feito através do método componentes(separatedBy:), que utiliza a String passada no parâmetro como elementos separados. Com isso, nosso array words irá conter as palavras “Steven”, “Paul” e “Jobs”. Depois utilizamos o método map nesse array para gerar um novo array (**firstLetters**) formado pela primeira letra de cada palavra e, por fim, retornamos a String gerada pela chamada do método joined() nesse array. O método joined() serve para combinar todas as Strings de um Array formando uma só. Com isso, temos agora uma variável (initials) que nos retorna as iniciais do nome presente naquela String.

Na segunda parte, foi criado um método chamado **getVowelsCount()**, que cria uma variável **total** (que armazenará o total de vogais) e um array de Character (**characterArray**), construído da transformação da String (**self**) em Array (**Array(self)**). Esse array é varrido e é verificado se cada uma das letras (transformada em String e em lowercase) for igual a “a”, “e”, “i”, “o” ou “u”. Se for, significa que a letra

é uma vogal e a propriedade **total** é incrementada. Ao final, retornamos total que conterá o total de vogais presentes na String.

Veja que, mesmo sem termos criado o tipo String, através de **extension**, nós podemos dar novas funcionalidades a esse tipo. O mesmo é válido para tipos (sejam classes ou structs) que você mesmo cria.

### 4.7 Protocol (protocolo)

Protocolo é um recurso disponível em *Swift* que nos permite criar um conjunto de regras e especificações (métodos, propriedades e outros requerimentos) que um tipo (classe, struct ou enum) precisa implementar para que esteja em conformidade com determinado protocolo. Com essa técnica, podemos obrigar determinados tipos, ao serem criados ou estendidos, implementem determinadas funcionalidades. Quando um tipo implementa todas as diretrizes definidas por um protocolo, dizemos que está em conformidade com aquele protocolo.

Uma das vantagens de utilizar esse recurso é que vários tipos diferentes podem ser tratados como sendo do mesmo tipo (mesmo protocolo) e isso nos abre um leque grande, pois sabemos que, por exemplo, uma classe pode herdar de outra classe, porém, com protocolos, nós podemos fazer com que uma classe implemente quantos protocolos desejar. Em outras linguagens, esse recurso é conhecido como **Interface**.

**DICA:** O trabalho com protocolos é tão poderoso, que abriu um novo paradigma de programação que vai além da programação orientada a objetos. Estamos falando da programação orientada a protocolos (POP).

A sintaxe de definição de um protocolo é bem parecida com a de definição de uma classe, exceto que, neste caso, usamos a palavra reservada protocol. No exemplo abaixo, para entendermos como utilizar protocolos, criaremos quatro classes (File, Gif, Png, Mov e Mp3), sendo que File é a classe mãe, que representa um arquivo, e Gif, Png, Mov e Mp3, são classes que herdam File, porém possuem construtores que definem a sua extensão. A classe Gif, por ser um formato de imagem que aceita animação, possui uma propriedade extra chamada animated.

```
class File {
    let name: String
    let ext: String
    let size: Double
    init(aName: String, anExt: String, aSize: Double) {
        name = aName
        ext = anExt
        size = aSize
    }
}

class Gif: File {
    var animated: Bool

    init(aName: String, aSize: Double, isAnimated: Bool) {
        animated = isAnimated
        super.init(aName: aName, anExt: "gif", aSize: aSize)
    }
}

class Png: File {
    init(aName: String, aSize: Double) {
        super.init(aName: aName, anExt: "png", aSize: aSize)
    }
}

class Mov: File {
    init(aName: String, aSize: Double) {
        super.init(aName: aName, anExt: "mov", aSize: aSize)
    }
}

class Mp3: File {
    init(aName: String, aSize: Double) {
        super.init(aName: aName, anExt: "mp3", aSize: aSize)
    }
}
```

Código-fonte 75 – Protocol
Fonte: Elaborado pelo autor (2019)

Sabemos que todo arquivo (**File**) possui nome (**name**), uma extensão (**ext**) e um tamanho (**size**) e, com isso, toda e qualquer classe que herde de File também terá essas propriedades. É o caso de Gif, Png, Mov e Mp3.

Com essas classes criadas, vamos fazer algumas modificações. Sabemos que os formatos de arquivo Png e Mov representam imagens/vídeos e, como tal, podem possuir também as propriedades **width** (largura) e **height** (altura), que definem seu

tamanho em pixels. Também sabemos que certos tipos de arquivos podem ter uma duração (mov, mp3 etc.), ou seja, seria ideal terem uma propriedade para armazenar tal característica. Nesse cenário, teríamos que fazer com que nossas classes herdassem, por exemplo, das classes que pudessem representar arquivos que possuem largura e altura (por exemplo, uma classe Sizeable), e outras que representassem um arquivo que possui duração (por exemplo, Playable). No entanto, alguns arquivos possuem tamanho e duração (Mov), outros somente/tamanho (Png) e outros somente duração (Mp3), portanto, trabalhar com heranças nesse caso seria bem complicado (lembrando que uma classe pode herdar outra classe). Em cenários como esse, o ideal é trabalharmos com protocolos, o que faremos abaixo. Vamos criar os protocolos Sizeable e Playable, cada um definindo o que precisa ser feito para que um tipo possa implementá-los e, dessa forma, ser tratado como tal. O código final ficará desta forma:

```
class File {
    let name: String
    let ext: String
    let size: Double
    init(aName: String, anExt: String, aSize: Double) {
        name = aName
        ext = anExt
        size = aSize
    }
}

protocol Sizeable {
    var width: Int {get set}
    var height: Int {get set}
}

protocol Playable {
    var duration: Double {get set}
    func play()
    func stop()
}


class Gif: File, Sizeable {
    var width: Int
    var height: Int
```

```
    var animated: Bool

    init(aName: String, aSize: Double, isAnimated: Bool,
        aWidth: Int, aHeight: Int) {

        width = aWidth
        height = aHeight
        animated = isAnimated
        super.init(aName: aName, anExt: "gif", aSize: aSize)
    }
}

class Png: File, Sizeable {
    var width: Int
    var height: Int

    init(aName: String, aSize: Double, aWidth: Int,
         aHeight: Int) {

        width = aWidth
        height = aHeight
        super.init(aName: aName, anExt: "png", aSize: aSize)
    }
}

class Mov: File, Sizeable, Playable {
    var width: Int
    var height: Int
    var duration: Double

    func play() {
        print("Vídeo está tocando")
    }

    func stop() {
        print("Vídeo está parado")
    }

    init(aName: String, aSize: Double, aWidth: Int,
         aHeight: Int, aDuration: Double) {

        width = aWidth
        height = aHeight
        duration = aDuration
        super.init(aName: aName, anExt: "mov", aSize: aSize)
    }
}

class Mp3: File, Playable {
    var duration: Double
```

```
    func play() {
        print("O som está tocando")
    }

    func stop() {
        print("O som está parado")
    }

    init(aName: String, aSize: Double, aDuration: Double) {

        duration = aDuration
        super.init(aName: aName, anExt: "mp3", aSize: aSize)
    }
}
```

Código-fonte 76 – Protocol
Fonte: Elaborado pelo autor (2019)

Para resolver nosso problema, elaboramos dois protocolos, o Playable (que representa um arquivo de reprodução, como áudio ou vídeo) e o Sizeable (representando arquivos que possuem tamanho em pixels). O protocolo Sizeable obriga a classe que implementá-lo a ter as propriedades width e height, ambas do tipo **Int** e que possam ser tanto recuperadas como definidas (get e set). O protocolo Playable define em suas regras que seja criada uma propriedade duration, do tipo Double, get e set, que servirá para definir a duração em segundos do arquivo. Além disso, será necessário implementar dois métodos, play() e stop(), que servem para iniciar e interromper a execução do arquivo.

Perceba que o protocolo não define o valor das propriedades e não define a implementação dos métodos, apenas sua assinatura. Essa tarefa cabe ao tipo que irá implementar esse protocolo, ou seja, a maneira como um Mov implementará o play pode ser totalmente diferente de como o Mp3 irá fazê-lo, o que é bem interessante, afinal, sabemos que todo arquivo de animação deve saber como tocar e interromper o seu conteúdo, porém cada um tem sua forma particular de executar essa ação.

Definidos os protocolos, chegou a hora de implementá-los. Para implementar um protocolo, é preciso inseri-lo (ou inseri-los) separado por vírgula, após o nome da classe mãe. Nossa classe Png implementou o protocolo Sizeable (pois é uma imagem), a classe Mov implementou Sizeble e Playable (pois é um vídeo) e Mp3 implementou Playable (é um áudio). Ao implementar o protocolo, a classe deve implementar seus respectivos métodos e variáveis e, novamente, cada uma o faz da

maneira que desejar. Isso fica bem nítido na implementação dos métodos play() e stop(), que são diferentes entre Mov e Mp3 (neste exemplo simples, apenas fizemos o texto do **print()** ser diferente, mas poderiam ser implementações bem complexas e completamente diferentes entre si).

### 4.8 Type Casting

Veja que, com o uso dos protocolos, conseguimos resolver o problema de classes com características semelhantes e ao mesmo tempo distintas entre si, porém herdando das mesmas classes. Como comentei acima, outra vantagem bem interessante é podermos tratar os tipos de maneira diferente. No nosso exemplo anterior, todos objetos são do tipo File (através de herança), ou seja, se eu criar um Array de File, poderei colocar objetos que são Mov, Mp3 etc. Veja o exemplo abaixo:

```
//Criando os arquivos
var pngFile = Png(aName: "foto", aSize: 1.5, aWidth: 2000,
aHeight: 1200)
var movFile = Mov(aName: "video", aSize: 250.0, aWidth: 1280,
aHeight: 720, aDuration: 600)
var gifFile = Gif(aName: "animado", aSize: 1.0, isAnimated:
true, aWidth: 800, aHeight: 600)
var mp3File = Mp3(aName: "musica", aSize: 3.2, aDuration:
240)

//Criando array de File
let files: [File] = [pngFile, movFile, gifFile, mp3File]
```

Código-fonte 77 – Type Casting
Fonte: Elaborado pelo autor (2019)

Criamos objetos do tipo Mov, Mp3, Gif e Png e conseguimos colocá-los dentro do array files, que é um Array de File. Agora, se formos varrer esse array, como podemos saber se determinado objeto é um Playable, por exemplo? E quando descobrirmos, como fazemos para executar o método **play()** nesse objeto? (afinal, a classe File não possui esse método).

Para que isso seja possível, precisamos primeiro verificar se a classe é um Playable e, depois de verificado, tratar a classe como tal. A técnica de tratar uma

classe como sendo outra é chamada de **Type Casting**. Para fazer isto em *Swift* utilizamos a palavra reservada **as,** que nos informa que iremos manipular um tipo como se fosse outro, nos dando acesso às características desse outro tipo. Vejamos como resolver isto no exemplo abaixo (inclua o código logo após o exemplo anterior):

```
//Varrendo o array e verificando se o arquivo é Animatable
//Caso seja, iremos rodar o arquivo. Para isso, tratamos
//como Animatable e chamamos o método play()
for file in files {

    //Abaixo, verificamos se o objeto file é do tipo
    //Animatable utilizando a palavra reservada is

    if file is Playable {

        print("Tocando o arquivo \(file.name).\(file.ext)")

        //file.play()
        //Não adianta executar da forma acima por que file é
        //do tipo File e este tipo não possui o método
        //play(). Para isso, precisamos tratá-lo como sendo
        //um Playable. O nome desta técnica é Type Casting,
        //ou apenas Cast.
        (file as! Playable).play()
    }
}
```

Código-fonte 78 – Type Casting
Fonte: Elaborado pelo autor (2019)

Utilizando a palavra reservada **is**, podemos verificar se um determinado objeto é de determinado tipo, e é isto que usamos para identificar se o file em questão é ou não um Playable. Esse tratamento nos retorna um Bool e, caso seja true, significa que file, além de ser File, também é um Playable, podendo ser tratado dessa forma. Na sequência, executamos o método play() dentro de file, porém, para isso é necessário que seja feito o Type Casting de File para Playable, que é realizado usando **as!**. Como tratar um File como sendo Playable não é uma operação que pode ser verdade em todos os casos, o uso do **as** deve ser feito utilizando **?** ao seu lado, o que gerará um **Playable?**, ou seja, um Optional de Playable. Esta seria a forma segura de fazer isto, pois assim, caso file não fosse Playable, não acarretaria erro no nosso código. No

caso do nosso exemplo, como verificamos anteriormente se file é um Playable, podemos utilizar **!** ao lado de **as**, pois temos certeza absoluta de que é um Playable, e ao utilizar **as!** estamos gerando um **Playable** e não um **Playable?**.

Agora que nosso file está sendo tratado da forma que desejamos, podemos chamar o método play() tranquilamente.

## CONCLUSÃO

No capítulo **Voe com Swift** você pôde conhecer essa linguagem robusta, moderna, simples e fácil de se trabalhar. É a linguagem definitiva no desenvolvimento dos aplicativos iOS, sendo escolhida para novos projetos de aplicativos e, gradativamente, substituindo o Objective-C em projetos mais antigos. Aprender *Swift* é o pontapé inicial e fundamental para entrar no mundo de desenvolvimento de Apps para as plataformas da Apple, e através deste módulo você conheceu, treinou e aprendeu como utilizar todos os principais recursos desta linguagem, desde a simples criação de variáveis até o uso dos recursos mais avançados, como closures, generics, protocolos e outros mais.

# REFERÊNCIAS

APPLE. **The Swift Programming Language (Swift 5.9)**. 2023. Disponível em: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/thebasics/ >. Acesso em: 28 jun. 2023.