



BEM VINDOS!

VENTURUS⁴TECH

VENTURUS⁴TECH
JULHO/16

Módulo I
Introdução a linguagem
Swift

VENTURUS⁴TECH

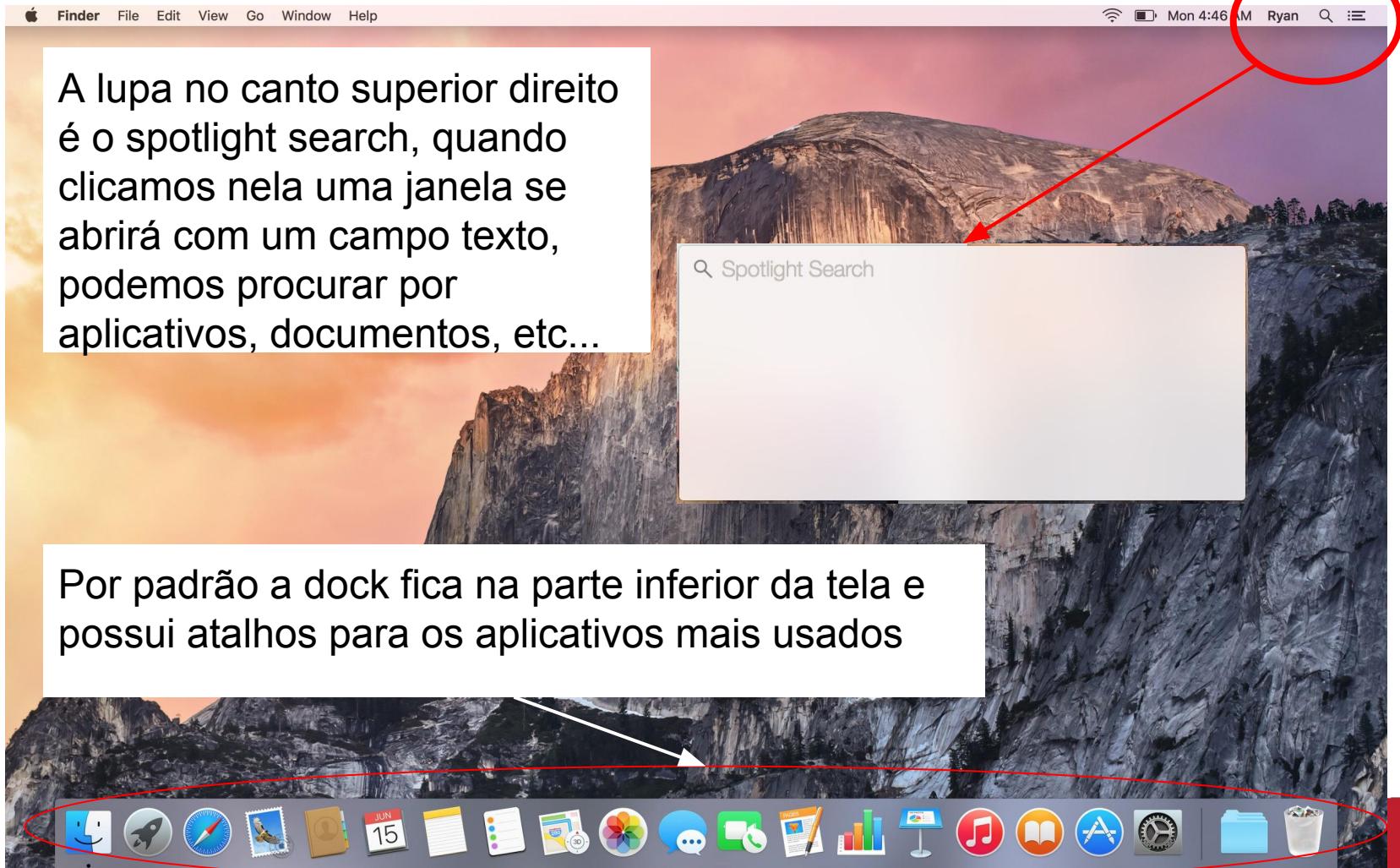
Com Rafael Munhoz
rafael.munhoz@venturus.org.br

VENTURUS⁴TECH
JULHO/16

Sistema operacional MAC OS X

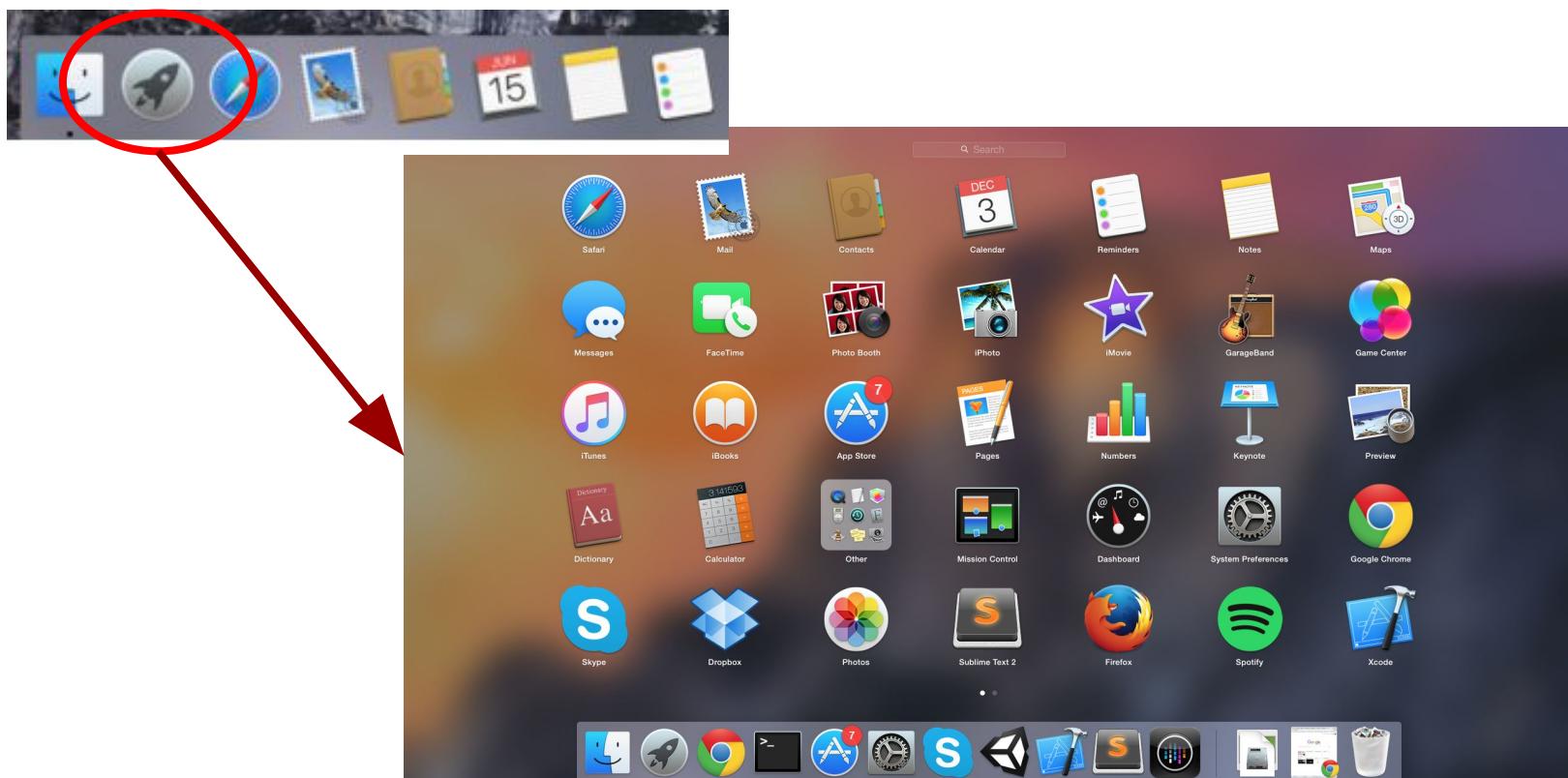
O MAC OS X é um sistema operacional de interface gráfica baseado no UNIX, roda apenas em computadores Apple.

Sistema operacional MAC OS X



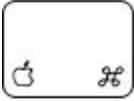
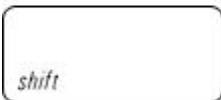
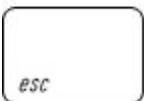
Sistema operacional MAC OS X

Este ícone leva para o launch pad que mostra todos os aplicativos que estão na pasta applications, de forma similar aos dispositivos móveis.



Sistema operacional MAC OS X

Equivalência de teclas para os menus do MAC OS X
e a tradução para as teclas do teclado do PC

Key	Name	Symbol	Abbreviation	
	Command	⌘	cmd	
	Option	⌥	opt	
	Control	⌃	ctrl	
	Shift	⇧	shft	
	Escape	⌃	esc	
	Delete	⌫	del	

Sistema operacional MAC OS X

Atalhos para copiar e colar



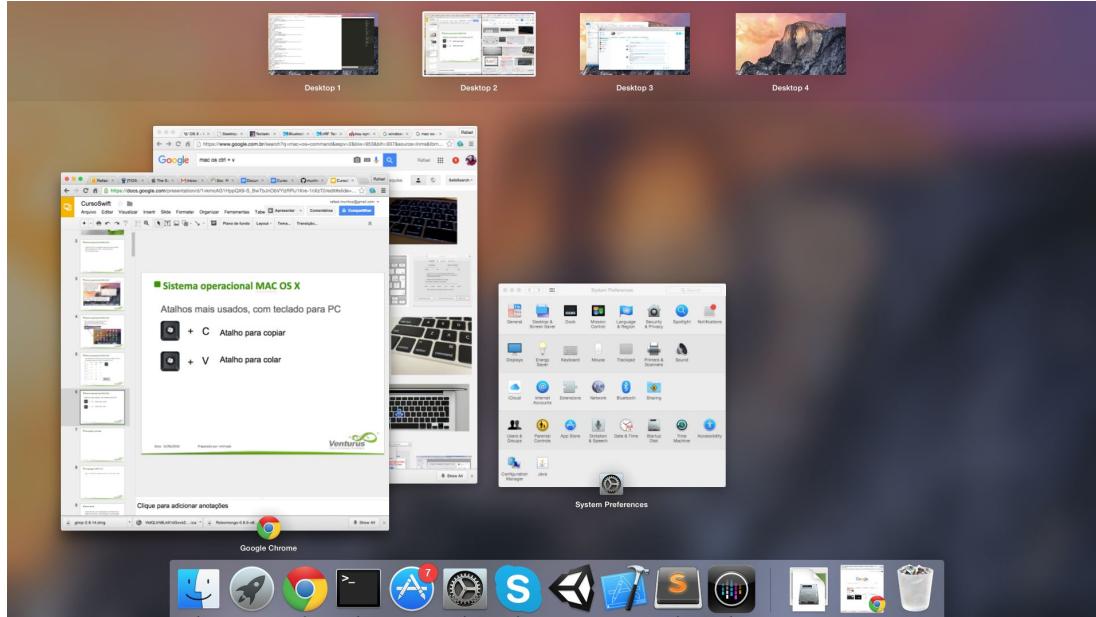
+ C Atalho para copiar



+ V Atalho para colar

Sistema operacional MAC OS X

O MAC OS X possui vários desktops, para trabalharmos com eles com o teclado do PC usamos os seguintes atalhos:



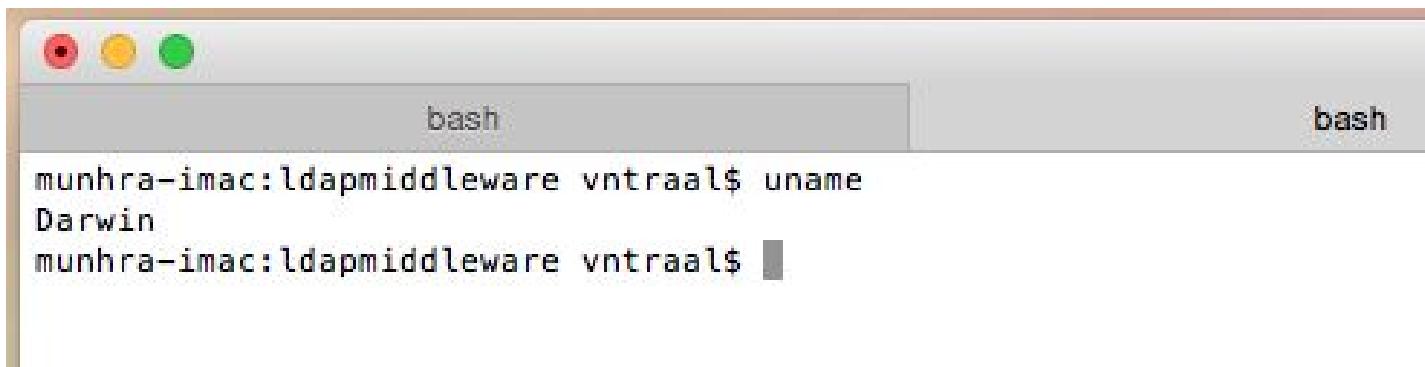
CTRL + SETA PARA CIMA,
mostra os desktops criados

CTRL + SETA PARA
ESQUERDA, navega para o
desktop da esquerda

CTRL + SETA PARA
DIREITA, navega para o
desktop da direita

Sistema operacional MAC OS X

Como o MAC OS X é baseado numa versão do UNIX chamado Darwin, os comandos do UNIX são bem similares ao Linux, para abrirmos o terminal usamos o spotlight e no campo texto procuramos por terminal.



A linguagem SWIFT 2

Swift é a nova linguagem de programação para iOS, OS X, Apple Watch e Apple TV

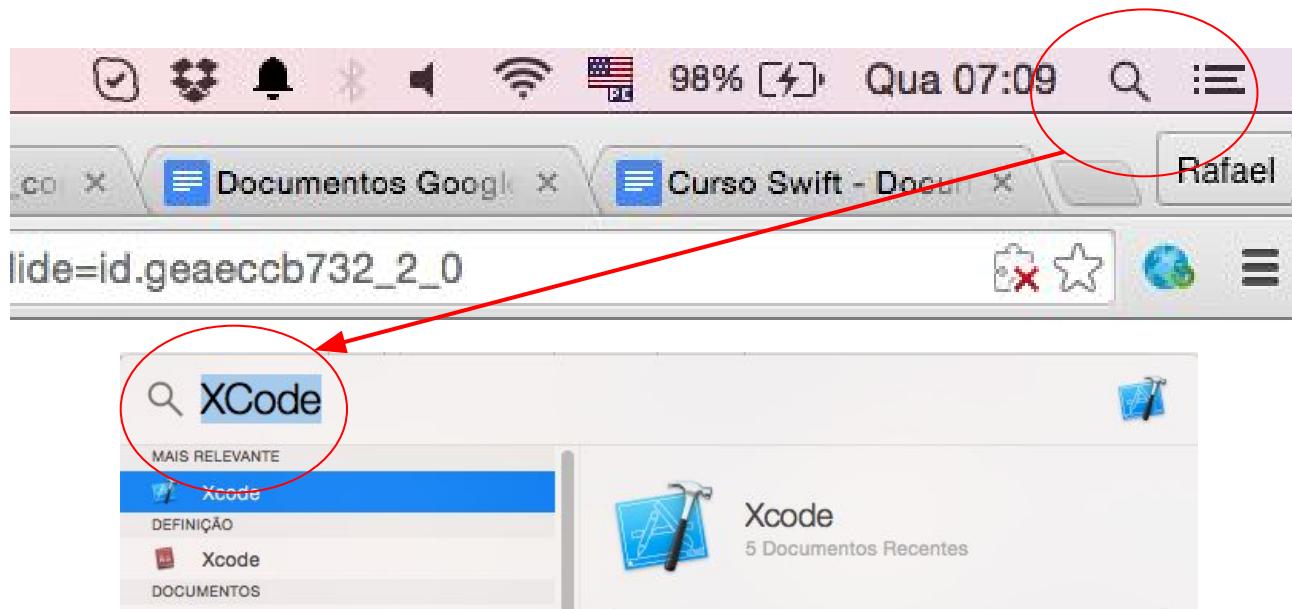
Hello World

Para termos uma visão geral do ambiente de desenvolvimento, vamos fazer uma aplicação bem simples chamada Hello World.

Vamos então abrir o XCode que é a IDE de desenvolvimento de aplicativos para os dispositivos Apple.

Hello World

Existem várias maneiras de abrir o XCode, vamos usar o Spotlight do MAC OS X, para isso clique na lupa no canto superior direito da tela, um campo texto vai ser aberto, digite XCode e aperte Enter.



Hello World

Welcome to Xcode

Version 7.1.1 (7B1005)

Get started with a playground
Explore new ideas quickly and easily.

Create a new Xcode project
Start building a new iPhone, iPad or Mac application.

Check out an existing project
Start working on something from an SCM repository.

MetalAttack
...cts/MetalAttack/MetalAttack.spritebuilder

TrackID
~/Projects/trackid-ios

FetchResultExample
~/Projects/iosfetchresultexample

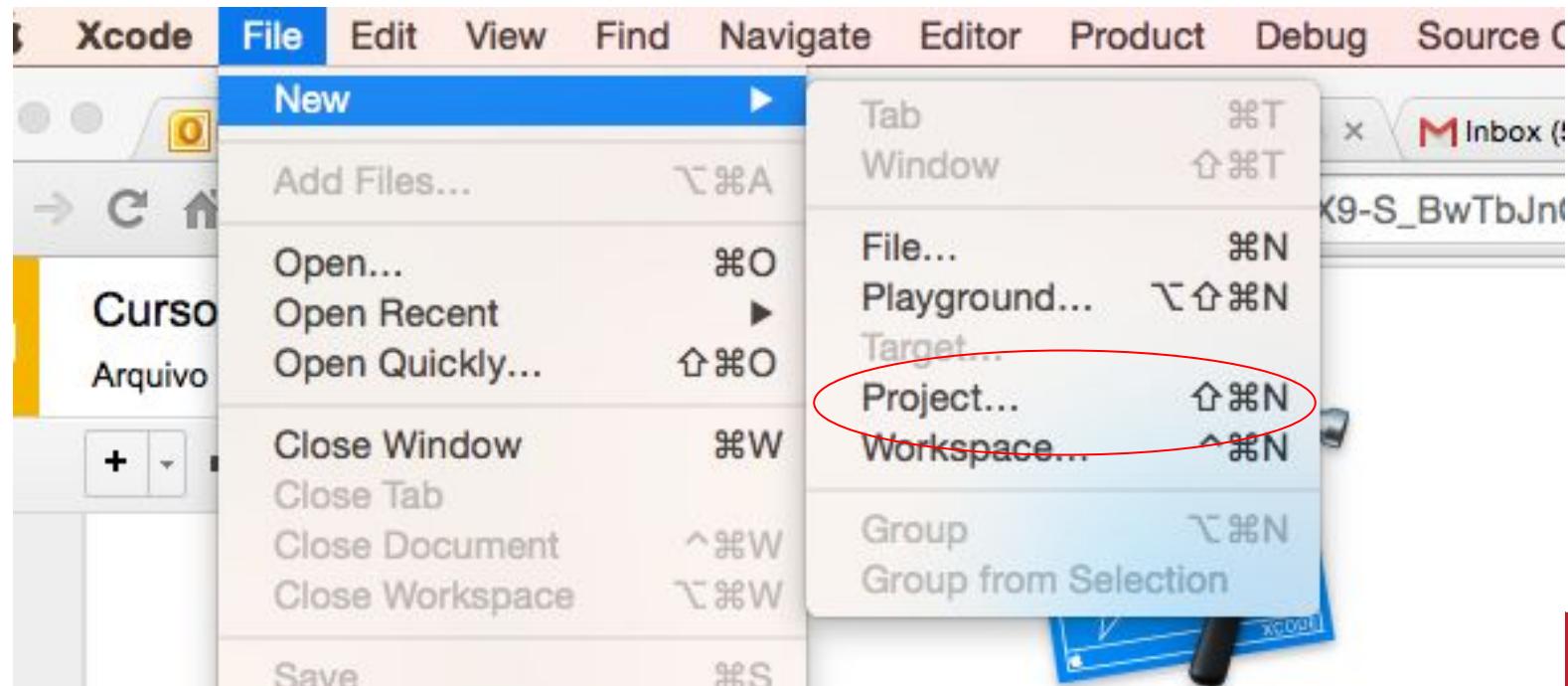
TrackID
~/Projects/oldTrackId/trackid-ios

VideoPlayer
~/Projects

Open another project...

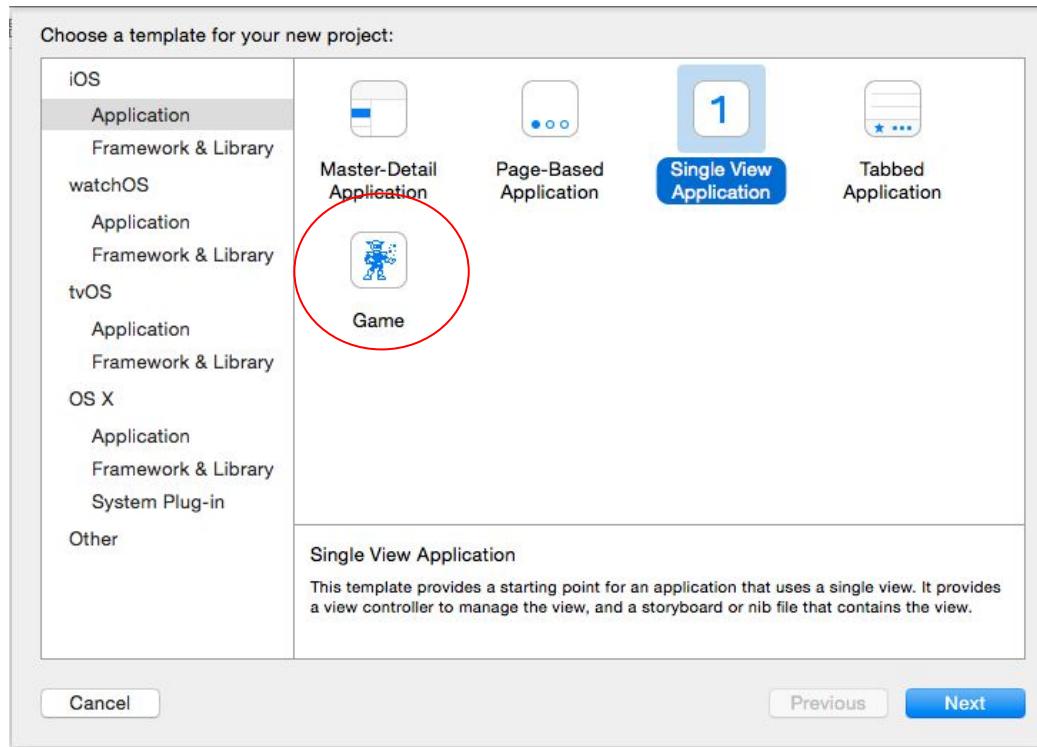
Hello World

Agora vamos criar um novo projeto,
para isso clique em File -> New -> Project



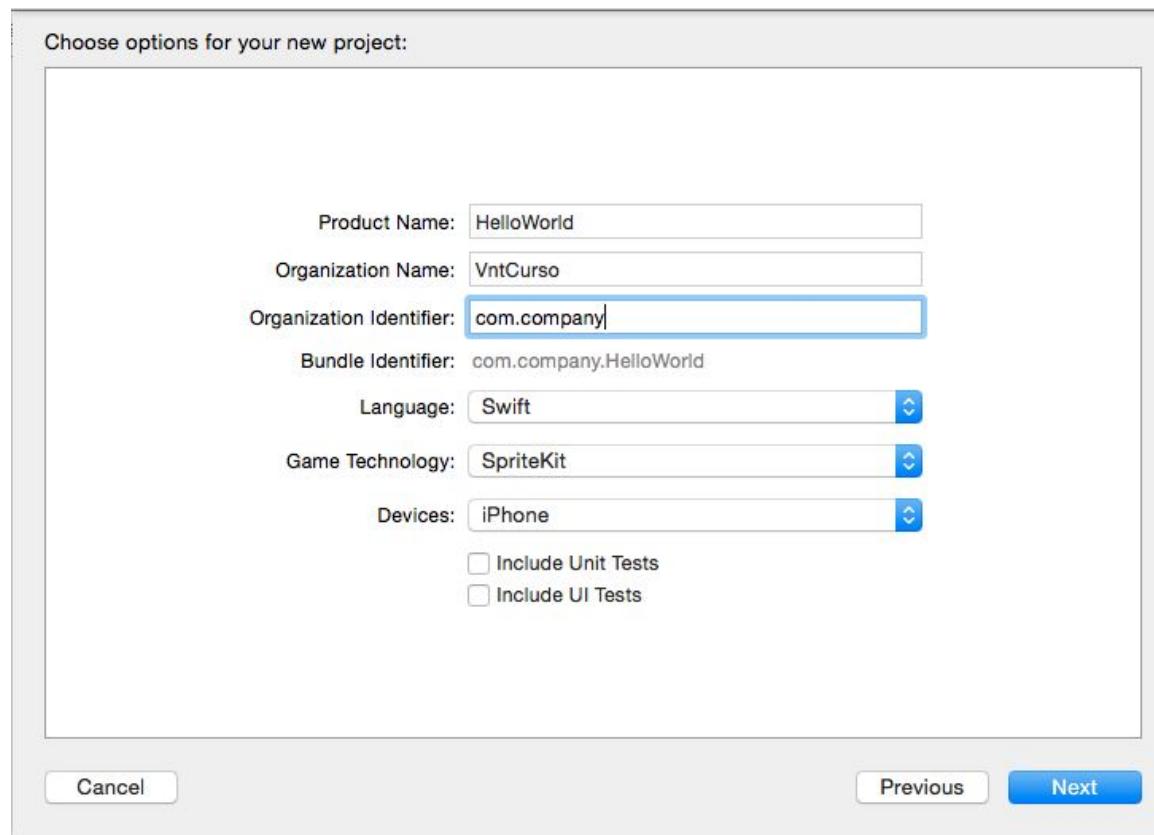
Hello World

Vamos escolher o template de projeto chamado Game, nativo do iOS para fazermos o nosso Hello World, este template importa as bibliotecas do framework SpriteKit da Apple.



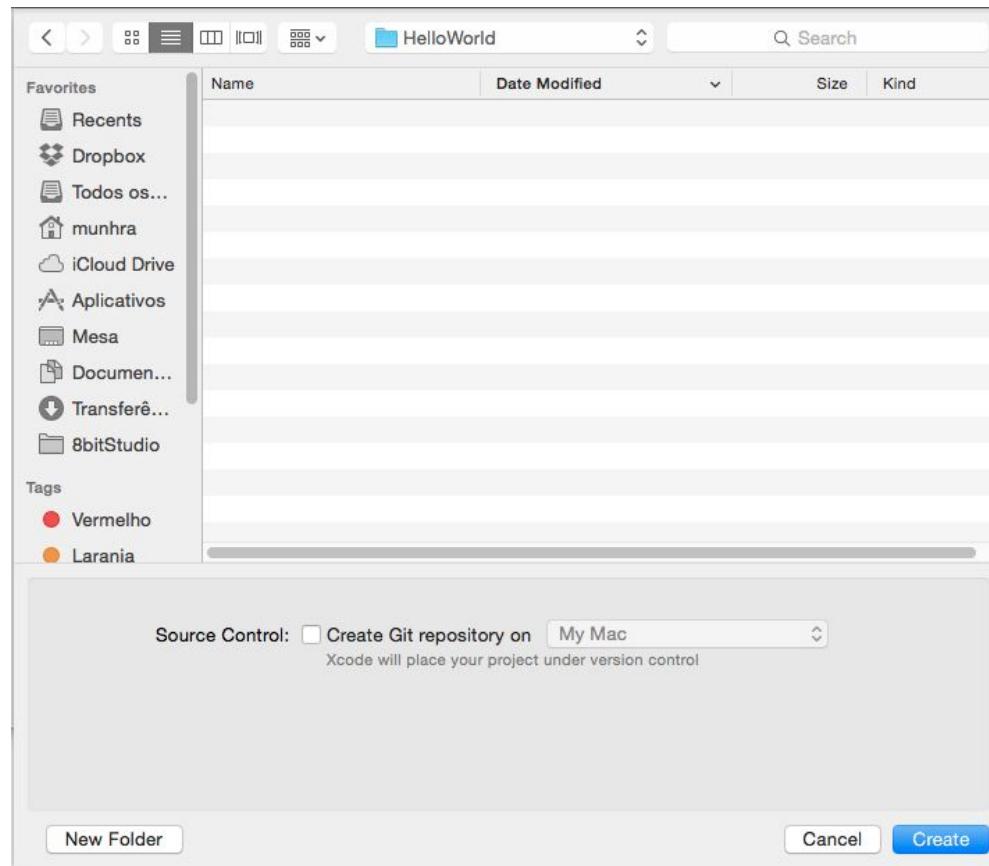
Hello World

Selecione as opções do projeto como mostrado na figura e clique em Next.



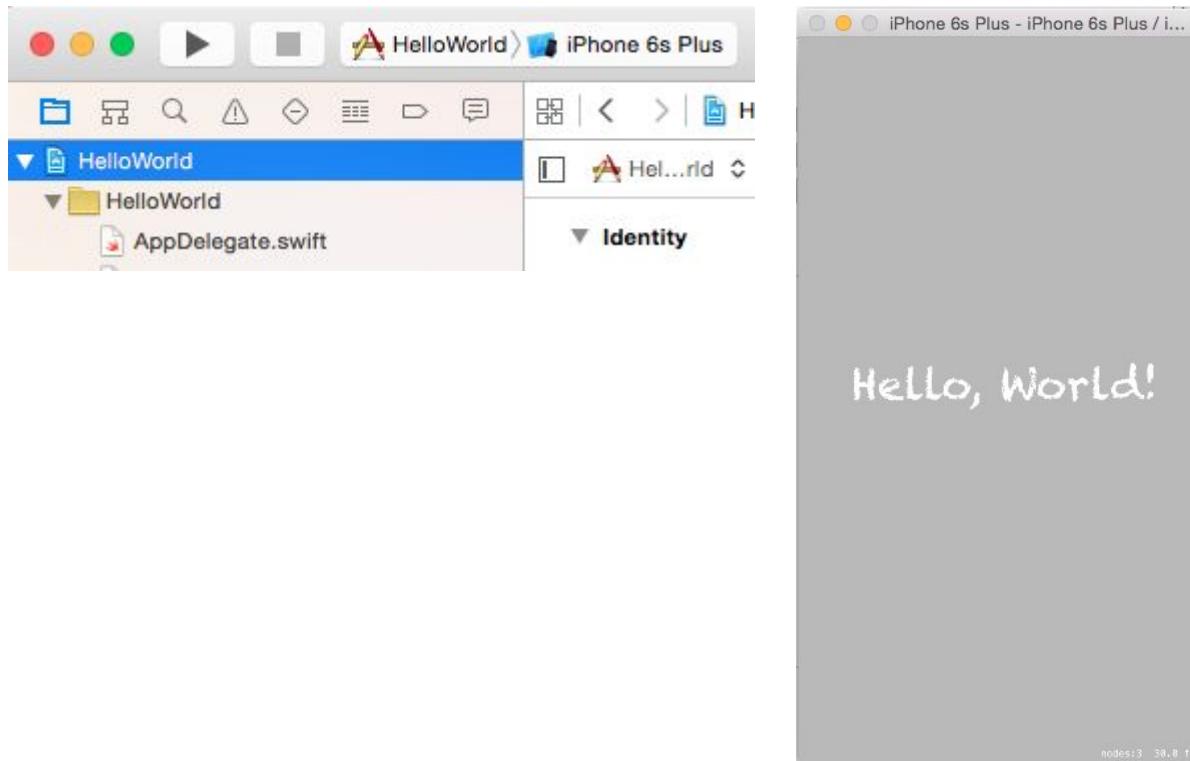
Hello World

Escolha uma pasta para salvar o projeto e clique em create.



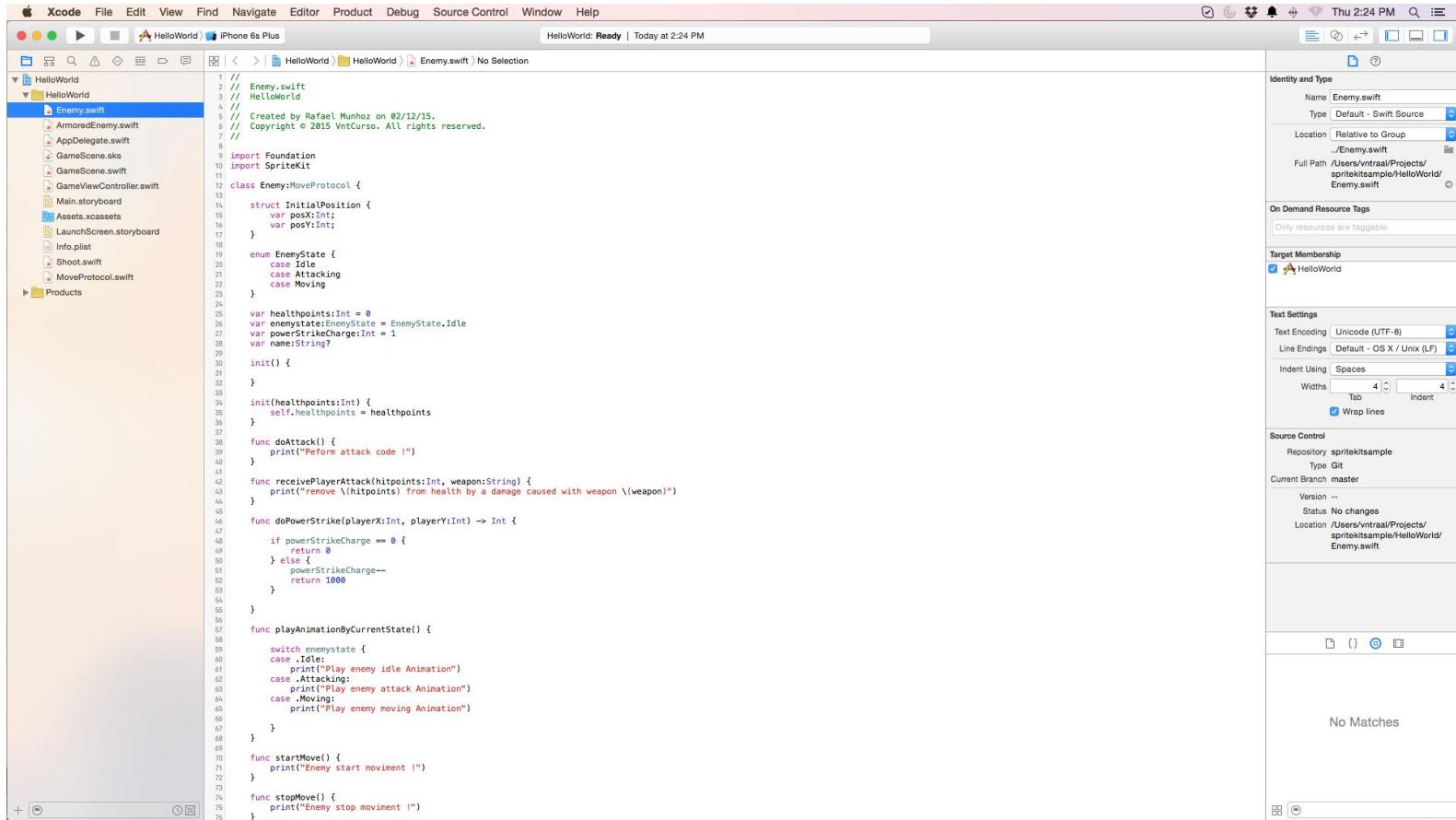
Hello World

Com o projeto criado, podemos compilar o código e rodar o aplicativo no simulador, para isso clique no botão play do XCode



HelloWorld

Principais elementos da tela do XCode



Tipos Básicos da linguagem SWIFT

- Int : variáveis deste tipo armazenam valores inteiros de -2.147.483.648 até 2.147.483.647.
- Float: variáveis deste tipo armazenam valores de 32-bit de ponto flutuante.
- Double: variáveis deste tipo armazenam valores de 64-bit de ponto flutuante.
- String: variáveis deste tipo armazenam cadeias de caracteres
- Bool: variáveis dest tipo armazenam apenas dois valores true ou false

Constantes e variáveis

Para declarar uma variável no SWIFT, usamos a palavra chave **var**, vamos declarar a variável chamada **score** do tipo **Int**, e vamos dar o valor inicial de zero.

```
import SpriteKit

class GamePlayScene : SKScene {

    override func didMoveToView(view: SKView) {

        var score:Int = 0
    }
}
```

Constantes e variáveis

Agora vamos definir a posição inicial do jogador, esse valor nunca vai mudar então vamos usar uma constante para essa definição, no SWIFT usamos a palavra chave `let`. Para definirmos a posição do jogador precisamos de duas constantes uma que representa a posição no eixo x e outra no eixo y.

```
import SpriteKit

class GamePlayScene : SKScene {

    override func didMoveToView(view: SKView) {

        var score:Int = 0
        let xPosition = 200
        let yPosition = 400

    }
}
```

Constantes e variáveis

Vamos declarar mais 3 variáveis sendo uma chamada gameover do tipo Bool, outra chamada hp do tipo float e por fim uma chamada scoretext do tipo String. E mover as outras já declaradas para o escopo de classe.

```
import SpriteKit

class GamePlayScene : SKScene {

    var score:Int = 0
    let xPosition = 200
    let yPosition = 400
    var hp:Float = 100.00
    var gameover:Bool = false
    var scoretext:String = "Score : "
}

override func didMoveToView(view: SKView) {

    print("Position x \(xPosition)")
    print("Position y \(yPosition)")

}

}
```

Nomeando variáveis e constantes

Nomes de variáveis e constantes não podem conter espaços em branco, símbolos matemáticos, setas, Unicode de uso privado, não podem começar com número.

Mostrando valores de variáveis e constantes

Para mostrarmos o valor de uma variável ou uma constante utilizamos o método print, como mostrado abaixo.

```
import SpriteKit

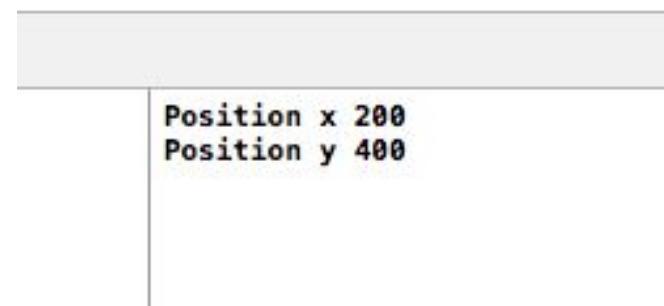
class GamePlayScene : SKScene {

    override func didMoveToView(view: SKView) {

        var score:Int = 0
        let xPosition = 200
        let yPosition = 400

        print("Position x \(xPosition)")
        print("Position y \(yPosition)")

    }
}
```



T-Uuples

Podemos agrupar vários valores em apenas um valor composto, damos a esta estrutura o nome de T-Uuples. Vamos fazer como exemplo uma T-Uple que contem o tipo de arma e o dano que ela causa.

```
import SpriteKit

class GamePlayScene : SKScene {

    var score:Int = 0
    let xPosition = 200
    let yPosition = 400
    var hp:Float = 100.00
    var gameover:Bool = false
    var scoretext:String = "Score : "
    let weapon = ("shuriken",100)

    override func didMoveToView(view: SKView) {

        print("Position x \(xPosition)")
        print("Position y \(yPosition)")

    }

}
```

T-Uuples

O código abaixo imprime o conteúdo da T-Uple weapon criada anteriormente.

```
override func didMoveToView(view: SKView) {  
  
    print("Position x \(xPosition)")  
    print("Position y \(yPosition)")  
    print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
  
}
```

Comentando Código

Comentários de linha simples são marcados com // e comentários de bloco são marcados com /* comentário */

```
override func didMoveToView(view: SKView) {  
    /*  
        print("Position x \(xPosition)")  
        print("Position y \(yPosition)")  
        print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
    */  
}
```

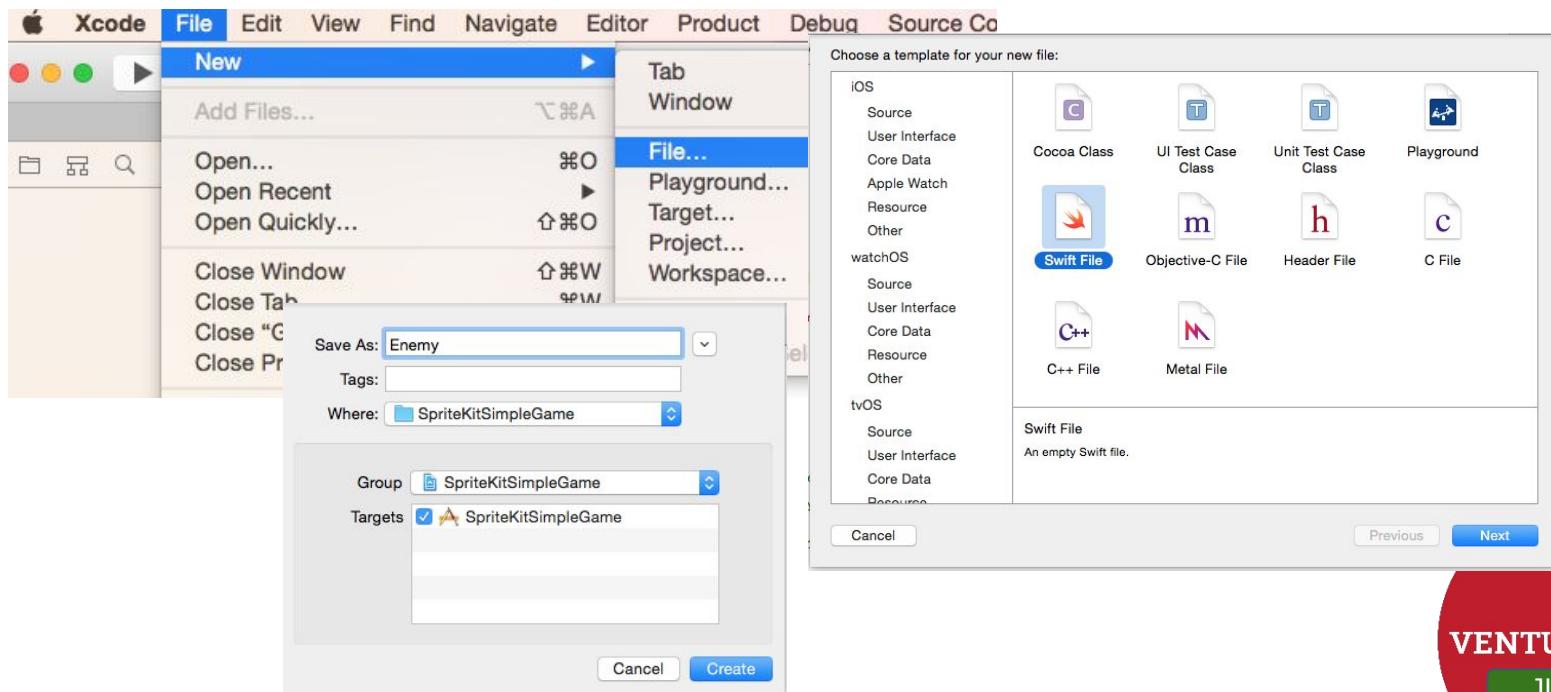
Convertendo Float para Int

Para converter uma variável de Float para Int usamos o método Int().

```
override func didMoveToView(view: SKView) {  
    var integerhp = Int(hp)  
    print("Position x \(xPosition)")  
    print("Position y \(yPosition)")  
    print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
    print("HP integer part \(integerhp)")  
}
```

Classes

Classes, são construções flexíveis e podem ser consideradas os blocos básicos de construção do código da aplicação. Como Exemplo vamos criar uma classe chamada Enemy para o nosso jogo.



Classes

Após criado o arquivo que vai conter o código da classe que vamos criar teremos, apenas uma linha fazendo o import da biblioteca Foundation que contem as classes básicas para o desenvolvimento iOS. Vamos então definir a classe Enemy usando a palavra chave **class**.

```
import Foundation

class Enemy {

}
```

Classes properties (atributos)

Propriedades associam valores a uma determinada classe ou instancia de classe, vamos definir a propriedade healthpoints para a classe Enemy da seguinte maneira

```
import Foundation

class Enemy {
    var healthpoints:Int = 0
}
```

Criando uma instância da classe Enemy

Vamos agora criar um inimigo para o jogo, para isso vamos declarar uma variável chamada enemy do tipo Enemy e nela vamos armazenar uma referência para uma nova instância da classe Enemy.

```
class GamePlayScene : SKScene {  
  
    var score:Int = 0  
    let xPosition = 200  
    let yPosition = 400  
    var hp:Float = 100.99  
    var gameover:Bool = false  
    var scoretext:String = "Score : "  
    let weapon = ("shuriken",100)  
    let enemy:Enemy = (Enemy())  
  
    override func didMoveToView(view: SKView) {  
        var integerhp = Int(hp)  
        print("Position x \(xPosition)")  
        print("Position y \(yPosition)")  
        print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
        print("HP integer part \(integerhp)")  
    }  
}
```

Acessando uma propriedade da instância

Para acessarmos uma propriedade de instância usamos o operador `". "` no código abaixo vamos acessar a propriedade **healthpoints** da instância **enemy**.

```
override func didMoveToView(view: SKView) {  
    var integerhp = Int(hp)  
    print("Position x \(xPosition)")  
    print("Position y \(yPosition)")  
    print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
    print("HP integer part \(integerhp)")  
    print("Enemy healthPoints \(enemy.healthpoints)")  
}
```

```
Position x 200  
Position y 400  
Weapon Type 100 Weapon Damage shuriken  
HP integer part 100  
Enemy healthPoints 0
```

Palavra chave self

Usamos a palavra chave self para fazer referencia a própria instância da Classe.

Class initialization

Toda classe possui o um inicializador básico implementado no método `init()`, vamos agora construir outro inicializador que vai receber o valor do `healthpoints` como parâmetro e inicializar essa instância com esse valor.

```
class Enemy {  
    var healthpoints:Int = 0  
  
    init() {}  
  
    init(healthpoints:Int) {  
        self.healthpoints = healthpoints  
    }  
}
```

Class initialization

Agora que construímos o inicializador parametrizado da classe Enemy, vamos usa-lo quando criarmos uma nova instância de Enemy chamada enemy2.

```
class GamePlayScene : SKScene {  
  
    var score:Int = 0  
    let xPosition = 200  
    let yPosition = 400  
    var hp:Float = 100.99  
    var gameover:Bool = false  
    var scoretext:String = "Score : "  
    let weapon = ("shuriken",100)  
    let enemy:Enemy = Enemy()  
    let enemy2:Enemy = Enemy(healthpoints: 200)  
  
    override func didMoveToView(view: SKView) {  
        var integerhp = Int(hp)  
        print("Position x \(xPosition)")  
        print("Position y \(yPosition)")  
        print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
        print("HP integer part \(integerhp)")  
        print("Enemy healthPoints \(enemy.healthpoints)")  
        print("Enemy 2 healthPoints \(enemy2.healthpoints)")  
    }  
}
```

Enumerations (enum)

Um enum define um tipo comum para um grupo de valores relacionados. Podemos pensar em enum como sendo um nome relacionado a um conjunto de valores inteiros.

Vamos usar o conceito de enum para montar os estados do inimigo na classe Enemy.

Enumerations (enum)

Vamos então criar um **enum** na classe **Enemy** que contemple os seguintes estados do inimigo: **Idle**, **Moving**, **Attacking**, além disso vamos criar uma variável chamada **enemystate** que irá armazenar o estado atual do inimigo.

```
class Enemy {  
  
    enum EnemyState {  
        case Idle  
        case Attacking  
        case Moving  
    }  
  
    var healthpoints:Int = 0  
    var enemystate:EnemyState = EnemyState.Idle  
  
    init() {  
    }  
  
    init(healthpoints:Int) {  
        self.healthpoints = healthpoints  
    }  
}
```

Enumerations (enum)

Agora vamos acessar a propriedade enemystate e imprimir o seu valor.

```
override func didMoveToView(view: SKView) {  
    var integerhp = Int(hp)  
    print("Position x \(xPosition)")  
    print("Position y \(yPosition)")  
    print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
    print("HP integer part \(integerhp)")  
    print("Enemy healthPoints \(enemy.healthpoints)")  
    print("Enemy 2 healthPoints \(enemy2.healthpoints)")  
    print("Enemy state \(enemy.enemystate)")  
}
```

```
Position x 200  
Position y 400  
Weapon Type 100 Weapon Damage shuriken  
HP integer part 100  
Enemy healthPoints 0  
Enemy 2 healthPoints 200  
Enemy state Idle
```

Structures

Classes e Structures são muito semelhantes, tendo como principal diferença a capacidade de herança, que só as Classes possuem. As Structures também são armazenadas como valor nas variáveis e não por referência como as instâncias, sendo assim não ficam no heap. Vamos criar a Struct InitialPosition como exemplo:

```
import Foundation

class Enemy {

    struct InitialPosition {
        var posX:Int = 0
        var posY:Int = 0
    }
}
```

Métodos

Métodos são funções associadas a um tipo (classe, struct ou enum) em particular, podemos resumir os métodos em 3 tipos:

- Métodos sem parâmetros e sem retorno.
- Métodos com um ou mais parâmetros.
- Métodos com um ou mais parâmetros e que retornam um valor.

Métodos

Métodos sem parâmetros e sem retorno, como exemplo vamos criar um método na classe Enemy chamado doAttack(), este método fará com que o inimigo ataque.

```
class Enemy {  
    struct InitialPosition {  
        var posX:Int = 0  
        var posY:Int = 0  
    }  
  
    enum EnemyState {  
        case Idle  
        case Attacking  
        case Moving  
    }  
  
    var healthpoints:Int = 0  
    var enemystate:EnemyState = EnemyState.Idle  
  
    init() {  
    }  
  
    init(healthpoints:Int) {  
        self.healthpoints = healthpoints  
    }  
  
    func doAttack() {  
        print("Perform attack code !")  
    }  
}
```

Métodos

Métodos com um ou mais parâmetros, vamos escrever um método chamado receiveHitFromPlayer, este método vai receber dois parâmetros, um deles é um Int com o valor do dano e o outro uma String que será o nome da arma.

```
func doAttack() {  
    print("Perform attack code !")  
}  
  
func receivePlayerAttack(hitpoints:Int, weapon:String) {  
    print("remove \(hitpoints) from health by a damage caused with weapon \(weapon)")  
}
```

Métodos

Métodos com um ou mais parâmetros e que retornam um valor, agora vamos definir um desses métodos na classe Enemy, que recebe a posição do jogador como parâmetro e retorna o dano especial que o inimigo vai causar, vamos chamar esse método de doPowerStrike.

```
func doAttack() {  
    print("Perform attack code !")  
}  
  
func receivePlayerAttack(hitpoints:Int, weapon:String) {  
    print("remove \(hitpoints) from health by a damage caused with weapon \(weapon)")  
}  
  
func doPowerStrike(playerX:Int, playerY:Int) -> Int {  
    return 1000  
}
```

Métodos

Para executarmos os métodos fazemos as seguintes chamadas utilizando o operador ":".

```
override func didMoveToView(view: SKView) {  
    var integerhp = Int(hp)  
    print("Position x \(xPosition)")  
    print("Position y \(yPosition)")  
    print("Weapon Type \(weapon.1) Weapon Damage \(weapon.0)")  
    print("HP integer part \(integerhp)")  
    print("Enemy healthPoints \(enemy.healthpoints)")  
    print("Enemy 2 healthPoints \(enemy2.healthpoints)")  
    print("Enemy state \(enemy.enemystate)")  
    enemy.doAttack()  
    enemy.receivePlayerAttack(200, weapon: "Sword")  
    print("\(enemy.doPowerStrike(xPosition, playerY: yPosition))")  
}
```

Perform attack code !
remove 200 from health by a damage caused with weapon Sword
1000

Collections Types - Array

Um Array armazena valores do mesmo tipo em uma lista ordenada, um mesmo valor pode aparecer várias vezes na mesma lista em posições diferentes, para o nosso exemplo vamos criar um array onde iremos armazenar os inimigos da cena.

```
class GamePlayScene : SKScene {  
  
    var score:Int = 0  
    let xPosition = 200  
    let yPosition = 400  
    var hp:Float = 100.99  
    var gameover:Bool = false  
    var scoretext:String = "Score : "  
    let weapon = ("shuriken",100)  
    let enemy:Enemy = Enemy()  
    let enemy2:Enemy = Enemy(healthpoints: 200)  
    var enemiesList:Array<Enemy> = Array<Enemy>()
```

Collections Types - Array

Agora vamos adicionar os dois inimigos que já temos, no Array de inimigos, para isso vamos usar o método `append()`.

```
enemiesList.append(enemy)
enemiesList.append(enemy2)|
```

Collections Types - Array

Para acessar um valor no array passamos a posição desejada entre colchetes [] na frente do nome do array, veja no exemplo do array enemiesList.

```
print("Health points of enemy at 0 \EnemiesList[0].healthpoints")
print("Health points of enemy at 1 \EnemiesList[1].healthpoints")
```

```
----
Health points of enemy at 0 0
Health points of enemy at 1 200
```

Collections Types - Array

Para remover um valor do array usamos o método `removeAtIndex()`, e passamos a posição do objeto que queremos remover, vamos remover o objeto na posição 0 .

```
enemiesList.removeAtIndex(0)
print("Health points of enemy at 0 \\" + enemiesList[0].healthpoints + ")")
```

```
Health points of enemy at 0 0
Health points of enemy at 1 200
Health points of enemy at 0 200
```

Collections Types - Array

Para sabermos quantos elementos temos no array usamos a propriedade count, vamos verificar quantos inimigos sobraram no array enemiesList.

```
print("Enemies left \n(enemiesList.count) ")
```

```
Health points of enemy at 0 0
Health points of enemy at 1 200
Health points of enemy at 0 200
Enemies left 1
```

Operadores Lógicos

- NOT (!)
- AND (a && b)
- OR (a || b)

Operadores Aritiméticos

- Adição (+)
- Subtração (-)
- Multiplicação (*)
- Divisão (/)
- Resto (%)

Operadores de comparação

- Igual ($==$)
- Não igual ($!=$)
- Maior que ($a > b$)
- Menor que ($a < b$)
- Maior ou igual ($a \geq b$)
- Menor ou igual ($a \leq b$)

Controle de fluxo

As principais declarações para controle de fluxo na linguagem SWIFT são:

- Condisional **if**
- Condisional **switch**
- Loop com **for**
- Loop com **While**

Condicional if

Vamos supor que o inimigo do nosso jogo possua apenas uma carga para um ataque poderoso chamado power strike, para controlar o power strike vamos criar uma propriedade do tipo Int chamada powerStrikeCharge na classe Enemy e inicializa-la com o valor 1.

```
class Enemy {  
  
    struct InitialPosition {  
        var posX:Int;  
        var posY:Int;  
    }  
  
    enum EnemyState {  
        case Idle  
        case Attacking  
        case Moving  
    }  
  
    var healthpoints:Int = 0  
    var enemystate:EnemyState = EnemyState.Idle  
    var powerStrikeCharge:Int = 1
```

Condicional if

A definição básica do if em SWIFT é a seguinte:

```
if [condição] {  
    [declaração executada se condição for verdade]  
}
```

Podemos ter também a definição do if com else:

```
if [condição] {  
    [declaração executada se condição for verdadeira]  
}else{  
    [declaração executada se condição for falsa]  
}
```

Condicional if

Agora, voltemos a implementação do power strike para o nosso inimigo, vamos definir que o inimigo só pode atacar com o power strike, se ele possuir a carga desta arma, vamos então criar uma condição no método doPowerStrike da classe Enemy.

Condicional if

O código dessa implementação ficará assim:

```
func doPowerStrike(playerX:Int, playerY:Int) -> Int {  
  
    if powerStrikeCharge == 0 {  
        return 0  
    } else {  
        powerStrikeCharge--  
        return 1000  
    }  
  
}
```

Condicional if

Vamos agora chamar o método doPowerStrike duas vezes e verificar que na segunda vez o dano causado pelo inimigo será zero pois a munição da arma acabou.

```
print("Power strike damage \\"+enemy.doPowerStrike(xPosition, playerY:yPosition)+"")  
print("Power strike damage \\"+enemy.doPowerStrike(xPosition, playerY:yPosition)+"")
```

```
Power strike damage 1000  
Power strike damage 0
```

Condicional switch

Vamos supor que queremos controlar as animações do inimigo para cada um dos estados, por exemplo, queremos tocar uma animação de movimento quando o inimigo se mover e outra quando o inimigo estiver parado e assim para os demais estados, a melhor estrutura condicional para este tipo de problema é o switch.

Condicional switch

Uma declaração switch faz uma comparação entre um valor com vários outros possíveis valores e no caso de uma comparação de sucesso o código referente ao caso será executado, veja a definição da declaração switch abaixo.

```
switch [valor para comparar] {  
  
    case [value 1]:  
        [declaração para o caso do valor 1]  
    case [value 2],[value 2]:  
        [declaração para o caso do valor 2 ou 3]  
    default:  
        [declaração para o caso de num valor acima atender]  
}
```

Condicional switch

Agora, voltemos ao nosso problema, usamos o código abaixo para implementar o controle de animação usando a declaração **switch**.

```
func playAnimationByCurrentState() {  
  
    switch enemystate {  
        case .Idle:  
            print("Play enemy idle Animation")  
        case .Attacking:  
            print("Play enemy attack Animation")  
        case .Moving:  
            print("Play enemy moving Animation")  
  
    }  
}
```

Condicional switch

Vamos agora testar o método criado anteriormente:

```
enemy.playAnimationByCurrentState()  
enemy.enemystate = Enemy.EnemyState.Attacking  
enemy.playAnimationByCurrentState()
```

Play enemy idle Animation
Play enemy attack Animation

Loop com for

Vamos supor agora que queremos criar 10 inimigos para o nosso nível atual, seria inviável escrever 10 linhas, cada uma instanciando um objeto da classe Enemy, para isso vamos usar a declaração **for**.

Loop com for

A definição do for em swift é a seguinte:

```
for [inicialização] ; [condição] ; [incremento] {  
    [declarações que serão repetidas]  
}
```

Loop com for

Usamos o código abaixo para criar os 10 inimigos do nível. Este código será implementado na classe GameScene.

```
func createEnemies(quantity:Int) {  
    for (var i = 1 ; i <= quantity ; i++) {  
        enemiesList.append(Enemy())  
    }  
}
```

Loop com for

Vamos agora executar o método createEnemies e verificar quantos objetos temos no array enemiesList:

```
createEnemies(10)
print("Created \u2019enemiesList, count) new enemies")
```

```
Created 10 new enemies
```

Loop com While

Uma outra instrução de repetição é o Loop com While:

```
while [condição] {  
    [código que será repetido até que a  
    condição seja falsa]  
}
```

Loop com While

Vamos agora criar os 10 inimigos, mas desta vez vamos usar o loop while.

```
func createEnemies(quantity:Int) {  
    var enemyCount:Int = 1  
  
    while (enemyCount <= quantity) {  
        enemiesList.append(Enemy())  
        enemyCount = enemyCount + 1  
    }  
}
```

```
Enemy start moviment !  
Shoot started to move  
Created 10 enemies
```

Optionals

Optionals em SWIFT é um conceito usado em situações onde não sabemos se um valor de uma variável existe ou não. Portanto uma variável optional pode ter valor **nil** ou não.

Suponha que nossa classe `Enemy` tenha uma propriedade `name`, para o nome do inimigo, mas não podemos garantir se a essa propriedade será dado um valor ou não.

Optionals

Para declararmos uma variável como sendo **Optional** usamos o simbolo "?" na frente do tipo da variável, como mostrado abaixo na declaração da propriedade name:

```
var healthpoints:Int = 0
var enemystate:EnemyState = EnemyState.Idle
var powerStrikeCharge:Int = 1
var name:String?
```

Optionals

Agora, vamos mudar o valor da propriedade name da instância enemy2 para "zombie" e imprimir o resultado:

```
enemy2.name = "zombie"  
print("The name of enemy 2 is \u2028(enemy2.name)")
```

Perceba que a propriedade foi impressa como Optional("zombie").

```
remove 200 from health by a damage caused with weapon  
Sword  
Power strike damage 1000  
Power strike damage 0  
Play enemy idle Animation  
Play enemy attack Animation  
Created 10 new enemies  
The name of enemy 2 is Optional("zombie")
```

Mas porque não como apenas "zombie" ?

Optionals

Para acessarmos o valor de uma variável optional teremos que usar o operador "!", esse operador abre a variável optional e extrai o valor armazenado na variável.

```
enemy2.name = "zombie"  
print("The name of enemy 2 is \u2028(enemy2.name)")  
print("The name of enemy 2 is \u2028(enemy2.name!)")
```

```
Sword  
Power strike damage 1000  
Power strike damage 0  
Play enemy idle Animation  
Play enemy attack Animation  
Created 10 new enemies  
The name of enemy 2 is Optional("zombie")  
The name of enemy 2 is zombie
```

Optionals

O que será que acontece se tentarmos acessar a propriedade name da instância enemy utilizando o operador "!" ?

```
enemy2.name = "zombie"  
print("The name of enemy 2 is \u0027enemy2.name\u0027")  
print("The name of enemy 2 is \u0027enemy2.name!\u0027")  
print("The name of enemy 2 is \u0027enemy.name!\u0027")
```

Optionals

Teremos uma EXCEÇÃO, ou seja um crash na aplicação em tempo de execução pois o programa chegou em um ponto onde ele não pode continuar.

The screenshot shows the Xcode debugger interface. The top navigation bar displays the file "Enemy.swift" and the function "0 specialized _fatalErrorMessage(StaticString, StaticString, StaticString, UInt) -> ()". The left sidebar shows the project structure with "HelloWorld" selected, and various metrics like CPU, Memory, and Disk usage. The main pane shows assembly code for the crash. At line 20, there is a trap instruction (opcode=EXC_ARM_BREAKPOINT, subcode=0xe7fdefe). The assembly code is as follows:

```
0x5e6e50 <+68> bl 0x632104 ; function signature specialization <Arg[0] = Exploded, Arg[1] = Exploded, Arg[2] = Dead, Arg[3] = Dead> of Swift._fatalErrorMessage (Swift.StaticString, Swift.StaticString, Swift.StaticString, Swift.UInt) -> () (closure #2)
0x5e6e51 <+72> trap Thread 1: EXC_BREAKPOINT (code=EXC_ARM_BREAKPOINT, subcode=0xe7fdefe)
```

The assembly code continues with several instructions involving registers r0, r1, r2, and r3, and memory locations like [r0], [r1], [r2], and [r3]. The bottom right pane shows the error message: "fatal error: unexpectedly found nil while unwrapping an Optional value (lldb)".

Optionals

Por esse motivo devemos sempre verificar se a propriedade marcada como optional não está com valor nulo (nil), senão ao aplicar o operador "!" teremos um crash para isso podemos usar a seguinte estrutura, de maneira a evitar o crash.

```
if let name = enemy.name {  
    print("The value of name is \(name)")  
} else {  
    print("The value of name is not defined")  
}
```

Essa estrutura é conhecida como Optional Binding

```
Power strike damage 1000  
Power strike damage 0  
Play enemy idle Animation  
Play enemy attack Animation  
Created 10 new enemies  
The name of enemy 2 is Optional("zombie")  
The name of enemy 2 is zombie  
The value of name is not defined
```

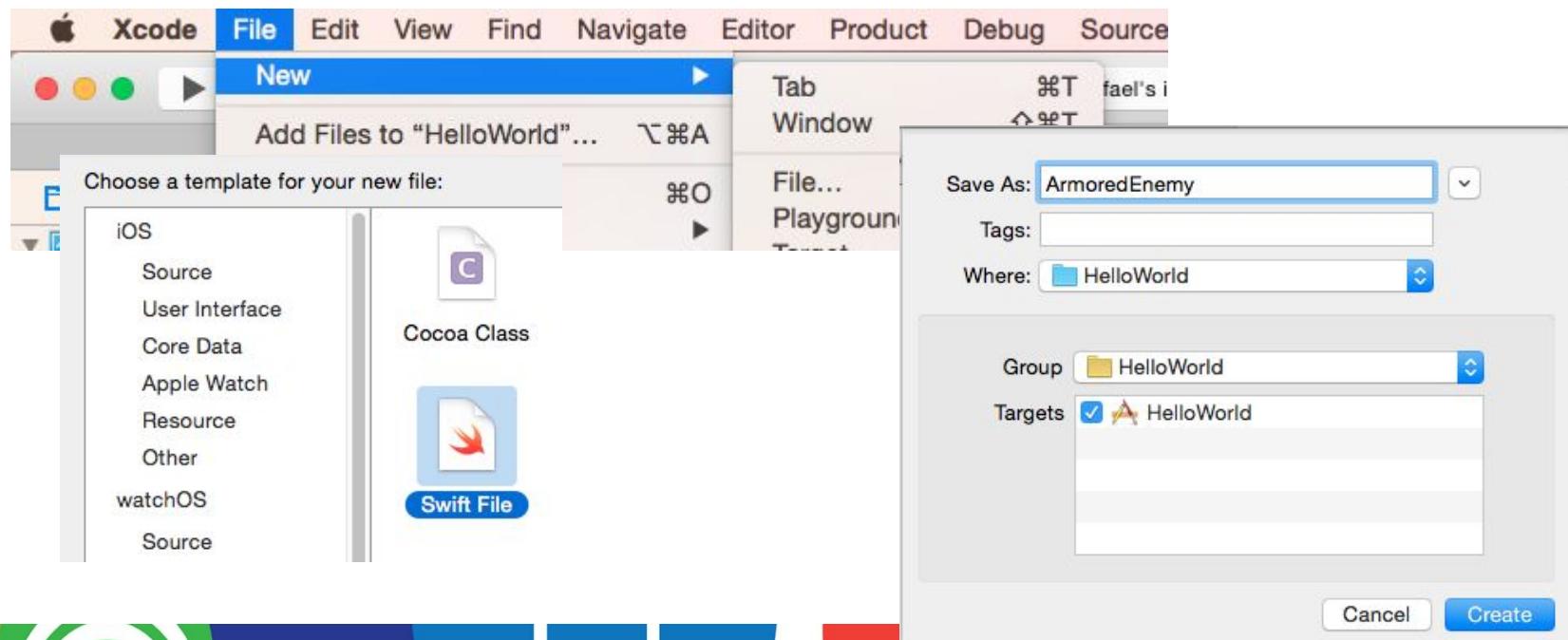
Herança

Vamos supor que agora temos um novo tipo de inimigo que possui uma **armadura** e um tipo de ataque diferente chamado **blastAttack**.

Mas que tem todo o resto das características do inimigo implementado na classe Enemy. Usaremos o conceito de herança para criarmos essa classe.

Herança

Vamos criar a classe `ArmoredEnemy` que será uma subclasse de `Enemy`. Para isso vamos criar um arquivo chamado `ArmoredEnemy.swift`.



Herança

No arquivo ArmoredEnemy vamos usar o símbolo ":" na declaração da classe para criar a relação de herança.

```
import Foundation

class ArmoredEnemy:Enemy {
    |
}
```

Herança

Qual será o resultado quando criarmos uma instância da classe ArmoredEnemy e chamarmos o método doAttack ?

```
let enemy:Enemy = Enemy()  
let enemy2:Enemy = Enemy(healthpoints: 200)  
let armoredEnemy:ArmoredEnemy = ArmoredEnemy()  
  
override func didMoveToView(view: SKView) {  
    armoredEnemy.doAttack()  
}
```

Perform attack code !

Herança

Agora vamos adicionar uma propriedade chamada **armorPoints** do tipo Int para a classe **ArmoredEnemy** e um método chamado **blastAttack**.

```
class ArmoredEnemy: Enemy {  
    var armorPoints:Int = 0  
  
    func blastAttack() {  
        print("Perform Blast Attack !")  
    }  
}
```

Herança

Vamos agora mudar o valor da propriedade armorPoints para 100 e chamar o método blastAttack da instância da classe ArmoredEnemy

```
override func didMoveToView(view: SKView) {  
    armoredEnemy.doAttack()  
    armoredEnemy.armorPoints = 100  
    armoredEnemy.blastAttack()  
    print("Armor points \(armoredEnemy.armorPoints)")  
}
```

```
Perform attack code !  
Perform Blast Attack !  
Armor points 100
```

Type casting

Vamos supor agora que temos tanto instâncias da classe Enemy quanto instâncias da classe ArmoredEnemy no array enemiesList, para isso vamos modificar o método createEnemies para adicionar um inimigo a mais do tipo ArmoredEnemy.

```
func createEnemies(quantity:Int) {  
    for (var i = 1 ; i <= quantity ; i++) {  
        enemiesList.append(Enemy())  
    }  
  
    enemiesList.append(ArmoredEnemy())  
}
```

Type casting

Agora quando chamar o método `orderAllEnemiesToAttack`, queremos que o `ArmoredEnemy` responda com o `blastAttack`, mas como iremos diferenciar um tipo de inimigo do outro ?

Type casting

É ai que entra o conceito de Type casting e para esse exemplo vamos usar o operador `as?`.

```
func orderAllEnemiesToAttack() {  
    for (var i = 0 ; i < enemiesList.count ; i++){  
  
        if let storedEnemy = enemiesList[i] as? ArmoredEnemy {  
            storedEnemy.blastAttack()  
        }else{  
            enemiesList[i].doAttack()  
        }  
    }  
  
    override func didMoveToView(view: SKView) {  
        createEnemies(10)  
        orderAllEnemiesToAttack()  
    }  
}
```

```
Peform attack code !  
Perform Blast Attack !
```

Protocols

Vamos agora criar um método na classe Enemy chamado startMove e outro chamado stopMove, estes dois métodos vão fazer com que o inimigo comece a se mover e pare de se mover respectivamente.

```
func startMove() {  
    print("Enemy start moviment !")  
}  
  
func stopMove() {  
    print("Enemy stop moviment !")  
}
```

Protocols

Agora vamos criar uma nova classe chamada Shoot, e implementar os mesmos dois métodos startMove e stopMove.

```
class Shoot {  
    func startMove(){  
        print("Shoot started to move")  
    }  
    func stopMove(){  
        print("Shoot stoped to move")  
    }  
}
```

Protocols

Temos uma semelhança de comportamento entre as duas classes: ambas possuem um método que inicia o movimento e um método que para o movimento.

Vamos então criar um protocolo chamado **MoveProtocol**.

Protocols

Criaremos um arquivo chamado MoveProtocol e escreveremos o código como mostrado abaixo:

```
protocol MoveProtocol {  
    func startMove()  
    func stopMove()  
}
```

Protocols

Agora vamos fazer com que ambas as classes Enemy e Shoot usem o protocolo MoveProtocol

```
>class Shoot:MoveProtocol {  
  
    func startMove(){  
        print("Shoot started to move")  
    }  
  
    func stopMove(){  
        print("Shoot stoped to move")  
    }  
  
}
```

```
class Enemy:MoveProtocol {  
  
    struct InitialPosition {  
        var posX:Int;  
        var posY:Int;  
    }  
  
    func startMove(){  
        print("Enemy started to move")  
    }  
  
    func stopMove(){  
        print("Enemy stoped to move")  
    }  
  
}
```

Protocols

Vamos testar o MoveProtocol, criando uma instância de Shoot e fazendo uma referência

```
override func didMoveToView(view: SKView) {  
    let shoot:Shoot = Shoot()  
    let moveable1:MoveProtocol = enemy2  
    let moveable2:MoveProtocol = shoot  
    moveable1.startMove()  
    moveable2.startMove()  
}
```

Enemy start moviment !
Shoot started to move|

Desafio final

Transforme o projeto atual em um jogo de verdade usando as seguintes classes do spritekit:



Desafio final

Para isso você vai precisar conhecer apenas 3 classes do spritekit:

SKScene : Representa a cena do jogo.

SKPhysicsContactDelegate : É um protocolo que define o tratamento de colisão entre sprites.

SKSpriteNode : É a classe que representa os sprites do jogo.

Obrigado !

VENTURUS⁴TECH

Com Rafael Munhoz
rafael.munhoz@venturus.org.br

VENTURUS⁴TECH
JULHO/16