

**Eduardo Costa** 



#### 1 - PILARES

A Programação Orientada a Objetos (POO) é um paradigma de programação que organiza e estrutura o código em torno de "objetos", que combinam dados e funcionalidades relacionadas.

Ela possui 4 pilares que vamos aprender utilizando uma dinâmica de um mundo de magia.





#### 2 - OBJETO

Primeiramente devemos saber o que é um objeto

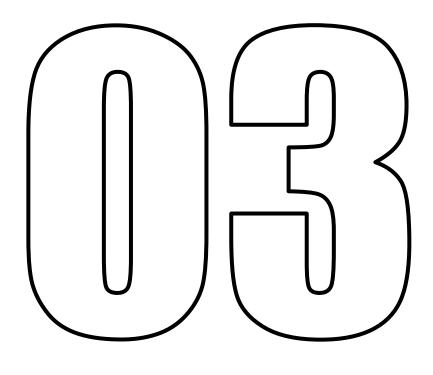
Os objetos são basicamente formas de retratar algo que possui características e ações. Como um personagem de jogo que possui nome, vida e ataque.



```
Heroi.swift

class Heroi {
   var nome: String
   var vida: Int

   func atacar(alvo: Monstro) {
      print("\(nome) ataca \(alvo.nome)!")
   }
}
```



# **ENCAPSULAMENTO**

#### 3 - ENCAPSULAMENTO

'Ninguém mais toca além de mim!'

Toda alteração de dados deve ser feita somente por métodos da própria classe, não podendo ser modificado por fora durante o código.



```
class Heroi {
    // Variáveis encapsuladas
    private var nome: String
    private var vida: Int

func receberDano(dano: Int) {
    if dano > 0 {
        vida -= dano
            print("\(nome) sofreu \(dano) de dano. Vida restante: \(vida)")
        }
    }
}
```



### 4 - HERANÇA

Que tal termos heróis diferentes?

Lembra da nossa classe "Heroi"? Podemos ter especializações que fazem tudo que os heróis fazem, e mais!



```
Class Guerreiro: Heroi {
  var arma: String
}

class Mago: Heroi {
  var poderMagico: Int
}
```



#### 5 - POLIMORFISMO

Fazem coisas iguais, mas de forma diferente!

A nossa classe "Heroi"
possui a função
"atacar" que nossa
guerreira e mago
herdaram, mas cada
um ataca de forma
diferente, né?



```
class Guerreiro: Heroi {
   var arma: String
   override func atacar(alvo: Monstro) {
        print("\(nome\) executa um ataque com sua \(arma\) em \(alvo.nome\)!")
   }
}

class Mago: Heroi {
   var poderMagico: Int
   override func atacar(alvo: Monstro) {
        print("\(nome\) lança um feitiço com um valor mágico de \(poderMagico\) em \(alvo.nome\)!")
   }
}
```



# 6 - ABSTRAÇÃO

#### Um exemplo a ser seguido

Como nosso "Heroi" é somente uma base, a classe não precisa de conteúdo em suas funções, só precisa dizer o que o guerreiro e o mago precisam saber.



```
Heroi.swift

class Heroi {
   func usarHabilidade() {
      // Método vazio a ser sobrescrito pelas subclasses
   }
}
```

# 6 - ABSTRAÇÃO

#### Um exemplo a ser seguido

Agora que eles sabem que precisar ter a função "usarHabilidade", cada um usa de seu jeito.



```
class Guerreiro: Heroi {
  override func usarHabilidade(dano: Int) {
    print("Causou \(dano) com sua habilidade!")

}
}
class Mago: Heroi {
  func usarHabilidade(qtdInimigos: Int) {
    print("Atinge \(quantidadeInimigosAtingidos) de inimigos!")
  }
}
```



### 7 - CONCLUSÃO

Vamos criar nossos personagens!

Para a criação dos nossos objetos, as classes precisam ter um inicializador.

É uma forma de criamos nossos personagens com seus atributos iniciais, nele passamos as informações essenciais que nosso personagem precisa ter, como nome e vida.

```
Mago.swift

class Mago: Heroi {
   init(nome: String, vida: Int) {
      super.init(nome: nome, vida: vida)
   }

func usarHabilidade(quantidadeInimigosAtingidos: Int) {
      print("Atinge \(quantidadeInimigosAtingidos\) de inimigos!")
   }
}
```

## 7 - CONCLUSÃO

Vamos criar nossos personagens!

Agora que sabemos como criar nossas classes, basta criar nossos objetos com elas!





```
let magoBom = Mago(nome: "Iluminado", vida: 120)
let magoMau = Mago(nome: "Multi-olhos", vida: 100)
```

# **AGRADECIMENTOS**

# Obrigado por ler até aqui!

O objetivo deste e-book é totalmente didático e foi criado como atividade de curso.

As imagens utilizadas são da empresa <u>Ankama</u> de seu jogo <u>Waven</u>, os códigos foram feitos pelo ChatGPT e modificado por mim e, para apresentação, colocados no ShowCode.

