



CRÔNICAS DO C#:  
A ARTE DO  
MONOBEHAVIOUR

VICTOR K. BAPTISTA

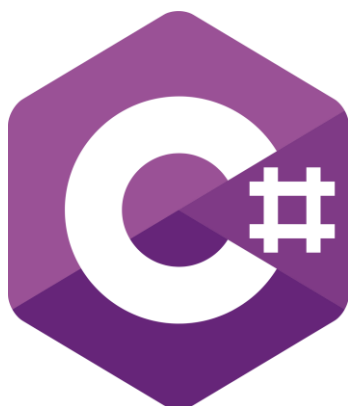


# INTRODUÇÃO AO MONOBEHAVIOR

## Componente Fundamental da Unity

O MonoBehaviour é um componente fundamental no ecossistema de desenvolvimento de jogos Unity, essencial para criar scripts de comportamentos e interações no mundo do jogo. Como uma classe C#, o MonoBehaviour forma os blocos de construção dos GameObjects do Unity, permitindo ganhar vida com funcionalidade dinâmica.

Neste eBook, vamos começar a conhecer o MonoBehaviour, explorando as suas várias funcionalidades e exemplos práticos para o ajudar a aproveitar o seu poder nos seus projetos de desenvolvimento de jogos. Independentemente de ser um principiante ou um programador experiente, dominar o MonoBehaviour é essencial para desbloquear todo o potencial da plataforma de desenvolvimento de jogos Unity.



# 01

## CONCEITOS BÁSICOS DE SCRIPTING

# 1.1 Variáveis

## Variáveis Primitivas

Neste capítulo, vamos mergulhar nos blocos de construção de qualquer linguagem de programação: variáveis e tipos de dados. Aprenda a declarar variáveis, compreenda os diferentes tipos de dados, como `int`, `float`, `string` e `bool`.

Algumas variáveis:

- **`int`**: Armazena números inteiros.
- **`float`**: Armazena números decimais.
- **`string`**: Armazena sequências de caracteres.
- **`bool`**: Armazena valores booleanos (verdadeiro ou falso).



Variáveis

```
int pontuacao = 100;  
float velocidade = 5.5f;  
string nomePlayer = "José";  
bool jogadorPerdeu = false;
```

# 1.1 Variáveis

## Variáveis Não Primitivas

Eles são usadas para armazenar objetos e coleções de dados. Essas variáveis não armazenam os valores diretamente, mas sim a referência (ou endereço na memória) onde os dados estão armazenados. As operações com essas variáveis geralmente envolvem manipulação de objetos ou coleções de dados.

Algumas variáveis não primitivas:

- **Classes:** Classes são as blueprints da onde criamos os objetos.
- **Arrays:** Uma das maneiras de armazenar tipos de variáveis.

```
● ● ● Variáveis não primitivas

public class Veiculo
{
    public string placa;
    public int ano;
    public string montadora;
    public string modelo;
}

//o array pode ser declarado de duas maneiras:
int[] nums = new int[] {1, 3, 5, 7, 9};
string[] txts = new string[];
```

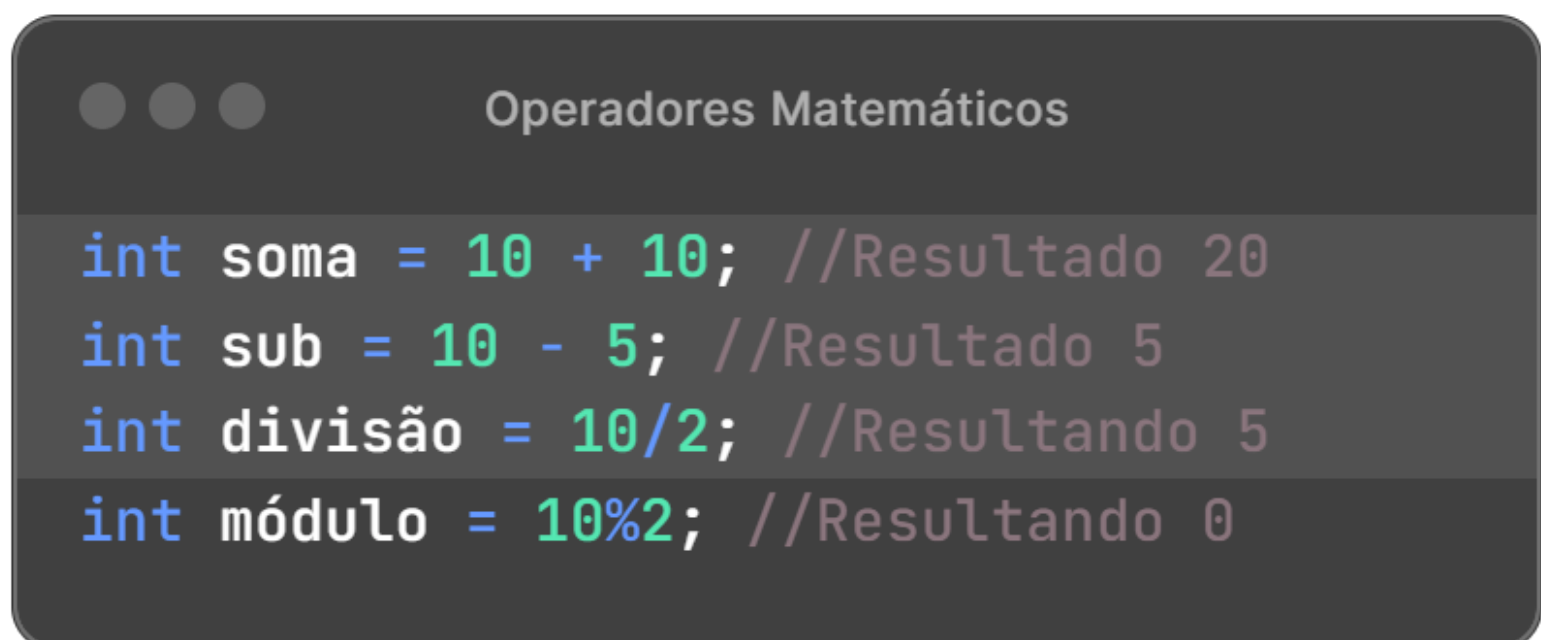
# 1.2.1 Operadores

## Operadores Aritméticos

A matemática é uma parte fundamental na programação, pois sem ela o jogador nunca iria tomar dano, por exemplo. E a sua aplicação é muito fácil.

Quais são os operadores matemáticos?

- **+** (**Adição**) : Soma os valores.
- **-** (**Subtração**) : Subtrai os valores.
- **\*** (**Multiplicação**) : Multiplica os valores.
- **/** (**Divisão**) : Divide os valores.
- **%** (**Módulo**) : Calcula o resto depois de dividir o seu primeiro operando pelo segundo.



```
int soma = 10 + 10; //Resultado 20
int sub = 10 - 5; //Resultado 5
int divisão = 10/2; //Resultando 5
int módulo = 10%2; //Resultando 0
```

# 1.2.3 Operadores

## Operadores de Atribuição

Quais são os operadores de atribuição?

- **= (Igual)** : O valor da esquerda vira o mesmo valor da direita.
- **+= (Adição)** : Soma os valores.
- **-= (Subtração)** : Subtrai os valores.
- **\*= (Multiplicação)** : Multiplica os valores.
- **/= (Divisão)** : Divide os valores.
- **%= (Módulo)** : Calcula o resto depois de dividir o seu primeiro operando pelo segundo.
- **&= (E bit a bit)** : Ele realiza uma operação AND bit a bit nos operandos e atribui o resultado ao operando da esquerda.
- **|= (Ou bit a bit)** : Ele realiza uma operação OR bit a bit nos operandos e atribui o resultado ao operando da esquerda.
- **^= (Exponencial)** : Multiplica o valor da esquerda por ele mesmo pela quantidade do valor da direita.
- **>>= (Deslocamento para a direita)** : Ele desloca os bits do operando da esquerda para a direita pelo número de posições especificado pelo operando da direita e atribui o resultado ao operando da esquerda.
- **<<= (Deslocamento para a esquerda)** : Ele desloca os bits do operando da esquerda para a esquerda pelo número de posições especificado pelo operando da direita e atribui o resultado ao operando da esquerda.

# 1.2.3 Operadores

## Operadores de atribuição

### Operadores de Atribuição

```
int valorA = 5; //Valor: 5
int valorB = 4; //Valor: 4
valorA += 5; //Valor: 10
valorB -= 3; //Valor: 1
valorA *= valorB; //Valor: 10
valorA /= 2; //Valor: 5
valorB %= 2; //Valor: 1
valorA &= 4; //Valor: 4
valorB |= 8; //Valor: 9
valorA ^= 2; //Valor: 16
valorA >>= 2; //Valor: 4
valorB <<= valorA; //Valor: 144
```



# 1.2.4 Operadores

## Operadores de Comparação

Os operadores de comparação são usados para comparar dois valores. Eles retornam um valor booleano: True se a comparação for verdadeira, False caso contrário.

Quais são os operadores de comparação?

- **== (Igual à)** : Verifica se os valores são iguais.
- **!= (Diferente de)** : Verifica se os valores são diferentes.
- **< (Menor que)** : Verifica se é menor.
- **> (Maior que)** : Verifica se é maior.
- **<= (Menor igual à)** : Verifica se é menor ou igual.
- **>= (Maior igual à)** : Verifica se é maior ou igual.



### Operadores de Comparação

```
bool igual = 10 == 6 //Retorna Falso
bool diferente = 10 != 6 //Retorna Verdadeiro
bool Menor = 10 < 6 //Retorna Falso
bool maior = 10 > 6 //Retorna Verdadeiro
bool menorIgual = 10 <= 6 //Retorna Falso
bool maiorIgual = 10 >= 6 //Retorna Verdadeiro
```

# 1.2.4 Operadores

## Operadores Lógicos

Os operadores lógicos são usados para combinar condições. Eles retornam um valor booleano: True se a condição lógica for satisfeita, False caso contrário.

Quais são os operadores lógicos?

- **&& (E)** : Duas booleanas devem retornar verdadeiro.
- **|| (Ou)** : Apenas uma booleana precisa retornar verdadeiro
- **! (Diferente de)** : Inverte o resultado da booleana.

### Operadores Lógicos

```
bool bool1 = 10 >= 6 && 50 < 100 //Retorna Verdadeiro
bool bool2 = 10 != 6 || 70 + 5 == 100 //Retorna Verdadeiro
bool bool3 = !(10 < 6) //Retorna Verdadeiro
```

# 1.3 Loops

## While Loop

O loop while percorre um bloco de código enquanto uma condição especificada for verdadeira



```
int i = 0;
while (i < 5)
{
    print("Loop Repetido: " + i);
    i++;
}
```

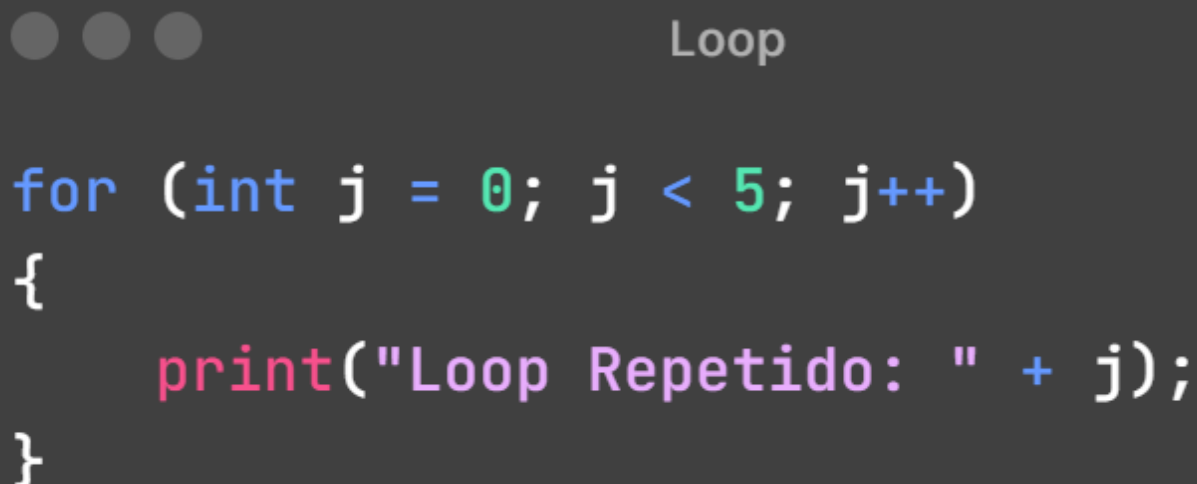
Neste exemplo, o loop continuará a ser executado enquanto a condição  $i < 5$  for verdadeira. Dentro do loop, `Console.WriteLine("Loop Repetido: " + i);` retornará o valor atual de `i`, e `i++` incrementa o valor de `i` em 1 em cada iteração. Quando `i` chega a 5, a condição torna-se falsa e o loop termina.



# 1.3 Loops

## For Loop

Quando sabe exatamente quantas vezes pretende repetir um bloco de código, utilize o ciclo for em vez de um ciclo while



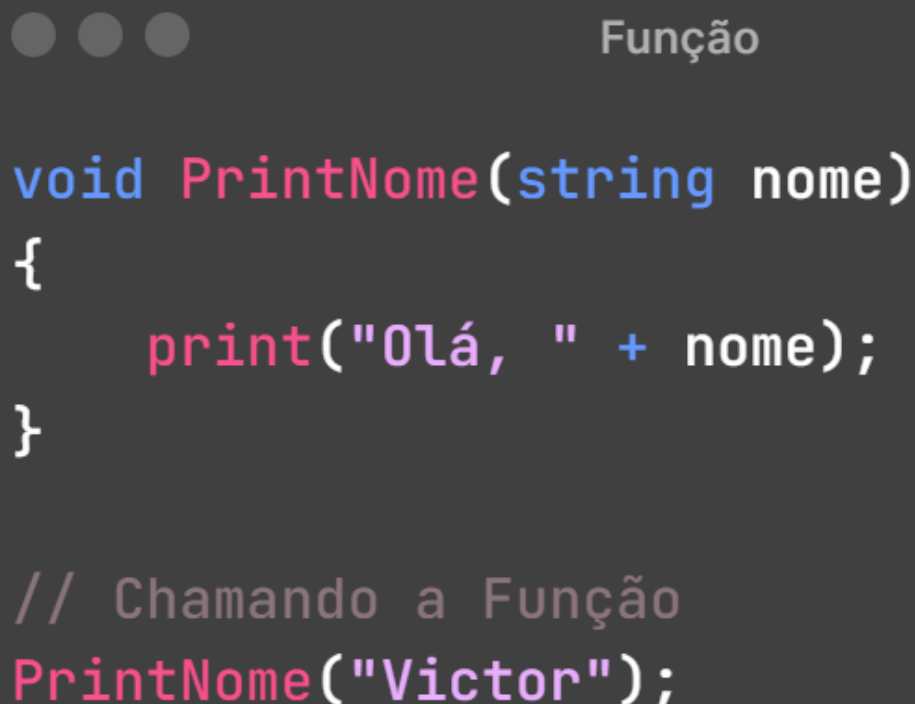
```
for (int j = 0; j < 5; j++)
{
    print("Loop Repetido: " + j);
}
```

Neste exemplo, o loop continuará a ser executado enquanto a condição  $i < 5$  for verdadeira. Dentro do loop, `Console.WriteLine("Loop Repetido: " + i);` retornará o valor atual de `i`, e `i++` incrementa o valor de `i` em 1 em cada iteração. Quando `i` chega a 5, a condição torna-se falsa e o loop termina.

# 1.4 Funções

## Void

As funções ajudam a organizar o seu código em blocos reutilizáveis, melhorando a legibilidade e a manutenção. Descubra como definir funções, passar parâmetros e retornar valores.



```
void PrintNome(string nome)
{
    print("Olá, " + nome);
}

// Chamando a Função
PrintNome("Victor");
```

Neste exemplo, void indica que a função não devolve nenhum valor. PrintNome é o nome da função e a cadeia de caracteres nome é o parâmetro que aceita. Dentro do corpo da função, print("(Olá, " + nome); retorna uma mensagem de saudação com o nome fornecido.

# 1.4 Funções

## Retornando Valores

As funções também podem devolver valores, permitindo calcular e fornecer resultados.



```
int Adicionar(int a, int b)
{
    return a + b;
}
print("Soma: " + Adicionar(5, 3)); //Resultado: Soma: 8
```

Neste exemplo, `int` indica que a função devolve um valor inteiro. `Adicionar` é o nome da função e aceita dois parâmetros `a` e `b`. No corpo da função, retorna `a + b`; calcula a soma de `a` e `b` e devolve o resultado.



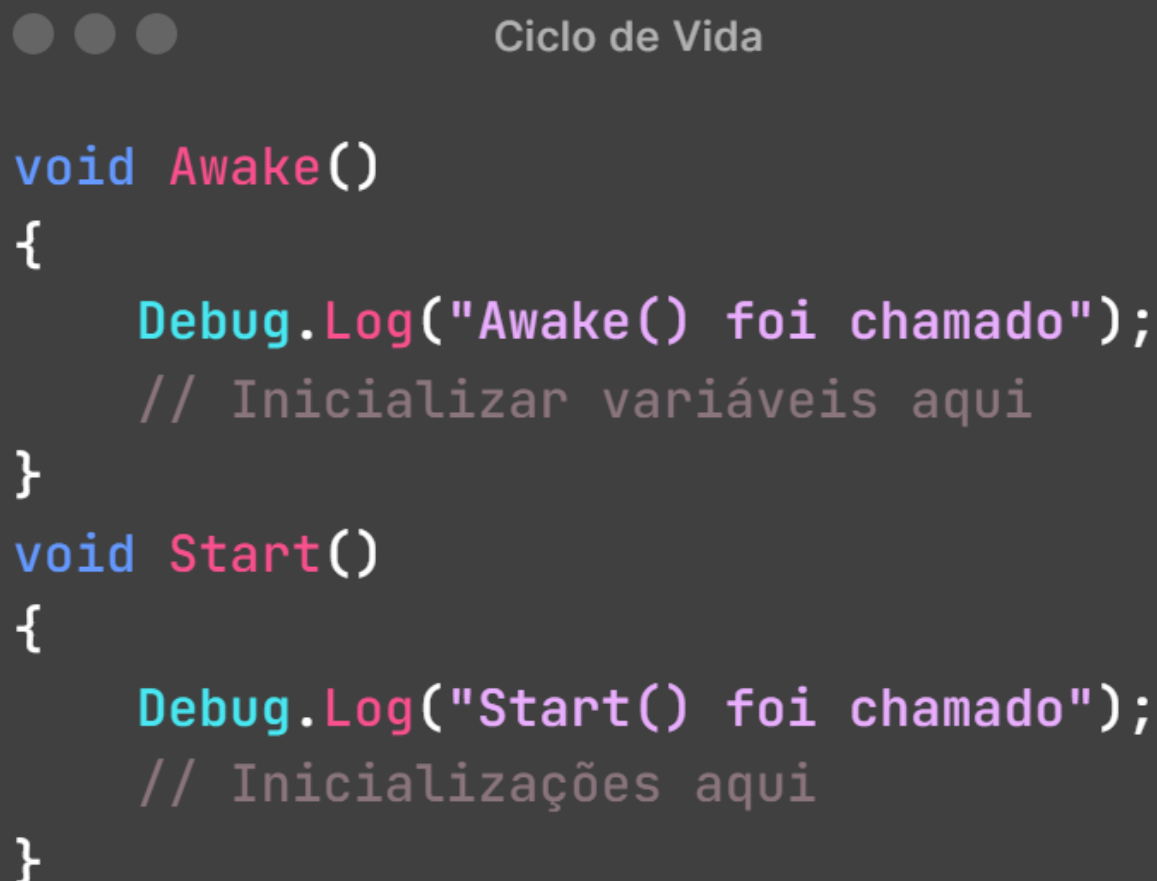
# Métodos do ciclo de vida da Unity

# 2.1 Ciclo de Vida

## Awake() e Start()

O método `Awake()` é chamado quando um `GameObject` é inicializado pela primeira vez. Ele é chamado antes do método `Start()` e é usado principalmente para inicializar variáveis ou estados de objetos.

O método `Start()` é chamado uma vez, no início do ciclo de vida do objeto, após o método `Awake()`. Ele é usado para inicializações que requerem que todos os objetos sejam inicializados primeiro.



```
void Awake()
{
    Debug.Log("Awake() foi chamado");
    // Inicializar variáveis aqui
}
void Start()
{
    Debug.Log("Start() foi chamado");
    // Inicializações aqui
}
```

## 2.2 Ciclo de Vida

### Update(), FixedUpdate() e LateUpdate()

O método `Update()` é chamado a cada quadro (frame) e é onde a maioria da lógica do jogo é colocada. Ele é usado para atualizar o estado do objeto com base no tempo decorrido desde o último quadro.

O método `FixedUpdate()` é chamado a cada intervalo de tempo fixo e é usado principalmente para física. É útil para cálculos que envolvem física, porque é chamado em intervalos regulares e independente da taxa de quadros.

O método `LateUpdate()` é chamado após o método `Update()` em todos os objetos. Ele é útil para ajustar a posição ou rotação de um objeto após todos os cálculos de atualização terem sido feitos.



## 2.2 Ciclo de Vida

Update(), FixedUpdate() e LateUpdate()

```
Ciclo de Vida

void Update()
{
    Debug.Log("Update() foi chamado");
    // Lógica de atualização aqui
}

void FixedUpdate()
{
    Debug.Log("FixedUpdate() foi chamado");
    // Lógica de física aqui
}

void LateUpdate()
{
    Debug.Log("LateUpdate() foi chamado");
    // Lógica de ajuste tardio aqui
}
```

## 2.3 Ciclo de Vida

### OnDestroy(), OnEnable() e OnDisable()

O método `OnDestroy()` é chamado quando o `GameObject` é destruído. É usado principalmente para liberar recursos ou fazer limpezas finais.

Esses métodos são chamados quando um `GameObject` é ativado ou desativado, respectivamente. Eles são úteis para realizar ações específicas quando um objeto se torna ativo ou inativo.

```
Ciclo de Vida

void OnDestroy()
{
    // Limpeza final aqui
}
void OnEnable()
{
    // Ações quando o GameObject é ativado
}
void OnDisable()
{
    // Ações quando o GameObject é desativado
}
```

# OS

## Eventos e CallBacks da Unity



# 3.1 Callbacks

## OnCollision()

Todo Callback tem três estágios: Entrada, Permanência e Saída. Ou seja, O método OnCollision tem:

- `OnCollisionEnter()`: O método `OnCollisionEnter()` é chamado quando este Collider/rigidbody entra em contato com outro Collider/rigidbody. É usado principalmente para detecção de colisões e interações físicas.
- `OnCollisionStay()`: O método `OnCollisionStay()` é chamado enquanto este Collider/rigidbody está em contato com outro Collider/rigidbody. Ele é chamado a cada quadro durante o tempo em que a colisão é mantida.
- `OnCollisionExit()`: O método `OnCollisionExit()` é chamado quando este Collider/rigidbody deixa de estar em contato com outro Collider/rigidbody. É usado para detectar quando uma colisão termina.

# 3.1 CallBacks

## OnCollision()

```
Callbacks

void OnCollisionEnter(Collision collision)
{
    Debug.Log("Colisão iniciada com " + collision.gameObject.name);
    // Lógica quando ocorre uma colisão
}
void OnCollisionStay(Collision collision)
{
    Debug.Log("Ainda tem colisão com " + collision.gameObject.name);
    // Lógica enquanto ainda tem colisão
}
void OnCollisionExit(Collision collision)
{
    Debug.Log("Colisão finalizada com " + collision.gameObject.name);
    // Lógica quando a colisão termina
}
```

## 3.2 Callbacks

### OnTrigger()

Com o evento OnTrigger() a regra se mantém, logo, ele tem três estágios.

- **OnTriggerEnter():** O método OnTriggerEnter() é chamado quando outro Collider/rigidbody entra no gatilho deste Collider. Ele é usado principalmente para detecção de gatilhos, como áreas de desencadeamento de eventos.
- **OnTriggerStay():** O método OnTriggerStay() é chamado enquanto outro Collider/rigidbody está dentro do gatilho deste Collider. Ele é chamado a cada quadro durante o tempo em que a colisão é mantida.
- **OnTriggerExit():** O método OnTriggerExit() é chamado quando outro Collider/rigidbody deixa o gatilho deste Collider. Ele é usado para detectar quando um objeto sai do gatilho.

# 3.2 Callbacks

## OnTrigger()

```
Callbacks

void OnTriggerEnter(Collider other)
{
    Debug.Log("Objeto entrou no gatilho: " + other.gameObject.name);
    // Lógica quando um objeto entra no gatilho
}

void OnTriggerStay(Collider other)
{
    Debug.Log("Objeto está dentro do gatilho: " + other.gameObject.name);
    // Lógica enquanto um objeto está dentro do gatilho
}

void OnTriggerExit(Collider other)
{
    Debug.Log("Objeto saiu do gatilho: " + other.gameObject.name);
    // Lógica quando um objeto sai do gatilho
}
```



# Agradecimientos

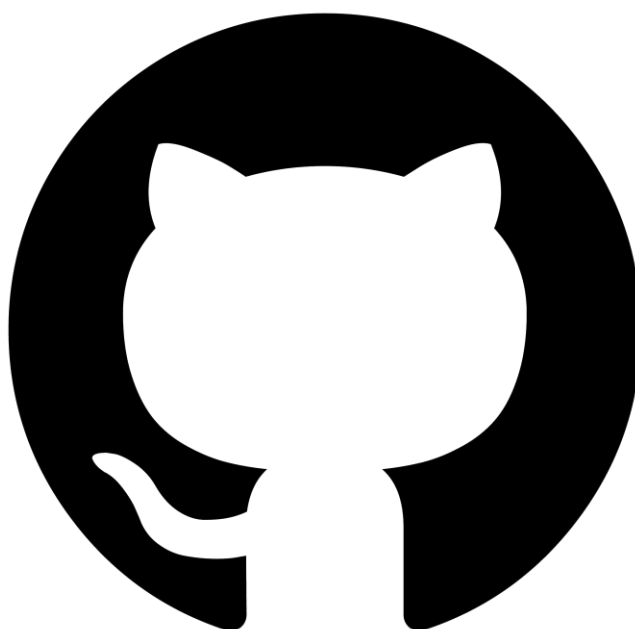
# Obrigado!

## Posfácio

Este eBook foi criado por uma inteligência artificial, e polido por um humano. Mostrando uma potencial abertura de novos horizontes para o ensino e aprendizado da programação com a parceria humano-IA.

Obrigado a todos que leram até aqui e boa sorte na jornada de programação para jogos. Que a curiosidade e a paixão pelo conhecimento continuem a ser suas companheiras constantes.

Link do GitHub do Autor



<https://github.com/Victor-Kanai>