**EXPLICA SCRIPT n. 01 - Mundo**

extends Node2D

# "extends Node2D" → diz que este script está ligado a um nó do tipo Node2D,

# que é usado para objetos em 2D.

# Aqui, nosso Mundo (a tela principal) é um Node2D.

@onready var cena\_asteroide = preload("res://Cenas/asteroide.tscn")

# @onready → só carrega essa variável quando a cena estiver pronta.

# var cena\_asteroide → é o nome da variável que guardará a cena do asteroide.

# preload("...") → carrega o arquivo "asteroide.tscn" antes do jogo começar,

# deixando pronto para ser usado.

# Isso significa: sempre que quisermos criar um asteroide, usamos esta variável.

@onready var cena\_inimigo = preload("res://Cenas/inimigo.tscn")

# Mesmo que o de cima, mas para os inimigos.

var vidas := 3

# Cria a variável "vidas" e já começa com 3.

# Essas são as 3 chances do jogador, como "corações" num jogo.

var score := 0

# Cria a variável "score" (pontuação) e começa em 0.

# Vai aumentando quando destruímos inimigos.

func \_ready():

# Esta função é chamada quando o Mundo aparece na tela.

randomize()

# randomize() → deixa o jogo "aleatório", para que asteroides e inimigos

# não apareçam sempre nos mesmos lugares.

$HUD.connect("start\_game", Callable(self, "iniciar\_jogo"))

# $HUD → o $ serve para pegar um nó filho chamado HUD.

# connect() → conecta o sinal "start\_game" do HUD a uma função aqui no Mundo.

# Ou seja: quando o botão START do HUD for clicado, a função iniciar\_jogo() será chamada.

$HUD.update\_lives(vidas)

# Chama a função update\_lives() do HUD para mostrar as vidas na tela.

$HUD.update\_score(score)

# Chama a função update\_score() do HUD para mostrar o placar na tela.

func iniciar\_jogo():

# Esta função reinicia o jogo quando apertamos o botão Start.

vidas = 3

score = 0

# Reseta as vidas e a pontuação.

$HUD.update\_lives(vidas)

$HUD.update\_score(score)

# Atualiza a tela com as novas vidas e o novo score.

if not has\_node("Jogador"):

# "if not has\_node" → se o Mundo não tiver o nó Jogador...

var cena\_jogador = preload("res://Cenas/jogador.tscn")

# carrega a cena do jogador.

var j = cena\_jogador.instantiate()

# cria (instancia) o jogador a partir da cena carregada.

j.position = Vector2(240, 400)

# coloca a nave em uma posição inicial (meio da tela).

add\_child(j)

# adiciona o jogador dentro do Mundo.

$Timer\_Asteroide.start()

$Timer\_Inimigo.start()

# Liga os dois timers, que vão criar asteroides e inimigos com o tempo.

func \_on\_timer\_asteroide\_timeout():

# Esta função roda sempre que o Timer\_Asteroide chega a 0.

var a = cena\_asteroide.instantiate()

# Cria um novo asteroide.

a.position = Vector2(randi\_range(30, 480), -20)

# Coloca ele em posição aleatória no topo da tela.

add\_child(a)

# Adiciona o asteroide no Mundo.

func \_on\_timer\_inimigo\_timeout():

# Mesma coisa, mas para inimigos.

var i = cena\_inimigo.instantiate()

i.position = Vector2(randi\_range(30, 480), -20)

add\_child(i)

func add\_score(valor: int) -> void:

# Função para somar pontos.

score += valor

$HUD.update\_score(score)

# Atualiza o HUD com o novo placar.

func perder\_vida(qtd := 1):

# Função que tira vidas quando o jogador é atingido.

if vidas <= 0:

return

# Se já acabou o jogo, não faz mais nada.

vidas -= qtd

# Diminui a quantidade de vidas.

$HUD.update\_lives(vidas)

# Atualiza o HUD.

if vidas <= 0:

game\_over()

# Se as vidas acabaram, chama o fim de jogo.

func game\_over():

# Função que finaliza o jogo.

$Timer\_Asteroide.stop()

print("para o spawn 1")

$Timer\_Inimigo.stop()

print("para o spawn 2")

# Para os dois timers → nada mais aparece na tela.

for n in get\_tree().get\_nodes\_in\_group("Inimigos"):

n.queue\_free()

for n in get\_tree().get\_nodes\_in\_group("Asteroides"):

n.queue\_free()

for n in get\_tree().get\_nodes\_in\_group("Tiros"):

n.queue\_free()

# Limpa todos os inimigos, asteroides e tiros que estavam na tela.

if has\_node("Jogador"):

$Jogador.queue\_free()

# Remove também o jogador (nave).

$HUD.show\_game\_over()

# Pede ao HUD para mostrar a mensagem de "Game Over".

* **$HUD.show\_game\_over()** → $ acessa o nó HUD, e show\_game\_over() é uma função que está no script do HUD, que mostra a mensagem “Game Over” piscando.
* **@onready var cena\_inimigo = preload("res://Cenas/inimigo.tscn")** → cria uma variável que vai carregar a cena do inimigo. O @onready espera a cena estar pronta, e o preload já deixa o arquivo carregado na memória.
* **var vidas := 3** → é só uma variável criada antes do jogo começar, para guardar o número de vidas.

Vamos dissecar essa linha passo a passo, como se fosse uma “receita de bolo” para as crianças 👇

* $HUD.connect("start\_game", Callable(self, "iniciar\_jogo"))

**1. O $HUD**

* O **$** em Godot é um atalho para procurar nós dentro da cena.
* Aqui, $HUD significa **“pegue o nó filho chamado HUD”**.
* Esse HUD é a interface do jogo (vidas, score, botão Start, Game Over).

**2. O .connect(...)**

* connect é um comando que **liga um sinal a uma função**.
* Um **sinal** em Godot é como um “aviso” ou “grito”: *“ei, algo aconteceu!”*.
* Quando o HUD emitir o sinal chamado "start\_game", essa ligação vai garantir que uma função seja chamada.

**3. O "start\_game"**

* É o **nome do sinal** que o HUD pode emitir.
* Esse sinal foi **criado no script do HUD** com a linha signal start\_game.
* Ele é disparado quando o jogador clica no botão **Start**.

**4. O Callable(self, "iniciar\_jogo")**

* Callable(...) é uma forma de dizer **qual função deve ser chamada** quando o sinal acontecer.
* self significa “este mesmo objeto” (no caso, o Mundo).
* "iniciar\_jogo" é o nome da função dentro deste script que será chamada.

Ou seja:  
👉 Quando o HUD emitir o sinal "start\_game", a função iniciar\_jogo() do **Mundo** será executada.

**Comparação simples 🎮**

Imagine um **controle de videogame**:

* O HUD é o **botão Start**.
* O sinal "start\_game" é o **clique do botão**.
* O connect é o **fio** que liga o botão a alguma ação.
* O Callable(self, "iniciar\_jogo") é o **comando que vai ser executado**: “quando o botão Start for apertado → chame a função iniciar\_jogo() no Mundo”.

Entendi o que a randomize() faz. Mas gostaria de saber que tipo de saída tem a randomize? É uma posição x,y aleatória? Dentro de que range?

👉 **randomize() não gera nenhum número aleatório por si só.**  
O que ela faz é **inicializar o gerador de números aleatórios do Godot**.

**Como funciona**

* Todo jogo precisa de um “gerador de aleatórios”.
* Se não chamarmos randomize(), o Godot sempre começa com a mesma “semente” (seed).
* Resultado: os números aleatórios saem iguais toda vez que rodamos o jogo → os inimigos e asteroides apareceriam sempre nos mesmos pontos.
* Quando chamamos randomize(), o Godot usa o **relógio do sistema** como semente, garantindo que cada vez o jogo gere valores diferentes.

**Então, qual é a saída?**

* **randomize() não retorna nada** (tipo void).
* Ela só prepara o gerador.
* Para realmente pegar um número aleatório, usamos funções como:
  + randi() → inteiro bem grande (0 até 2³²).
  + randf() → número decimal entre **0 e 1**.
  + randi\_range(min, max) → inteiro dentro de um intervalo, ex: randi\_range(30, 480) dá valores de 30 até 480.
  + randf\_range(min, max) → decimal dentro de um intervalo.

**Exemplo do nosso jogo**

a.position = Vector2(randi\_range(30, 480), -20)

* Aqui sim é usada a **saída aleatória**:
  + randi\_range(30, 480) → dá um valor de **x** aleatório entre 30 e 480.
  + -20 → y fixo (no topo da tela).
* Resultado: o asteroide aparece sempre em um ponto **aleatório na largura da tela**, mas sempre lá em cima.

📌 Resumindo:

* randomize() → só prepara.
* Quem gera o número mesmo são randi\_range() ou randf\_range().

Explique de forma simples o que fazem as funções/métodos randi\_range() ou randf\_range() . São funções ou métodos?

**📌 São funções ou métodos?**

No **Godot** (e em programação em geral):

* **Função** = um pedaço de código que pode ser chamado para fazer algo.
* **Método** = é quase a mesma coisa, mas “grudado” em um objeto.

👉 Como randi\_range() e randf\_range() não estão dentro de um objeto, e sim na **biblioteca global do Godot**, podemos chamá-los de **funções globais**.

**📌 O que fazem?**

* **randi\_range(min, max)**
  + Gera um **número inteiro aleatório** entre os valores min e max.
  + Exemplo: randi\_range(1, 6) → pode sair **1, 2, 3, 4, 5 ou 6** (como jogar um dado 🎲).
* **randf\_range(min, max)**
  + Gera um **número decimal aleatório** entre min e max.
  + Exemplo: randf\_range(0, 1) → pode sair **0.23, 0.78, 0.01…** qualquer valor entre 0 e 1.

**📚 Para as crianças**

* randi\_range() = sorteia um **número inteiro** (sem vírgula).
* randf\_range() = sorteia um **número com vírgula** (mais preciso).

⚡ Analogia:

* randi\_range(1, 10) = como escolher uma **carta numerada** de 1 a 10.
* randf\_range(1, 10) = como apertar o botão stop de um **cronômetro digital** o que pode parar em qualquer lugar entre 1 e 10 segundos.

E se eu usasse só **randi\_range()** ou **randf\_range()** sem o randomize() no início, não funcionaria?

Boa sacada 👌

Sim, **funcionaria** mesmo sem randomize().  
O problema é **o padrão repetido**:

* O Godot tem um **gerador de números pseudoaleatórios**.
* “Pseudo” porque ele sempre começa com a mesma **semente (seed)** quando você abre o jogo.
* Então, se você não chamar randomize(), os números que randi\_range() ou randf\_range() geram serão **sempre iguais em toda partida**.

**Exemplo**

🔹 Sem randomize():

* Primeira vez que roda → inimigos caem nas posições 120, 350, 90…
* Se fechar e abrir de novo → eles caem **nos mesmos lugares**.
* O jogo fica “programado” em vez de aleatório.

🔹 Com randomize():

* A semente muda (usa o relógio do computador).
* Cada vez que o jogo começa → a sequência de números muda.
* Inimigos e asteroides caem em lugares diferentes, deixando o jogo imprevisível.

📌 Então:

* **randi\_range() e randf\_range() funcionam sozinhos**, mas sem randomize() o jogo parece um **filme repetido**.
* Com randomize(), ele vira um **jogo de verdade**, porque sempre surpreende.

Vamos montar um **mini-experimento** para mostrar às crianças.  
Esse código imprime 5 números aleatórios no console — primeiro **sem randomize()**, depois **com randomize()**.

extends Node

func \_ready():

seed(100) # seed no Godot4 gera números aleatórios fixos

print("🔴 SEM randomize() ------------------")

for i in range(5):

print(randi\_range(1, 100)) # gera de 1 a 100

# Agora reinicia o jogo e verá que sempre repete os mesmos 5 números

print("🟢 COM randomize() ------------------")

randomize() # inicializa o gerador com base no relógio do PC

for i in range(5):

print(randi\_range(1, 100)) # gera de 1 a 100

**O que acontece**

* **Sem randomize()** → toda vez que você rodar o jogo, os 5 números saem **iguais** (exemplo: 42, 17, 93, 5, 88).
* **Com randomize()** → os 5 números mudam a cada vez que o jogo começa (exemplo: 23, 91, 12, 77, 64… e da próxima vez serão outros).
* **seed(valor)** = sempre gera a mesma sequência de “aleatórios”.  
  👉 Útil para testes, porque dá para repetir sempre o mesmo cenário.
* **randomize()** = pega a hora do computador como semente.  
  👉 Cada vez que rodar o jogo, os números mudam, deixando tudo diferente.

⚡ No **Godot 4**, mesmo sem randomize(), a engine já define uma semente baseada no tempo, então os números variam. Mas se você usar seed(100), volta ao comportamento **fixo**.

Então, significa que o atual script que temos no mundo.gd não precisaria de randomize(), pois no Godot 4 os números serão os mesmos somente se vc colocar a seed.

Exatamente ✅

No **Godot 4**, o script do mundo.gd funcionaria da mesma forma **mesmo sem o randomize()**:

* Os asteroides e inimigos já vão aparecer em posições diferentes a cada vez que você rodar o jogo.
* O randomize() está lá por tradição (era **obrigatório no Godot 3** para não repetir a mesma sequência).

👉 Hoje em dia, ele só faz sentido se você quiser **forçar uma semente aleatória nova**, mas como o Godot 4 já faz isso sozinho, é redundante.

📌 **Resumo prático:**

* **Godot 3** → precisa do randomize().
* **Godot 4** → já vem “randomizado” de fábrica, só repete se você usar seed(...).

Por que a variável cena jogador deve ser instanciada dentro de outra variável e não chamada diretamente já que está dentro da var cena\_jogador

Boa pergunta 👏 você pegou um ponto bem importante de GDScript: **a diferença entre preload, recurso e instância**.

**1. O que é preload("res://Cenas/jogador.tscn")?**

* Isso **carrega o arquivo da cena** (o recurso .tscn) e guarda dentro de uma variável.
* Mas isso ainda **não é o jogador na tela**, é só o **“molde”** (como uma receita de bolo).

Exemplo:  
👉 cena\_jogador é a **receita do bolo**.

**2. O que é .instantiate()?**

* instantiate() cria uma **cópia real** da cena na memória.
* Só depois disso podemos colocar o objeto no jogo com add\_child().

Exemplo:  
👉 j = cena\_jogador.instantiate() é o **bolo pronto** feito a partir da receita.

**3. Por que guardar na variável j?**

* Porque precisamos configurar coisas nessa cópia antes de colocar no jogo.
* Aqui, definimos a posição inicial:
* j.position = Vector2(240, 400)
* Se chamássemos direto assim:
* add\_child(cena\_jogador.instantiate())

também funcionaria ✅, mas não teríamos como ajustar a posição, velocidade, ou outros parâmetros antes de adicionar.

**4. Vantagem de separar em j**

* Dá mais clareza para o aluno (primeiro cria, depois configura, depois adiciona).
* Permite modificar o objeto **antes de entrar na cena**.
* Torna o código mais fácil de entender e expandir (se amanhã quiser mudar cor, escudo, etc., já temos a variável pronta).

Ótimo! Vou mostrar as **duas formas** de escrever esse trecho, e depois explicar qual é a diferença. Assim você pode apresentar em aula como “modo rápido” e “modo detalhado”.

**🔹 Forma simples (rápida)**

if not has\_node("Jogador"):

add\_child(preload("res://Cenas/jogador.tscn").instantiate())

👉 O que acontece aqui:

* Carrega a cena do jogador (preload).
* Instancia (cria a cópia).
* Já coloca direto no Mundo (add\_child).

⚡ É curto, mas você não consegue configurar nada antes de adicionar.

**🔹 Forma completa (detalhada, com variável)**

if not has\_node("Jogador"):

var cena\_jogador = preload("res://Cenas/jogador.tscn")

# receita (o molde da nave)

var j = cena\_jogador.instantiate()

# bolo pronto a partir da receita

j.position = Vector2(240, 400)

# ajusta a posição inicial antes de mostrar na tela

add\_child(j)

# coloca o jogador no Mundo

👉 Aqui temos controle total:

* Podemos mudar posição, cor, velocidade, vidas extras, etc.
* O código fica mais didático para entender os passos.

**📌 Como explicar para as crianças**

* **Versão simples**: “Dá pra colocar o jogador direto na tela sem mexer nele antes.”
* **Versão completa**: “É como segurar o boneco na mão antes de colocar no palco. Assim você pode arrumar ele antes do público ver.” 🎭

Então, significa que se eu chamo a cena direto, o jogador sempre aparecerá na posição em que ele foi colocado na cena original, mas com a var j é possível mudar a posição e/ou outros parâmetros como velocidade?

💯 Exatamente, você entendeu certinho!

👉 Quando você **instancia direto** a cena (preload(...).instantiate()):

* O objeto aparece com **todas as configurações originais** que você salvou no arquivo jogador.tscn.
* Isso inclui posição, escala, rotação, velocidade exportada, etc.

👉 Quando você cria uma variável (var j = ...):

* Você tem a chance de **alterar parâmetros antes de colocar na tela**.
* Exemplo:
* j.position = Vector2(240, 400) # mudar posição
* j.velocidade = 500 # mudar velocidade exportada
* j.scale = Vector2(2, 2) # deixar maior
* j.rotation\_degrees = 45 # girar
* add\_child(j)

**📌 Resumindo**

* **Instanciar direto** = usa sempre os valores da cena original.
* **Guardar na variável (j)** = você pode “personalizar” antes de aparecer no jogo.

⚡ Essa técnica é muito útil quando você precisa criar **vários objetos iguais, mas com diferenças** (ex.: inimigos em posições diferentes, ou tiros com velocidades diferentes).

**Como isso se encaixa na POO – Programação Orientada a Objetos?**

* **A cena (jogador.tscn)** é como uma **classe** → ela define como o objeto deve ser (sprite, colisão, velocidade padrão, etc.).
* **instantiate()** cria uma **instância da classe** → é o objeto real que entra no jogo.
* **A variável j** é a **referência a essa instância**, que você pode configurar (mudar posição, velocidade, escala).

👉 E como na POO:

* Você pode criar várias instâncias do mesmo “molde” (classe).
* Cada instância pode ter valores diferentes em suas propriedades.

**Exemplo bem simples**

Imagina uma **classe “Nave”** 🚗

* A classe diz: *toda nave tem asas, cor, velocidade*.
* Mas você pode criar:
  + Nave 1 = vermelha, 100 km/h
  + Nave 2 = azul, 80 km/h
  + Nave 3 = verde, 120 km/h

No Godot:

* preload("jogador.tscn") = o **molde da nave**.
* instantiate() = faz nascer uma **nave real**.
* Variável nave1, nave2, nave3 → cada nave com seus ajustes próprios.

📌 Resumindo para aula:

O Godot usa cenas como se fossem **classes**.  
Quando instanciamos (.instantiate()), criamos **objetos reais** que podem ser diferentes uns dos outros, mesmo vindo do mesmo molde.

### Exemplo no código

var cena\_nave = preload("res://Cenas/jogador.tscn")

# Criando 3 naves diferentes a partir do mesmo molde

var nave1 = cena\_nave.instantiate()

nave1.position = Vector2(100, 400)

nave1.velocidade = 300

add\_child(nave1)

var nave2 = cena\_nave.instantiate()

nave2.position = Vector2(240, 400)

nave2.velocidade = 500

add\_child(nave2)

var nave3 = cena\_nave.instantiate()

nave3.position = Vector2(380, 400)

nave3.velocidade = 200

add\_child(nave3)